N° d'ordre : 61/2016-C/INF

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université des Sciences et de la Technologie Houari Boumediène

Faculté d'électronique et informatique

THESE

Présentée pour l'obtention du **diplôme** de **DOCTORAT 3eme Cycle**

En : INFORMATIQUE

Spécialité : Intelligence artificielle

Par : **Chegrane Ibrahim**

Sujet

La recherche approchée de motifs : théorie et applications

Soutenue publiquement, le 07 /12/ 2016 , devant le jury composé de :

Mme.	Aicha Aissani-Mokhtari	Prof	USTHB/FEI	Président
Mme.	Nacéra Bensaou	MCA	USTHB/FEI	Directrice de thèse
M.	Thierry Lecroq	Prof	U. Rouen-Normandie	Examinateur
M.	Riadh BabaAli	Prof	USTHB/FEI	Examinateur
M.	Abdelmajid Boukra	Prof	USTHB/FEI	Examinateur
M.	Djamal Belazzougui	M.R	CERIST	Invité

بسم الله الرحمن الرحيم

قال **الخوارزمي** في مقدمة كتابه الشهير كتاب الجبر و المقابلة:

ولم تزل العلماء في الأزمنة الخالية والأمم الماضية يكتبون الكتب، مما يُصنفون من صنوف العلم ووجوه الحكمة، نظرًا لمن بعدهم واكتسابًا للأجر، بقدر الطاقة ورجاءً أن يلحقهم من أجر ذلك وذخره وذكره ويبقى لهم من لسان الصدق ما صغُر في جنبه كثيرٌ مما كانوا يتكلفونه من المؤونة ويحملونه على أنفسهم من المشقة في كشف أسرار العلم وغامضه.

1. إما رجل سبق إلى ما لم يكن مُستخرجًا قبله، فورثّه من بعده؛
2. و إما رجل شرح مما أبقى الأولون ما كان مستغلقًا، فأوضح طريقه وسهل مسلكه وقرّب مأخذه ؛
3. وإما رجل وجد في بعض الكتب خللاً فلمّ شعْثَه وأقام أوده وأحسن الظن بصاحبه غير زادٍ عليه ولا مُفتخرٍ بذلك من فعل نفسه.

AL-Khawarizmi a dit dans l'introduction de son fameux livre d'algèbre:

Les savants du temps passé et des nations d'antan continuaient d'écrire les livres qu'ils composaient dans les diverses sortes de la science et les divers aspects de la sagesse, en vue de ceux qui leur succéderaient et en prévision d'une récompense, richesse ou renommée, et garder la langue de la vérité devant laquelle s'amenuisent la plupart des ressources engagées et la difficulté qu'ils ont endurée pour étudier les secrets de la science et ce qu'elle renferme d'obscur

1. Ou bien c'est un homme qui est parvenu le premier à découvrir ce qui n'était pas découvert avant lui et l'a légué après lui ;

2. Ou bien un homme qui a expliqué ce que ses prédécesseurs avaient laissé inaccessible, pour en éclairer la méthode, en aplanir la voie et en rapprocher l'accès;

3. Ou bien un homme qui a trouvé une faille dans certains livres, et qui a alors rassemblé ce qui était éparpillé, a redressé sa stature en ayant bonne opinion de son auteur, sans renchérir sur lui et sans s'enorgueillir de ce que lui-même a réalisé.

Al-Khwarizmi, Le commencement de l'algèbre,
éd. Roshdi Rashed

Remerciements

Le Prophète Mohammed (que la paix et le salut soient sur lui) a dit : "***Ne remercie pas Dieu, celui qui ne remercie pas les gens.***" [1]

"لا يشكر الله من لا يشكر الناس"

En tout premier lieu, je remercie Dieu (Allah), Alhamdulillah, pour la force qu'il m'a donnée d'arriver au bout de cette thèse, de m'avoir entouré de personnes formidables qui m'ont aidé d'une manière ou d'une autre, tout au long de ce travail. Merci à Dieu (Allah) pour tout ce qu'Il m'a accordé.

"الحمد لله حمدا كثيرا طيبا مباركا فيه"

Je remercie très chaleureusement ma directrice de thèse, madame Nacéra Bensaou, d'abord d'avoir accepté de m'encadrer. Je la remercie pour ces nombreux conseils, pour son soutien et son encouragement durant toute la durée de la thèse. Je la remercie d'avoir partagé son propre bureau (335) avec moi pour que je puisse travailler. Je la remercie de m'avoir mis en contact avec des chercheurs spécialistes du domaine, et de m'avoir encouragé fortement à travailler et à publier avec eux. Je la remercie de m'avoir fait confiance, et pour la liberté qu'elle m'a accordée. Mille Mille Merci Madame Bensaou ☺.

Je remercie profondément monsieur Mathieu Raffinot de m'avoir accueilli au laboratoire LIAFA (université de Paris 7) à plusieurs reprises (la première fois pour un mois, la deuxième fois pour trois mois, et la troisième fois pour une simple visite), de m'avoir pris en charge en stage de deux mois, de m'avoir hébergé à Paris dans le domicile de ses parents pendant 8 jours. Je remercie aussi beaucoup les parents de Mathieu Raffinot pour toute leur gentillesse durant cette période. Merci pour tous tes conseils et tes directives, (Merci aussi pour le dîner du sushi ☺). Merci Mathieu, tu as été mon deuxième encadreur de thèse.

Djamal Belazzougui, mon co-encadreur, mon exemple, mon ami. Je ne pourrais jamais te remercier assez. Dès les premiers jours de notre rencontre, tu n'a pas cessé de me guider, de m'aider, de me donner des directives et des conseils. Merci pour tout le temps et séances de travail que tu m'a accordé au LIAFA durant mon premier stage, mais aussi à l'USTHB, à chaque fois que tu rentres à Alger et que tu viens faire une séance de travail avec moi,

1. Rapporté par Ahmad, Boukhari dans Al-Adab al-Moufrad

et jusqu'à maintenant où tu me reçois et m'accordes du temps au CERIST pour m'aider malgré ton importante charge de travail. Je suis très chanceux de ma rencontre avec toi.

Je remercie également très chaleureusement madame C.Ighilaza mon enseignante d'algorithmique, mon encadreur du mémoire de Licence et de Master, pour l'aide et les conseils qu'elle m'a donnés, pour m'avoir aidé dans mon travail, et pour sa participation à la correction de la thèse.

Je remercie le professeur Slimane Larabi et le Professeur Aicha Aissani-Mokhtari d'avoir accepté d'expertiser mes travaux.

Je remercie le professeur Aicha Aissani-Mokhtari d'avoir accepté de présider le jury de cette thèse.

Je remercie également le professeur Thierry Lecroq d'avoir accepté de participer à ce jury en tant qu'examinateur et de venir de loin pour participer à ce jury.

Je remercie le professeur Riadh BabaAli d'avoir accepté de participer au jury de cette thèse.

Je remercie le professeur Abdelmajid Boukra d'avoir accepté de participer au jury de cette thèse.

Mes remerciements vont aussi à Athmane Seghier, Meriem Beloucif et Aïcha Botorh pour leurs collaborations.

Mes remerciements vont aussi à tous mes amis, nombreux pour être cités individuellement, et qui se reconnaitront certainement. Je pense à mes amis de l'université USTHB, mes amis de la cité universitaire RUB 1 et RUB 3, mes amis les doctorant(e)s de la salle Poste Graduation du département informatique, mes amis du bureau 335 de madame Bensaou, mes amis du club informatique Micro-club et mes amis du sport.

Merci à mon ami et mon prof de sport Nourdine Zirara (Hamdane), pour son soutien moral et matériel, et pour son encouragement.

Je tiens à exprimer toute ma gratitude à ma famille pour leur soutien moral et matériel et sans qui je ne serais jamais allé aussi loin dans mes études.

Mon épouse Nacera, je te remercie pour ta présence, ton soutien, tes prières, tes pensées, tes encouragements et ta patience qui m'ont été indispensables.

Que tous ceux qui m'ont aidé et qui ont contribué de près ou de loin à ma formation trouvent ici le témoignage de ma sincère gratitude.

Mes remerciements vont également à mon pays l'Algérie qui, malgré ses insuffisances et ses problèmes, m'a permis d'arriver à ce niveau et de terminer mes études gratuitement.

Résumé

La recherche approchée de motifs est un problème fondamental et récurrent qui se pose dans la plupart des domaines informatiques. Ce problème peut être défini de la manière suivante :

> *Soit $D = \{x_1, x_2, \ldots x_d\}$ un ensemble de d mots définis sur un alphabet Σ, soit q une requête définie aussi sur Σ, et soit k un entier positive.*
> *On veut construire une structure de données pour D capable de répondre à la requête suivante : trouver tous les mots de D distants d'au plus k erreurs de q.*

Dans cette thèse nous étudions les méthodes de la recherche approchée dans les dictionnaires, les textes et les index, pour proposer des méthodes pratiques qui résolvent de façon efficace ce problème. Nous explorons ce problème dans trois directions complémentaires :

1) La recherche approchée dans un dictionnaire. Nous proposons deux solutions à ce problème, la première utilise les tableaux de hachage pour $k \geq 2$, la deuxième, utilise le Trie et le Trie inversé et se restreint à $k = 1$. Les deux solutions sont applicables, sans perte de performances, à la recherche approchée dans un texte.

2) La recherche approchée pour le problème de *l'auto-complétion* consiste à trouver tous les suffixes d'un préfixe contenant éventuellement des erreurs. Nous apportons une nouvelle solution meilleure, en pratique, que toutes les solutions précédentes.

3) Le problème de l'alignement de séquences biologiques peut être interprété comme un problème de recherche approchée. Nous proposons à ce problème une solution pour l'alignement par pairs et pour l'alignement multiple.

Tous les résultats obtenus montrent que nos algorithmes donnent les meilleurs performances sur des ensembles de données pratiques. Toutes nos méthodes sont proposées comme des bibliothèques et sont publiées en ligne.

ملخص

البحث النصي التقريبي هو إشكالية أساسية ومتكررة في أغلب ميادين الإعلام الألي. يمكن تعريف هذه الإشكالية على النحو التالي :

ليكن م={ك1، ك2، ك3،...، ك ن} مجموعة من (ن) كلمة، معرفة على حروف الأبجدية ∑، ليكن (س) كلمة استعلام معرفة أيضا على ∑، وليكن (خ) عدد صحيح موجب.

نريد أن نقوم بتصميم هيكل بيانات على (م) قادر على الإجابة على المطلب التالي: إيجاد كل الكلمات في (م) التي تختلف عن كلمة الاستعلام (س) على الأكثر ب (خ) خطأ.

في هذه الأطروحة تم دراسة مجموعة من الطرق للبحث النصي التقريبي في القواميس، والنصوص، وهياكل البيانات، بهدف اقتراح طرق عملية للمساهمة في حل هذه الإشكالية بفعالية.

نتوغل في حل هذه الإشكالية في ثلاثة محاور متكاملة:

1) البحث النصي التقريبي في القاموس. اقترحنا حلين لهذه الإشكالية، في الحل الأول نستخدم جداول التجزئة مع خ≤2 ، الحل الثاني يستخدم هياكل البيانات التراي والتراي العكسي، ويقتصر على (خ = 1). الحلان قابلان للتكيف، من دون خسارة في الأداء، في البحث النصي التقريبي في النصوص.

2) البحث النصي التقريبي في مجال الإكمال التلقائي، أين نهدف إلى العثور على جميع اللواحق لبادئة معينة التي قد تحتوي على أخطاء إملائية. قمنا في هذا المحور بتقديم حلا خاصا جديدا أفضل من الناحية العملية من كل الحلول المقترحة السابقة.

3) إشكالية محاذاة السلاسل البيولوجية، يمكن أن ترى على أنها إشكالية البحث النصي التقريبي. نقترح في هذا المحور حلا لمحاذاة السلاسل الثنائية والمتعددة.

أظهرت جميع النتائج أن خوارزمياتنا، تعطي الأداء الأفضل على مجموعات من البيانات العملية. كل خوارزمياتنا مقترحة على شكل مكتبات، و هي منشورة على الشبكة العنكبوتية.

Abstract

The approximate string matching is a fundamental and recurrent problem that arises in most computer science fields. This problem can be defined as follows :

> *Let $D = \{x_1, x_2, \ldots x_d\}$ be a set of d words defined on an alphabet Σ, let q be a query defined also on Σ, and let k be a positive integer.*
> *We want to build a data structure on D capable of answering the following query : find all words in D that are at most different from the query word q with k errors.*

In this thesis, we study the approximate string matching methods in dictionaries, texts, and indexes, to propose practical methods that solve this problem efficiently. We explore this problem in three complementary directions :

1) The approximate string matching in the dictionary. We propose two solutions to this problem, the first one uses hash tables for $k \geq 2$, the second uses the Trie and reverse Trie, and it is restricted to (k = 1). The two solutions are adaptable, without loss of performance, to the approximate string matching in a text.

2) The approximate string matching for *autocompletion*, which is, find all suffixes of a given prefix that may contain errors. We give a new solution better in practice than all the previous proposed solutions.

3) The problem of the alignment of biological sequences can be interpreted as an approximate string matching problem. We propose a solution for peers and multiple sequences alignment.

All the results obtained showed that our algorithms, give the best performance on sets of practical data (benchmark from the real world). All our methods are proposed as libraries, and they are published online.

Table des matières

Table des figures

Chapitre 1

Introduction générale

ألفت من حساب الجبر والمقابلة كتابا مختصرًا، جعلته حاصرًا للَطيف الحساب وجليله لما يلزم الناس من الحاجة إليه في مواريثهم ووصاياهم وفي مقاسماتهم وأحكامهم وتجاراتهم، وفي جميع ما يتعاملون به بينهم من مساحات الأرضين وكرى الأنهار والهندسة وغير ذلك من وجوهه وفنونه

*J'ai composé dans le calcul de l'algèbre et d'al-muqabala un livre concis ; **j'ai voulu qu'il enferme ce qui est subtil dans le calcul et ce qui en lui est le plus noble, ce dont les gens ont nécessairement besoin** dans leur héritages, leurs legs, leurs partages, leurs arbitrages, leurs commerces, et dans tout ce qu'ils traitent les uns avec les autres lorsqu'il s'agit de l'arpentage des terres, de la percée des canaux, de la mensuration, et d'autres choses relevant du calcul et de ses sortes.*

Al-Khwarizmi, Le commencement de l'algèbre, éd. Roshdi Rashed

1.1 Le contexte

Le Dictionnaire est une des structures de données les plus fondamentales et les plus utilisées dans les problèmes informatiques.

Cette structure organise les données comme un ensemble sur lequel il s'agit d'effectuer une *recherche efficace*. Soit D cet ensemble et n sa cardinalité (ou taille) : *la recherche* dans D consiste à savoir si un élément x donné appartient ou non à D. *L'efficacité* mesure les ressources (temps et espace) nécessaires pour obtenir la réponse.

Le contexte d'une recherche où les données dans D ainsi que x sont considérés comme ne contenant aucune erreur est celui d'une *recherche exacte*. Lorsqu'on admet l'existence d'erreurs dans les données (D et/ou x) c'est le contexte d'une *recherche approchée*.

Les performances de la recherche dépendent, en général, du contexte (recherche exacte ou approchée), des tailles de D et x et de l'organisation de D.

Lorsque D est de grande taille et la recherche est récurrente, une "*bonne organisation*" de D devient cruciale. L'idée serait d'organiser D de manière à ce que la recherche de x dépende le moins possible de la taille de D.

Dans sa forme de base, ce problème considère des données textuelles et s'apparente au très ancien domaine de l'algorithmique du texte : D est un ensemble de mots et x un mot. Les premiers travaux ont porté sur la recherche exacte pour laquelle de nombreuses solutions satisfaisantes ont été obtenues.

Avec l'explosion de la quantité d'informations textuelles due la croissance des données de l'Internet (bases de données, articles, livres, les titres des images, vidéos, musicales), la recherche exacte est devenue, dans la plupart des cas, moins pertinente : le mot requête et/ou les mots du dictionnaire ou du texte peuvent contenir des erreurs typographiques. L'erreur peut-être due à plusieurs raisons (selon le domaine) : par exemple, une erreur de frappe lors d'une saisie rapide ou due à la méconnaissance par l'utilisateur de l'orthographe correcte (nom de personne, nom d'un produit ... etc.) ou due à la multiplicité de formes d'écriture de ce mot. Les textes qui proviennent d'un outil de reconnaissance de caractère sont susceptibles de contenir des erreurs de mauvaise reconnaissance. Une altération des données lors de leurs transmissions est donc possible.

La recherche approchée résout le problème en permettant l'existence d'erreurs entre le motif et ses occurrences.

La recherche approchée dans un dictionnaire ou dans un texte est définie comme suit :

> Étant donné un dictionnaire/texte $D = \{w_1, w_2, ..., w_d\}$ de d mots et n caractères, défini sur un alphabet spécifique Σ, étant donné un mot requête q de longueur m et un entier k représentant un seuil maximal d'erreurs, étant donnée une fonction dist qui mesure la distance entre deux mots, alors la recherche approchée de q dans D consiste à trouver toutes les occurrences dans D qui diffèrent d'au plus k erreurs de q.

La fonction la plus couramment utilisée pour déterminer la distance entre deux mots est appelée la "*distance d'édition*" [2].

Il existe de nombreuses solutions à ce problème dans la littérature [3, 4].

La recherche approchée de motifs intervient dans plusieurs domaines, principalement dans les moteurs de recherche [5], la correction d'orthographe dans les éditeurs de texte [6], dans les systèmes OCR [7], dans le craquage de mots de passe [8], dans le nettoyage des données en ligne, dans les bases de données [9]...etc.

L'une de ces applications importantes est dans le domaine de la bio-informatique où elle permet de rechercher des séquences biologiques, de les assembler, ou encore de les aligner [10, 11, 12].

D'une manière générale, la recherche approchée de motifs est utilisée dans tous les domaines qui traitent un ensemble de données constitué et défini sur un alphabet Σ donné.

1.2 La motivation de notre travail

Dans son article survey sur le problème de la recherche approchée [3], Gonzalo Navarro constate le nombre important d'*améliorations* apportées à plusieurs méthodes de résolution de ce problème sur le plan de leur complexité théorique sans considérer leur aspect pratique. Ce qui a aboutit à une dichotomie indésirable, à savoir l'existence d'une part, d'un ensemble d'algorithmes performants en théorie et très lents en pratique et d'autre part, des algorithmes rapides en pratique mais dont la complexité théorique n'est pas satisfaisante.

Le fait que les algorithmes dont la complexité théorique performante ne coïncident pas avec les algorithmes efficaces en pratique pose un important problème, en particulier à la communauté des développeurs de logiciels ou des chercheurs en bio-informatique, mais aussi à tous les utilisateurs non spécialistes d'algorithmique de texte, lorsqu'il s'agit de choisir et d'utiliser un bon algorithme de recherche de motifs. Ce choix a toutes les chances de se porter sur l'algorithme le plus simple à implémenter qui est en général l'algorithme naïf qui a les performances les plus mauvaises, ce qui aura une incidence sur les performances de l'outil qui utilise un tel algorithme [13].

Le travail dans cette thèse est fondamentalement motivé par quelques-unes des questions posées à la fin de ce constat, à savoir :

1. est-il possible de trouver un algorithme en $O(kn)$ pour la complexité du cas pire et efficace en pratique ?
2. la complexité minimale du cas moyen est connue comme étant en $O(n + (k + \log \sigma m)/m)$ et il existe un algorithme qui le réalise. Est-il possible de trouver une solution efficace en pratique ayant cette performance théorique ?
3. est-il possible, d'améliorer en pratique, la plus performante actuellement des méthodes de filtrage [3].

Un débat récent [14], entre théoriciens de la complexité et chercheurs impliqués dans le développement de logiciels, pose la question de l'écart qui existe, entre la solution théorique exponentielle du pire cas du problème SAT et ses nombreuses solutions pratiques efficaces.

Le problème est similaire pour les algorithmes non exponentiels. En effet, bien que la complexité des algorithmes de texte soit l'un des problèmes les plus étudiés en informatique théorique, elle reste insuffisante pour juger si un algorithme est efficace en pratique ou non, s'il donne de bons résultats ou non, s'il est optimal ou non en pratique en temps d'exécution et en occupation mémoire.

Sans pouvoir encore répondre à la question générale de savoir que manque t-il à la théorie de la complexité pour capturer l'efficacité des solutions pratiques que pose ce débat, cette thèse s'intéresse à la recherche de solutions efficaces en pratique, sans détériorer les performances des complexités théoriques du problème de la recherche approchée de motifs dans différents contextes de son application.

Les paramètres quantitatifs importants pour juger la qualité des solutions de ce problème sont : l'utilisation de l'espace mémoire de la solution, le temps nécessaire pour construire la structure de données du dictionnaire, et le temps nécessaire pour répondre à une requête.

En plus de ces trois paramètres, d'autres considérations importantes doivent être prises en compte, comme le fait que la structure de données de la solution soit dynamique ou non (elle accepte ou non l'addition rapide d'autres chaînes de caractères à l'index du dictionnaire), et que la solution soit *simple* ou non (utilise ou non des structures de données complexes).

Cette thèse étudie le problème de la recherche approchée dans les dictionnaires, les textes et les index et propose des algorithmes qui ne se limitent pas à l'efficacité selon une complexité théorique mais qui résolvent aussi, en pratique de façon efficace, ce problème.

Dans notre travail, nous explorons la recherche approchée dans trois directions. Nous nous intéressons à la résolution du problème théorique d'une façon générale et à ses applications dans les moteurs de recherche, dans l'alignement des séquences biologiques, et dans l'auto-complétion dans le web du côté serveur et du côté client.

Le but de ce travail est de fournir des algorithmes validés aussi en pratique et très performants et très compétitifs vis-à-vis de toute autre solution qui existe aujourd'hui et de fournir comme résultat des bibliothèques utilisables par les autres développeurs/chercheurs dans leurs travaux.

1.3 Notre contribution

Dans cette thèse, nous apportons de nouvelles solutions au problème de la recherche approchée, toutes fondées sur l'idée d'associer une implémentation pratique au moins aussi

efficace que la solution théorique optimale : la première solution contribue à la résolution du problème pour un seuil d'erreurs $k \geq 2$. La seconde, explore de nouveau la possibilité d'utiliser de manière efficace l'utilisation de la structure de Trie associée à une recherche bidirectionnelle et d'observer en pratique pour quel seuil d'erreurs cette approche est satisfaisante. Les deux autres contributions de cette thèse sont deux applications dans des domaines spécifiques et importants de la recherche approchée : la recherche dans le Web et la recherche dans les structures du génome.

La recherche approchée pour $k \geq 2$ en utilisant les tableaux de hachage. Nous proposons un algorithme qui utilise une structure de donnée basée sur le hachage avec sondage linéaire et des signatures de hachage associées pour optimiser le temps de la recherche. Pour le dictionnaire exact, nous utilisons une structure de donnée basée sur plusieurs tableaux de hachage organisée par longueur des mots et exploitons l'idée d'utiliser un dictionnaire des listes de substitution [15].

Afin d'optimiser l'espace mémoire, nous compressons ces deux structures de données en utilisant une autre structure de données (le vecteur des bits). La version non compressée est *progressive* : nous pouvons ajouter de nouveaux mots jusqu'à un certain facteur.

La structure de données de cette solution occupe un espace $O(n)$, où n est la taille totale du dictionnaire et en $(O(n \log \sigma))$ bits. Si le compactage de la table de hachage est utilisé, le dictionnaire final occupe au plus un espace mémoire de $n(2 \log \sigma + O(1))$ bits. La recherche d'un mot q de m lettres et qui a un nombre d'occurrences $occrs$ s'effectue en un temps proche de $(m + occrs)$ et plus exactement en $(O(m \lceil \frac{m \log \sigma}{w} \rceil)$ où w représente la taille d'un mot mémoire (qui vaut 32 ou 64 bits).

Les expérimentations montrent que cette solution est en concurrence avec les solutions précédemment proposées et ce travail est publié dans [16], et son implémentation est disponible sur :
https://code.google.com/p/compact-approximate-string-dictionary/ [1]

Une recherche approchée en utilisant le Trie et le Trie inversé. La structure de Trie est particulièrement intéressante pour représenter un dictionnaire dans le contexte d'une recherche exacte. Il permet de rechercher les mots par préfixe commun. Ainsi le Trie permet de savoir si un mot q de m lettres appartient ou non au dictionnaire en temps proportionnel à la longueur du mot recherché, c'est-à-dire en $O(m)$.

Pour la recherche approchée, l'existence de k erreurs possibles doit être prise en compte tout en préservant une complexité indépendante de la taille du dictionnaire.

Dans ce travail, nous utilisons une structure de données bidirectionnelle (Trie, et Trie inversé), pour proposer une méthode qui résout la recherche approchée pour $k = 1$. Si

1. Visité le : 02-07-2016.

on admet l'existence d'une seule erreur, le mot q peut s'écrire comme étant égal à la concaténation d'un préfixe exact P_1, d'une lettre erronée c et d'un suffixe exact P_2, c'est-à-dire que $q = P_1 c P_2$. L'idée consiste à rechercher le préfixe P_1 dans le Trie et le suffixe P_2 dans le Trie inversé en parallèle. À la rencontre de l'erreur c il faut s'assurer que P_1 et P_2 appartiennent au même mot en réduisant au maximum le nombre de vérifications.

Notre méthode réduit, lors de la recherche dans le Trie, le nombre de branches sortantes testées dans chaque nœud. Elle ne choisit que les chemins qui peuvent mener à des solutions. La complexité du pire cas de cette recherche s'effectue en $O(m^2 \times \sigma)$. La complexité moyenne est en $O(m^2)$ si $m \geq 2\frac{logd}{log\sigma}$, et elle est en $O(m + m.occrs)$ si $m \geq 2(\frac{logd}{log\sigma} + \frac{logm}{log\sigma})$ où $occrs$ représente le nombre d'occurrences de la solution.

Les résultats expérimentaux montrent que cette méthode surpasse, en temps d'exécution, toutes les autres implémentations testées à ce jour. La performance de cette solution est en proportion constante par rapport à la recherche exacte, indépendante de la taille du dictionnaire.

Le résultat de ce travail est disponible sur : `https://github.com/chegrane/TrieRTrie` [1].

L'auto-complétion approchée. L'auto-complétion de requêtes sur le Web est un mode de travail qui permet d'assister un utilisateur lors de la saisie de sa requête. Elle consiste à lui proposer une liste de mots, les plus pertinents, pour compléter sa requête. Elle facilite et accélère la saisie en offrant une liste de suggestions pour compléter les caractères introduits/tapés par l'utilisateur dans le champ de texte.

L'auto-complétion approchée est un problème de recherche approchée où il s'agit de trouver tous les suffixes d'un préfixe contenant des erreurs.

Dans ce travail, nous présentons une méthode basée sur la structure de données Trie pour effectuer une auto-complétion approchée efficace admettant une seule erreur de type distance d'édition, dans une architecture client/serveur.

Nous proposons également une méthode qui réduit le nombre de transitions sortantes testées dans chaque nœud - en particulier dans le premier niveau - du Trie. La complexité temporelle moyenne de cette recherche approchée est $O(m^2)$.

Ce travail est publié dans [17], et son implémentation, une bibliothèque (nommée `appacolib`), est disponible sur : `https://github.com/AppacoLib/api.appacoLib` [2]

Alignement multiple des séquences d'ADN : L'une des applications importantes des algorithmes de texte et de la recherche approchée est l'alignement des séquences biologiques.

1. Visité le : 02-07-2016.
2. Visité le : 02-07-2016.

En bio-informatique, l'alignement des séquences biologiques permet de rechercher dans les séquences (ADN, ARN ou protéines) des régions similaires.

Trouver le meilleur alignement entre la paire de séquences x et y revient à trouver le plus petit nombre d'erreurs k entre x et y, autrement dit la meilleure façon de mettre x face à y avec un nombre minimum de différences.

Dans ce travail, nous présentons un nouvel algorithme d'alignement par paire de séquences biologiques et un algorithme d'alignement multiple basé sur l'algorithme connu DIALIGN.

Les résultats des différents tests montrent que notre approche est très efficace dans les deux cas et donne un très bon temps d'exécution comparé à (DIALIGN 2.2).

Ces résultats sont publiés dans deux articles [18, 19], et les codes sources et les logiciels sont disponibles sur : `https://github.com/chegrane/diaway` [1] pour la première version, et sur `https://github.com/chegrane/DiaWay_2.0` [2] pour la deuxième version.

Toutes nos implémentations sont distribuées sous la licence publique générale limitée GNU (GNU LGPL).

Pour évaluer nos méthodes en pratique, nous les avons comparées avec les algorithmes existants qui donnent les meilleurs résultats sur des ensembles de données pratiques. Les expérimentations sont réalisées sur des dictionnaires des langues (Anglais, WikiTitle, ...). Ceci pourrait éventuellement expliquer certaines améliorations pratiques obtenues (par rapport à la complexité théorique du pire cas) étant donné que pour les langues les combinaisons du pire cas n'existent pas ou sont très rares.

1.4 L'organisation de la thèse

Le reste de cette thèse est organisé comme suit :

Dans le chapitre qui suit cette introduction, nous donnons quelques préliminaires et définitions nécessaires à la compréhension du domaine et des chapitres qui suivent.

Ensuite, nous donnons un état de l'art orienté où nous expliquons les concepts généraux des méthodes de la recherche approchée et nous détaillons quelques méthodes qui sont liées à notre travail.

Dans les quatre chapitres suivants nous présentons respectivement nos quatre travaux : la recherche approchée en utilisant les tableaux de hachage, puis la recherche approchée en utilisant le Trie et le Trie inversé, puis l'auto-complétion approchée et enfin l'alignement multiples des séquences biologiques.

Le dernier chapitre conclut cette thèse et donne quelques perspectives de recherche.

1. Visité le : 02-07-2016.
2. Visité le : 02-07-2016.

Chapitre 2

Préliminaires

Ce chapitre introduit les définitions de base de l'algorithmique de texte et de la théorie des langages. Il est basé sur les références fondamentales, parmi lesquelles nous citons : [20, 21, 22, 11]. Il rappelle aussi quelques structures de données fondamentales utilisées comme dictionnaire ou structure auxiliaire pour représenter un dictionnaire. Nous donnons les algorithmes de ces structures dans le chapitre suivant.

2.1 Alphabet, mot

Alphabet Un *alphabet* Σ est un ensemble fini non vide d'éléments appelés symboles ou lettres. La cardinalité de cet ensemble est notée $|\Sigma| = \sigma$.
Exemples : $\Sigma_1 = \{a, b, ..., z\}$, $\Sigma_2 = \{0, 1\}$, $\Sigma_3 = \{A, C, T, G\}$.

Mot Un *mot* v défini sur un alphabet Σ est une concaténation d'éléments de Σ, $v = e_1 e_2 ... e_m$. La *longueur* d'un mot est le nombre de lettres qui le composent, et on la note $|v| = m$. Exemple : $\Sigma_2 = \{0, 1\}$, $v = 01101001$ est défini sur Σ_2 et $|v| = 8$.
Le *mot vide* (sa longueur vaut zéro) est noté par ϵ.
L'ensemble de tous les mots définis sur l'alphabet Σ est noté par Σ^*. $\Sigma^+ = \Sigma^* - \epsilon$.
La concaténation de deux mots $u = aax$, et $v = bb$ défini sur l'alphabet $\Sigma = \{a, b, x\}$ est le mot noté uv obtenu en mettant bout à bout u et v, $uv = aaxbb$.

2.2 Préfixe, suffixe, facteur, miroir

Préfixe Un mot u est un préfixe d'un mot w s'il existe un mot v tel que $w = uv$. Soit w un mot dans Σ^*, $w = a_1 \cdots a_n$, alors tout mot $u \in \{\epsilon, a_1, a_1 a_2, ..., a_1 \cdots a_n\}$ est préfixe de w. On note $Pref(w)$ l'ensemble $\{\epsilon, a_1, a_1 a_2, ..., a_1 \cdots a_n\}$ de tous les préfixes de w. L'ensemble $Pref(w) - \{w\}$ est dit l'ensemble des *préfixes propres* de w.

Suffixe Un mot u est un suffixe du mot w s'il existe un mot v tel que $w = vu$. Soit $w = a_1 \cdots a_n$ un mot dans Σ^*, alors tout mot $u \in \{\epsilon, a_n, a_{n-1} a_n, ..., a_1 \cdots a_n\}$

est suffixe de w. On note *Suff(w)* l'ensemble de tous les suffixes de w. L'ensemble *Suff(w)* $- \{w\}$ est dit l'ensemble des *suffixes propres* de w.

Facteur Soient w, u, v, v' des mots dans Σ^* tels que $w = vuv'$. Alors u est dit facteur de w. Remarquons que si $v = \epsilon$ alors u est préfixe de w et si $v' = \epsilon$ alors u est suffixe de w.

L'ensemble des facteurs de T que l'on note $Fact(T)$ est défini par l'expression suivante :

$$Fact(T) = Pref(Suff(T))$$

Mot miroir Le mot miroir du mot $w = a_1 \cdots a_n$ est le mot $\bar{w} = a_n \cdots a_1$.

2.3 Fonctions de distance

Soit E un ensemble donné. Une métrique *dist* sur E est une application :

$$\begin{aligned} dist : E \times E \Longrightarrow \mathbb{R}^+ \quad & \text{telle que } \forall x, y, z \\ dist(x, y) &= dist(y, x) \\ dist(x, y) &= 0 \Longleftrightarrow x = y \\ dist(x, y) &\leq dist(x, z) + dist(z, y) \end{aligned}$$

2.3.1 Distance entre mots

Lorsque x et y sont des chaînes de caractères, alors $dist(x, y)$ mesure le degré de ressemblance entre x et y et est définie comme étant le nombre minimum d'opérations telles que l'insertion d'un caractère, la suppression d'un caractère, la substitution d'un caractère par un autre, etc., nécessaires pour transformer le mot x en le mot y.

Dans ce calcul, le coût de chaque opération vaut 1 et la distance entre mots est donc un nombre entier.

2.3.2 Quelques fonctions de distance entre mots

Il existe plusieurs fonctions de distance qui se distinguent par les opérations que l'on peut appliquer sur les mots, les plus connues en recherche de motifs sont :

La distance de Levenshtein [2] (ou distance d'édition) : elle autorise trois opérations : la substitution ($a \longrightarrow b$), la suppression $a \longrightarrow \epsilon$, et l'insertion ($\epsilon \longrightarrow a$), où $a, b \in \Sigma$, $a \neq b$, et ϵ : le mot vide. La distance de Levenshtein est la métrique la plus utilisée pour déterminer la différence entre deux mots.

Exemple : Soit $V = ABCjEF$ et $W = xBCEFy$, pour transformer V en W, on substitue le 1^{er} caractère A par x, et on supprime le $4^{\text{ème}}$ caractère j, et on insère le y à la fin.

La distance de Hamming [23] Cette fonction autorise une seule opération : la substitution.

La distance de Damerau [24] En plus des opérations de la distance de Levenshtein, elle autorise l'opération de transposition ($ab \longrightarrow ba$). Exemple : $V = ABCxyEF$ et $W = ABCyxEF$, on a une seule opération, la transposition de ($xy \longrightarrow yx$).

2.4 Les structures de données pour dictionnaires et textes

Un dictionnaire est un ensemble $D = \{w_1, w_2, ..., w_d\}$ de d mots, tels que $w_i \neq w_j$ ($i \neq j$). La taille du dictionnaire est $|D| = \sum_{i=1}^{i=d} |w_i| = n$.

Un texte T est une suite de n caractères. Il peut être vu aussi comme étant un ensemble de d mots $T = \{w_1, w_2, ..., w_d\}$, où chaque mot peut avoir plusieurs occurrences.

Pour une recherche efficace dans le texte et/ou dans le dictionnaire, il est important d'organiser et de stocker les données en une structure qui en facilite l'exploitation (l'accès et les modifications).

Généralement une structure de données occupe plus d'espace mémoire que les données sur lesquelles elle est construite. Une *structure de données succincte* occupe un espace proche de la taille des données qu'elle doit représenter. Elle occupe un espace plus petit si elle exploite des techniques de compression.

Dans la bibliographie classique de l'algorithmique de texte [11, 21, 25, 26], il existe plusieurs structures de données pour index et dictionnaires. Nous nous limitons dans ce qui suit au rappel de quelques structures de données utilisées dans la représentation d'un dictionnaire/index et exploitées dans cette thèse.

2.4.1 Arbre de suffixes

Soit T un mot défini sur un alphabet Σ tel que $|T| = n$.

L'arbre des suffixes [11, 27] de T est un arbre enraciné à n feuilles terminateur tel que :

1. Chaque feuille i représente un seul et unique suffixe, c'est le mot $T[i..n]$;
2. Ses branches sont étiquetées par des mots non vides.
3. Ses nœuds internes sont de degrés (nombre de branches sortantes) > 1 .
4. Les étiquettes des branches sortantes d'un nœuds ne peuvent pas commencer par le même caractère.
5. La concaténation des étiquettes sur le chemin qui commence à la racine R et qui se termine à une feuille i forme le suffixe $T[i..n]$.

Il existe différents algorithmes permettant la construction de l'arbre de suffixes dans un temps et espace linéaire [28, 29, 30].

2.4.2 Trie

Soit D un dictionnaire de d mots. Un Trie [31, 32, 33]) pour D est un arbre enraciné de préfixes communs et qui a d feuilles. Chaque nœud représente un préfixe commun des mots de D et a un ou plusieurs nœuds fils. Chaque arête est étiquetée par un caractère des mots de D. Deux arêtes qui sortent du même nœud ne peuvent pas être étiquetées par le même caractère. Chaque chemin de la racine à une feuille est un mot du dictionnaire.

La construction d'un Trie est en $O(n)$ où n est la taille du dictionnaire.

La recherche d'un mot p de longueur $|p| = m$ consiste à parcourir le Trie de la racine en suivant les lettres de p sur les arêtes et est en $O(m)$.

2.4.2.1 Trie inversé

On note par $\bar{w}$ le miroir de w et par $\bar{D} = \{\bar{w_1}, \bar{w_2}, ..., \bar{w_d}\}$ le dictionnaire des mots miroirs de D. Le Trie inversé de D est le Trie de $\bar{D}$. Il permet de faire la recherche de droite à gauche.

2.4.2.2 Un Trie compact

Un Trie compact [11, 31] est une structure de données qui optimise l'espace mémoire par le compactage des chemins du Trie.

Soient les nœuds $\{nd_i, nd_{i+1}, ..., nd_{i+j}\}$ étiquetés respectivement par les caractères suivant $\{c_i, c_{i+1}, ..., c_{i+j}\}$, tous sur le même chemin, et chaque nd_{k+1} est un fils unique de nd_k , avec $k \in [i..i + j - 1]$. Le compactage de ce chemin consiste à fusionner ces nœuds en un seul nœud nd, et en concaténant tous leurs caractères en une seule sous-chaine $\{c_i c_{i+1} ... c_{i+j}\}$ sur la transition sortante du nœud parent.

Remarquons enfin que l'arbre de suffixes utilisé dans l'indexation d'un texte T est un Trie compact construit sur tous les suffixes de T. Chaque chemin de la racine à une feuille représente un suffixe du texte T.

2.4.3 Le tableau LCP

Le tableau des plus long préfixes communs d'un dictionnaire D ([34, 35]) est une structure de données qui permet de stocker les longueurs des plus longs préfixes communs entre les paires de mots consécutifs de D dans l'ordre lexicographique.

Soient $D = \{w_1, w_2, ..., w_n\}$ le dictionnaire et $LCP(x, y)$ la fonction qui calcule la longueur du plus grand préfixe commun entre deux mots x et y. Le tableau Tab_LCP contient l'ensemble des entiers $LCP(w_i, w_{i+1})$, pour $i = 1, \ldots n-1$.

$$Tab\_LCP = \{LCP(w_1, w_2), LCP(w_2, w_3), ..., LCP(w_{n-1}, w_n)\}$$

2.4.4 Le Tas

La structure de données Tas ([36, 21]) est un arbre binaire presque complet : l'arbre est complètement rempli à tous les niveaux, sauf (dans certains cas) le niveau le plus bas, rempli de la gauche jusqu'à un certain point. Il y a deux types de Tas, Tas Max et Tas Min. Dans le Tas Max, la clé dans chaque nœud a une valeur supérieure à celle de ses nœuds fils, et l'inverse dans le Tas Min.

Le Tas peut être implémenté par un tableau d'une taille de $2n+1$ où n est le nombre de nœuds. Les deux fils d'un nœud i sont dans les positions $(2i, 2i+1)$.

2.4.5 La file de priorité

La file de priorité [37, 38] est une structure de données qui permet l'accès en premier à l'élément ayant la plus grande priorité. Il existe deux types de file de priorité, celle qui donne la plus grande clé en premier (la plus grande priorité = la plus grande clé), et l'inverse, celle qui donne la plus petite clé en premier (la plus grande priorité = la plus petite clé).

La file de priorité permet trois opérations : 1) ajouter un élément, 2) accéder à l'élément qui a la plus grande priorité, 3) supprimer l'élément qui a la plus grande priorité.

La file de priorité peut être implémentée avec une variété de structure de données comme par exemple un tableau ou une liste chainée, mais généralement, elle est implémentée avec la structure du Tas.

2.4.6 Prefix-sum

La structure de données prefix-sum [39, 40, 41] permet de stocker des éléments numériques (généralement dans un tableau) tels que la valeur stockée dans chaque position est la somme de toutes les valeurs des éléments précédents. $val(i) = \sum_{j=0}^{j=i} val(j)$.

La méthode consiste à transformer un ensemble de valeurs $S = \{v_1, v_2, ..., v_n\}$ en un autre ensemble $S' = \{v'_1, v'_2, ..., v'_n\}$, tel que $v'_i = \sum_{j=0}^{j=i} v_i = v'_{i-1} + v_i$.

Exemple :

S :	{1,1,1,2,5,5,5}
prefix-sum (S) :	{1,2,3,5,10,15,20}

2.4.7 Le tableau de bits

La structure de données tableau de bits (bit-vector) est un tableau utilisé simplement pour stocker et retrouver les bits (1 ou 0) dans une position donnée. Elle permet de savoir si l'élément dans une position existe ou non.

Généralement, chaque case du tableau a une taille d'un mot mémoire w. Dans les machines actuelles $w = 32$ ou 64 selon le système utilisé de 32 ou 64 bits respectivement.

Dans une structure de dictionnaire, le tableau de bits est augmenté avec des informations supplémentaires afin de supporter les opérations de rang et de sélection (rank/select) [42, 43, 44]. L'opération de rang (rank(1/0,i)) calcule le nombre de 1 (ou 0) depuis le début du tableau jusqu'à une position donnée i, et de même, l'opération de la sélection (select(1/0,i)) permet de retourner la position de l'occurrence numéro i de l'élément 1 (ou 0).

2.5 Les tables de hachage

Une table de hachage est une structure de données qui généralise la notion simple d'un tableau ordinaire pour implémenter les dictionnaires. On utilise une fonction de hachage h pour appliquer l'ensemble des clés $U = \{cl_1, cl_2, ..., cl_n\}$ vers les positions $\{1, 2, .., n\}$ des cases de la table de hachage Tab de n cases

Dans le cas idéal, lorsque la fonction de hachage h est bijective, elle associe à chaque clé une position unique et toute position est associée à une seule clé. Ce hachage est dit *« parfait »*. Lorsque h n'est pas injective elle retourne la même valeur de position pour des clés différentes. On dit qu'il y a *« collision »*.

Il existe plusieurs solutions pour résoudre le problème de collision comme le hachage avec chaînage, l'adressage ouvert, le sondage linéaire, le double hachage, etc. [21, 22].

Les deux sous-sections suivantes présentent respectivement le hachage parfait minimal et le hachage avec sondage linéaire.

2.5.1 Le hachage parfait minimal

Une fonction de hachage parfait est une fonction bijective qui ne donne donc pas de collision. Étant donné un ensemble $U = \{w_1, w_2, \ldots, w_n\}$ de n éléments, la fonction associe chaque élément de U à un numéro distinct dans l'intervalle [0..m-1] :

$$\begin{array}{rccl} h: & U & \longrightarrow & \{1, 2, ..., m-1\} \text{ avec } m \geq n \\ & w_i & \longmapsto & p \end{array}$$

$p = h(w_i)$ est la position de w_i dans D.

Une fonction de hachage est dite minimale $mphf$ [1] ([45, 46, 47]) si $m = n$. Une fonction $mphf$ associe n éléments à n valeurs entières successives. Le but est de réduire l'espace mémoire de stockage utilisé par la fonction de hachage parfait.

Habituellement, la construction de la $mphf$ implique des étapes qui consistent à utiliser une fonction de hachage qui applique les mots à des nombres distincts de $O(w)$ bits, puis de ranger ces nombres dans un tableau de n cases en utilisant une autre fonction de hachage.

2.5.2 Table de hachage avec sondage linéaire

Le hachage avec sondage linéaire [48] est une méthode (parmi d'autres) pour résoudre le problème de collision. La méthode consiste à placer la donnée cl dans la première case vide dans la table de hachage si la position $h(cl)$ est non vide.

La taille T du tableau est plus grande que le nombre d'éléments n. Le taux de remplissage du tableau ou le facteur de chargement (Load Factor) doit être inférieur à 1 ($\alpha < 1$). Si un tableau de hachage stocke n éléments, alors sa capacité est $T = \lceil n/\alpha \rceil$.

Pour insérer un nouvel élément, si la position $h(cl)$ est vide, alors il n'y a pas de collision et l'élément est inséré à la position $h(cl)$. Dans le cas contraire, on cherche la première case vide après $h(cl)$ pour insérer l'élément, on vérifie dans les positions $h(cl)+1, h(cl)+2 \ldots$, jusqu'à trouver une case vide (si l'une des positions est égale à $T-1$, alors la prochaine position sera la position 0).

La recherche d'un élément se fait par la comparaison avec le contenu de la table de hachage à la position $h(cl)$. Si l'élément n'est pas à cette position, alors on parcourt toutes les cases consécutives à partir de cette position jusqu'à ce qu'on trouve l'élément recherché, ou on atteint une case vide ce qui implique que l'élément recherché n'existe pas dans la table de hachage.

Pour supprimer un élément, on doit faire un décalage (vers la gauche) aux éléments qui se trouvent directement après la case supprimée, et qui ont la même valeur de hachage de la case supprimée. Cette opération est nécessaire afin de ne pas laisser une case vide et donc perdre l'accès aux éléments qui ont la valeur de hachage que la case supprimée.

2.6 Notion de complexité algorithmique

La complexité algorithmique [49] est une fonction qui permet d'évaluer la quantité de ressources (temps et espace mémoire) nécessaire utilisée pour le fonctionnement d'un algorithme donné. Pour calculer la complexité du cas pire, nous utilisons une évaluation asymptotique, où le nombre des éléments n manipulés est assez grand (tend vers l'infini), et nous ignorons les constantes.

1. Minimal perfect hashing.

2.6.1 Les notations de la complexité

Il existe plusieurs notations pour designer la complexité d'un algorithme donné. Parmi les notations les plus utilisées, nous avons les notations : $O(n)$, $\Omega(n)$, $\Theta(n)$ [50]. Soient f et g deux fonctions mathématiques et n le nombre d'éléments manipulés, supposé suffisamment grand (tend vers l'infini).

Grand O $f(n) = O(g(n))$, la fonction f est bornée asymptotiquement par la fonction g. $|f(n)| \leq k \times |(g(n))|$ avec k une constante strictement positive. Cela est équivalent à dire que $\frac{f(n)}{g(n)} \leq k$. La notation O décrit une borne supérieure asymptotique.

Grand Omega $f(n) = \Omega(g(n))$, la fonction f est minorée asymptotiquement par la fonction g (à un facteur près), $|f(n)| \geq k \times |(g(n))|$, pour un $k > 0$. La notation Ω décrit une borne inférieure asymptotique.

Grand Theta $f(n) = \Theta(g(n))$, la fonction f est dominée et soumise asymptotiquement par la fonction g, $k_1 \times |(g(n)) \leq |f(n)| \leq k_2 \times |(g(n))|$, pour un $k_1 > 0$ et un $k_2 > 0$.

La notation Θ décrit une borne supérieure et inférieure en même temps, $\Theta(g(n)) = \Omega(g(n))$ et $O(g(n))$.

2.6.2 Les modèles de calcul de la complexité

L'évaluation des ressources nécessaires à l'exécution d'un algorithme se fait dans un modèle abstrait de machines.

Il existe plusieurs modèles de calcul dans la littérature, comme la machine de Turing, la machine RAM (Random Access Machine), la machine avec mémoire externe EMM [51], le modèle Cache-Oblivious (machine à plusieurs niveaux de mémoire) [52], la machine PRAM (Parallel Random Access Machine) [53], etc.

Dans ce qui suit, nous décrivons brièvement les deux modèles les plus utilisés qui sont la machine de Turing et la machine RAM.

La machine de Turing Une machine de Turing [54] est une machine abstraite composée d'un ruban infini divisé en cases consécutives sur lesquelles la machine écrit/efface des symboles (un alphabet fini) en déplaçant une tête de lecture/écriture vers l'avant ou vers l'arrière.

La machine de Turing est caractérisée par la position de la tête dans le ruban, le contenu de la case sur laquelle pointe la tête de lecture/écriture, et une table des actions à faire (les règles de transition entre les cases) qui sert comme un programme à la machine. Le ruban est initialement vide, toutes les cases contiennent le caractère $\sqcup \notin \Sigma$.

Une machine de Turing est formellement définie par : $(Q, \Sigma, \Gamma, \sqcup, q_0, q_f, \delta)$

- Q un ensemble fini des états.
- Σ un ensemble fini d'alphabet (les entrées-sorties) ; le choix typique est $\Sigma = \{0, 1\}$.

- Γ un ensemble fini d'alphabet du ruban, $\Sigma \subseteq \Gamma$.
- $\sqcup \in \Gamma$ un symbole (caractère) blanc, avec $\sqcup \notin \Sigma$.
- $q_0 \in Q$ un état initial.
- $q_f \in Q$ un état final.
- $\delta : Q \setminus \{q_f\} \times \Gamma \to Q \times \Gamma \times \{L, R, N\}$, une fonction de transition partiellement définie. $\{L, R, N\}$ signifie que la tête de lecture peut se déplacer vers la gauche ou vers la droite par une seule case, ou ne se déplace pas.
  Si δ n'est pas défini sur l'état actuel et le symbole actuel du ruban alors la machine s'arrête.

La machine RAM Le modèle RAM (Random Access Machine) [55, 56] est un modèle d'une machine abstraite assez proche de l'architecture d'un ordinateur. La machine abstrait RAM est composée principalement d'une unité du calcul, des registres, d'une mémoire divisée en deux parties, l'une, pour stocker les données, et l'autre pour stocker les instructions (le programme).
La machine RAM utilise l'adressage indirect pour accéder aux différentes cases de la mémoire. La taille d'une cellule mémoire est notée par w, toutes les cellules ont le même coût d'accès ; le coût pour accéder à une case mémoire est unitaire.
Dans ce modèle, chaque opération arithmétique ou logique à un coût unitaire, sauf les deux opérations la multiplication et la division. Certains chercheurs les considèrent comme deux opérations à coût unitaire, et d'autres non.

2.6.3 La notion de la complexité moyenne

La complexité en moyenne [57, 58] calcule la quantité des ressources (typiquement, le temps) nécessaire utilisée par un algorithme qui agit sur des données en entrée qui sont équiprobables, c'est-à-dire que les données en entrée ont une distribution qui ne provoque pas le pire cas.

La complexité en moyenne est nécessaire pour calculer la performance qui est proche de la réalité, lorsque les données en entrée ne provoquent pas le pire cas (ou le pire cas se produit rarement). La complexité en moyenne permet de distinguer l'algorithme le plus efficace en pratique entre l'ensemble de tous les autres algorithmes qui peuvent avoir la même complexité dans le pire des cas.

L'analyse de la complexité moyenne dans certains domaines (comme la cryptographie) permet de bien comprendre le comportement de l'algorithme et le réadapter par exemple pour générer des instances difficiles dans le domaine de la cryptographie.

Chapitre 3

Algorithmes et structures d'index pour le problème de la recherche approchée.

L'algorithmique du texte, en tant que domaine qui porte sur l'étude des structures de données textuelles et d'algorithmes les traitants, a donné lieu à de nombreuses contributions, en particuliers les articles survey de Navarro [3, 59] et Boytsov [4] ou bien les livres : de M.Crochemore [25, 26], de D. Gusfield [11], de C.Charras et T.Lecroq [60] et de G.Navarro et M.Raffinot [13].

Ce chapitre présente quelques-uns de ces travaux, en particuliers ceux fondés sur quelques idées que nous exploitons pour proposer de nouvelles solutions présentées dans les prochaines chapitres.

Les algorithmes utilisés pour résoudre le problème de la recherche approchée peuvent être regroupés selon le type de données traitées, leurs tailles, le scénario choisi etc. Ils peuvent être divisés en deux grandes catégories : 1) les algorithmes en-ligne (online), 2) les algorithmes hors-ligne (offline). On peut aussi les classer en deux autres catégories : 1) algorithmes traitants le texte, 2) algorithmes traitants le dictionnaire.

Les algorithmes en-ligne et hors-ligne : Les méthodes en-ligne sont utilisées lorsque le texte ne peut être pré-traité et indexé. C'est le cas lorsque ce texte n'est par exemple pas connu à l'avance ou s'il est susceptible d'être modifié dans le temps de très nombreuses fois (par exemple l'édition de texte et l'OCR). La complexité temporelle de la recherche par ces méthodes est proportionnelle à la taille du texte.

Les méthodes hors-ligne sont utilisées lorsque la quantité d'informations textuelles est importante et le texte est connu à l'avance. Les méthodes hors-ligne construisent des structures de données pour stocker et pré-traiter le texte afin de répondre rapidement à une requête dans un temps proportionnel à la longueur du mot requête (linéaire dans le meilleur des cas).

Les algorithmes du texte et de dictionnaire : La recherche dans un dictionnaire est différente de la recherche dans un texte car dans un dictionnaire, les mots sont ordonnés et ils n'apparaissent qu'une seule fois contrairement à un texte où les mêmes mots apparaissent plusieurs fois dans différentes positions, en plus de l'utilisation de différents types de mots vides (stopwords) et les signes de ponctuation...etc.

Il arrive que le texte soit une seule chaîne de caractères comme par exemple les chaînes génomiques comme l'ADN.

La recherche approchée dans un dictionnaire doit retourner tous les mots du dictionnaire qui diffèrent d'au plus k erreurs du mot requête q. La recherche dans un texte consiste à trouver toutes les solutions approchées et leurs occurrences (leurs positions).

En général, toutes les méthodes peuvent être appliquées dans les deux contextes (texte/dictionnaire) avec de légères modifications ou bien par l'ajout de simples structures de données (par exemple, un index inversé). Comme exemple typique de structures de données pour la recherche dans un dictionnaire, on peut citer, le Trie. Pour la recherche dans un texte, on peut citer l'arbre des suffixes qui est un Trie (compact) contenant tous les suffixes du texte.

Dans les sections de ce chapitre, nous donnons un aperçu sur quelques concepts de la recherche approchée, ensuite nous présentons les travaux qui sont fortement liés à notre travail.

3.1 Classification des méthodes de la recherche approchée

Les différents algorithmes utilisés pour résoudre le problème de la recherche approchée dans ses deux versions (en-ligne / hors-ligne) peuvent être regroupés dans des catégories selon leurs méthodes et techniques communes. Dans ce travail, toutes les méthodes sont regroupées comme suit :

1. La programmation dynamique.
2. Indexation (seulement dans le cas hors-ligne).
3. Les méthodes de filtrage.
4. La génération de voisinage.
5. Les méthodes qui utilisent le hachage.
6. Les méthodes de bit-parallélisme.
7. Les méthodes qui utilisent le parallélisme.
8. Les méthodes hybrides.

Dans les sous-sections qui suivent, nous expliquons chaque catégorie et nous donnons comme exemples quelques algorithmes qui utilisent les méthodes et les techniques de cette catégorie.

3.1.1 La programmation dynamique

La programmation dynamique est un paradigme de programmation permettant de résoudre des problèmes complexes en les décomposant en une collection de sous-problèmes plus simples. La méthode examine les solutions des sous-problèmes déjà calculées et combine leurs résultats pour trouver la meilleure solution du problème donné [61, 62]. La programmation dynamique est utilisée dans plusieurs domaines pour résoudre des problèmes d'optimisation.

Dans l'algorithmique du texte, la programmation dynamique est utilisée pour calculer la distance entre deux mots $dist(x, y)$, lors d'une recherche approchée des motifs ou d'un alignement de deux séquences.

Le principe de base est de construire une matrice M de $|x+1| \times |y+1|$ cases où $M[i, j]$ représente le nombre minimum d'opérations nécessaires pour que la sous chaîne $x[1..i]$ ($x[1..i] = \{x_1, x_2, ..., x_i\}$) corresponde à la sous-chaîne $y[1..j]$. La matrice est remplie selon les règles suivantes :

$M[i, 0] = i$
$M[0, j] = j$
$M[i, j] = M[i-1, j-1] \quad si\ (x_i = y_j)$
$M[i, j] = 1 + min(M[i-1, j], M[i, j-1], M[i-1, j-1]) \quad sinon$

Le résultat final (la distance d'édition) se trouve à la dernière case $M[|x|, |y|]$, la distance des sous-chaînes ($x[1..i]$ et $y[1..j]$) se trouve dans la case $M[i, j]$. Nous donnons ci-dessous l'explication de ces règles.

- $M[i, 0] = i$ ($M[0, j] = j$) : la distance d'édition entre une chaîne x et la chaîne vide ϵ est $|x|$. L'opération pour transformer x en ϵ est la suppression. La case $M[i, 0]$ représente la distance d'édition entre une sous-chaîne $x[1..i]$ de longueur i et la chaîne vide, donc $dist(x[1..i], \epsilon) = i$, (de même pour la case $M[0, j]$, $dist(\epsilon, y[1..j]) = j$).
- $M[i, j] = M[i-1, j-1]$: dans le cas où les deux caractères x_i et y_j sont égaux, alors nous n'avons aucune opération. Donc la valeur de la distance d'édition revient à la valeur de 2 sous-chaînes précédentes c'est-à-dire $x[1..i-1]$ et $y[1..j-1]$. Donc si $(x_i = y_j)$ alors $dist(x[1..i], y[1..j]) = dist(x[1..i-1], y_[1..j-1])$.
- $M[i, j] = 1 + min(M[i-1, j], M[i, j-1], M[i-1, j-1])$: dans le cas où $(x_i \neq y_j)$, cela signifie que nous avons l'une des trois opérations (substitution, suppression, insertion), donc la distance d'édition de (x_i, y_j) revient à ajouter 1 à la valeur minimale des trois cases $(M[i-1, j], M[i, j-1], M[i-1, j-1])$ qui contient la valeur de

la distance d'édition des trois cas suivants : $(x[1..i-1], y[1..j])$, $(x[1..i], y[1..j-1])$, $(x[1..i-1], y[1..j-1])$.

Si nous prenons la valeur$(M[i-1, j-1])$, alors l'opération de (x_i, y_j) est la substitution. Si nous prenons $M[i-1, j]$ alors l'opération est la suppression, et pour $M[i, j-1]$ c'est l'insertion.

Les deux opérations (la suppression et l'insertion) sont interchangeables, selon le sens de transformation des deux mots. Transformer x en y où l'inverse.

Pour plus de détails le lecteur pourra consulter le survey de Navarro [3].

L'algorithme du calcul de la distance d'édition peut être adapté pour la recherche de motifs dans un texte [63, 64, 65, 66].

Pour trouver l'alignement entre une séquence x et une séquence y, il suffit juste de retrouver le chemin de la distance d'édition dans le sens inverse depuis la case $M[|x|, |y|]$ vers la case $M[0, 0]$. Parmi les algorithmes les plus célèbres qui calculent l'alignement entre les séquences, nous citons : [67, 68].

3.1.2 L'indexation

L'idée de base est de simplement indexer le texte/dictionnaire et d'appliquer une méthode récursive pour vérifier, en un temps relativement rapide, toutes les occurrences qui sont des solutions approchées.

Il existe de nombreuses structures de données utilisées à la fois pour la recherche exacte et approchée. Parmi ces structures, le Trie est considéré comme une des structures les plus fondamentales pour représenter un dictionnaire, l'arbre de suffixes pour représenter le texte [28, 29, 30], les structures de données compressées comme le Trie compact [11, 69], le tableau des suffixes [34] compressé [70], les graphes orientés acycliques de mots $DAWG$ (directed acyclic word graphs) [71, 72], et les graphes orientés acycliques compacts de mots $CDAWG$ (compact directed acyclic word graphs) [73] ou le FM-index qui compresse les données et l'index en même temps [74, 75].

Un algorithme de recherche parcourt d'une façon récursive la hiérarchie de l'arbre à partir de la racine vers les feuilles. À chaque étape, on détermine si le chemin mène à une solution approchée ou non ; s'il n'y a pas de solution, l'algorithme retourne au nœud précédent pour choisir une autre transition sortante pour la vérifier. Cet algorithme récursif, vérifie toutes les transitions sortantes dans chaque nœud d'erreur possible.

Des algorithmes basés sur les différentes structures de données d'indexation proposent des solutions au problème de la recherche approchée de motifs, comme par exemple les algorithmes connus et présentés dans les références suivantes : [65, 76, 77, 78, 79].

3.1.3 Les méthodes basées sur le filtrage

Chaque mot ayant un certain nombre d'erreurs contient des sous-chaînes exactes (des parties qui ne contiennent pas d'erreurs). Afin d'accélérer la recherche approchée, on parcourt rapidement le dictionnaire/texte pour trouver des pièces qui peuvent être des solutions approchées possibles. L'approche consiste à faire une recherche exacte sur certains morceaux du mot requête dans le dictionnaire/texte pour obtenir des motifs candidats, et ensuite, faire les vérifications pour s'assurer s'ils représentent des solutions approchées ou non. La première étape utilise généralement des index (Trie, arbre des suffixes) pour localiser rapidement des parties exactes, puis des algorithmes classiques sont utilisés pour la vérification.

Les algorithmes de la recherche exacte sont beaucoup plus rapides que ceux de la recherche approchée. Par conséquent, les algorithmes de filtrage peuvent améliorer la recherche approchée d'une manière considérable [3].

Étant donnés un motif P et un seuil k, le motif P peut être divisé en $k+1$ morceaux $P = \{P_1, P_2, ..., P_{k+1}\}$. La première étape consiste à faire une recherche exacte sur ces $k+1$ morceaux. Au moins l'un des P_i ($i \in [1..k+1]$) est une sous-chaîne exacte dans P. La deuxième étape, consiste à vérifier s'il y a des solutions approchées [59].

Nous citons quelques algorithmes qui utilisent ce concept [80, 81, 82, 83].

3.1.4 La génération du voisinage

Une approche de génération de voisinage génère, pour un mot requête, une liste de mots ayant une certaine distance d'édition avec le mot original, puis cette liste de mots générés est recherchée en utilisant des méthodes exactes. Dans la génération de voisinage, il y a trois méthodes différentes :

Le voisinage complet : La génération de voisinage complet implique le calcul de tous les mots possibles ayant un nombre d'erreurs de distance d'édition k par rapport au mot requête. Chaque élément des mots générés est recherché dans le dictionnaire par la méthode exacte. Puisque la taille de la génération de voisinage complet est en $O(m^k \sigma^k)$ [65], cet algorithme est seulement pratique lorsque les paramètres suivants sont de petites tailles : la taille de l'alphabet σ, le nombre maximum autorisé d'erreurs k, et la longueur du mot requête m.

Le voisinage avec joker : Diminuer la taille du voisinage complet en remplaçant certains caractères par un caractère joker (wild-card). Un caractère joker peut remplacer n'importe quel autre caractère dans la chaîne.

Soit le mot requête $ABCD$, et ϕ le caractère joker, on obtient seulement 4 mots par une seule substitution : ϕBCD, $A\phi CD$, $AB\phi D$, $ABC\phi$.

Donc au lieu de substituer tous les caractères de l'alphabet dans chaque position, il suffit juste de placer un caractère joker qui peut matcher tous les caractères.

Le voisinage réduit : Dans cette approche, les mots sur l'alphabet d'origine sont associés à des mots sur un alphabet réduit, plus petit. L'objectif de cette approche consiste à compresser la génération de voisinage en diminuant la taille de l'alphabet. La technique consiste à associer l'alphabet d'origine Σ à un alphabet réduit Σ' par l'intermédiaire d'une fonction de hachage.

Il existe plusieurs algorithmes basés sur cette approche [84, 85, 86, 87, 88, 15] (voir [89] pour plus de détails sur l'approche de génération de voisinage).

3.1.5 Les algorithmes qui utilisent le hachage

Les méthodes de hachage (voir la section 2.5) sont utilisées généralement dans la version hors-ligne du problème de la recherche approchée bien qu'on puisse utiliser le hachage sans indexation et donc l'utiliser dans la version en-ligne du problème.

Parmi les algorithmes célèbres pour la recherche exacte et qui utilise le hachage : l'algorithme de "Rabin-Karp" [90]. Les performances de l'algorithme Rabin-Karp proviennent de l'utilisation d'une fonction de hachage efficace crée par Rabin-Karp. Cette fonction est utilisée par [15]. Nous avons utilisé la même fonction dans notre travail. Les détails de l'utilisation de cette fonction sont présentés dans la sous-section 3.3.1.2.

Nous donnons quelque autres algorithmes qui utilisent le hachage : [15, 91, 92, 84, 89].

3.1.6 Les méthodes de bit-parallélisme

Ce type d'algorithmes est basé sur l'exploitation du fait qu'un ordinateur manipule des données par des blocs de w bits, où w est la taille d'un seul mot mémoire. Dans le modèle standard du calcul de complexité RAM, $w = \Omega(\log n)$ avec n la taille totale du texte/dictionnaire. Généralement $w = 32$ ou $w = 64$ bits dans les ordinateurs actuels.

Cette technique est appliquée pour tous les algorithmes qui utilisent les approches d'indexation directe, de génération de voisinage, de filtrage, de programmation dynamique, parce que l'idée est d'encoder plusieurs éléments de données d'un algorithme dans un seul mot mémoire pour traiter de nombreux éléments en même temps lors d'une seule opération du processeur.

Le vocable *bit-parallélisme* traduit l'idée de traiter plusieurs éléments encodés en bits en même temps, c'est comme si on traitait plusieurs éléments en parallèle avec plusieurs processeurs.

Les algorithmes de bits parallélisme ne sont pas nouveaux dans le domaine de la recherche approchée de motifs, ils ont été proposés dans les années 1990.

Parmi les algorithmes qui utilisent cette technique, nous citons les travaux suivants : [93, 81, 94, 95, 96, 97, 98, 99, 100, 101].

3.1.7 Les méthodes hybrides

Il y a de nombreux algorithmes qui utilisent des approches hybrides (la programmation dynamique, l'indexation directe, la génération de voisinage, le filtrage) afin d'obtenir de meilleurs résultats.

Dans toutes les approches qui traitent le problème de la recherche approchée dans sa version hors-ligne, les méthodes (la génération de voisinage, le filtrage...) sont utilisées avec des index pour accélérer la recherche. Par exemple, l'indexation est utilisée avec la génération de voisinage, le filtrage, le hachage : [87, 88, 15, 102, 82].

Le bit-parallélisme peut aussi être combiné avec les autres approches afin d'accélérer le traitement comme nous l'avons expliqué dans la sous-section 3.1.6. Par exemple, les algorithmes expliqués dans [95, 98] combinent le bit-parallélisme avec la programmation dynamique.

Le calcul de la distance d'édition entre deux mots x et y se fait avec la programmation dynamique, cela implique généralement que les différentes méthodes (le filtrage, la génération de voisinage ...) vérifient la distance d'édition des derniers mots avec le mot requête. Nous citons quelques exemples pour les algorithmes qui utilisent la programmation dynamique avec les autres approches [65, 66, 103].

3.1.8 Le parallélisme

Le parallélisme permet de traiter des informations de manière simultanée sur plusieurs unités de calculs. Au lieu d'exécuter les tâches sur une seule unité de calcul, on les divise sur plusieurs. Le but est de réaliser le plus grand nombre d'opérations en un temps record (voir [104, 105]).

Il existe un modèle de machine parallèle qui fût très en vogue dans les années 80 (un modèle théorique) appelée PRAM (Parallel Random Access Machine) [53]. Nous citons quelque algorithmes datant de cette époque : [106, 107] et quelques travaux récents : [108, 109, 110].

3.2 Quelques résultats connus

3.2.1 K-errata trie de Cole et al

Dans [102] , Cole et al ont proposé une nouvelle structure de données appelée K-errata Trie. Cette structure permet de résoudre le problème de la recherche approchée pour n'importe quel nombre d'erreurs k.

La solution est basée sur le concept de la génération de voisinage et l'utilisation du Trie avec erreur pour créer l'index K-errata Trie.

La méthode consiste à créer un arbre de suffixes et ensuite de trouver les "centroid path"[1] pour décomposer des sous-arbres en "centroid path decompositions". La méthode insère des erreurs dans l'arbre de suffixes, puis fusionne récursivement les sous-arbres de "centroid path decompositions".

Rappelons les définitions du *Centroid paths* et du *Centroid path decompositions* avant d'expliquer brièvement la construction de K-errata Trie.

Centroid paths (chemins barycentre) : le chemin commence à la racine de l'arbre T. Chaque nœud nd sur le chemin, se branche au nœud fils qui a le plus grand nombre de feuilles dans son sous-arbre (donc ce nœud devient la racine de ce sous-arbre).

Centroid path decompositions (décomposition de chemin barycentre) : après l'identification des *centroid path*, l'arbre est décomposé de façon récursive en sous-arbres. Chaque transition qui sort du *centroid path* va représenter un sous-arbre.

Soit C un des *centoid path* résultants et soit v un nœud sur le *centroid path* C. On forme un nouveau sous-arbre qui commence avec un caractère joker qui sort de v, ce sous-arbre comprenant la fusion de tous les sous-arbres qui sortent de v et qui ne sont pas sur le *centroid path* en remplaçant le premier caractère de chaque sous-arbre par un nouveau symbole joker ϕ ($\phi \notin \Sigma$).

Ainsi, 1-errata Trie est créé. Pour créer le 2-errata Trie, on prend chaque sous-arbre qui commence par ϕ, et on applique la même procédure, c-à-d, trouver les *centroid path*, et créer des sous-arbres avec le caractère joker, ensuite, fusionner tous les sous-arbres qui sortent d'un nœud sur le *centroid path*. On applique cette méthode d'une façon récursive afin de créer le k-errata Trie.

La complexité temporelle et spatiale atteinte est comme suit :

La recherche approchée dans un texte : l'espace mémoire de la structure de données : $O(n\frac{(c_1 log\, n)^k}{k!})$. Le temps de la construction : $O(n\frac{(c_1 log\, n)^k}{k!})$. Le temps de la requête (3 types d'erreurs) : $O(\frac{(c_2 log\, n)^k\, log\, log\, n}{k!} + m + 3^k \times occ)$. c_1, c_2 sont des constants > 1, et le nombre d'erreurs k est $k \leq log\, n$.

La recherche approchée dans un dictionnaire : l'espace mémoire de la structure de données : $O(n\, log^k\, n)$. Le temps de la construction : $O(n\, log^k\, n + n\, log\, \Sigma)$. Le temps de la recherche : $2^k log\, log\, n + m + occ$.

1. centroid path : Le barycentre, ou centre de masse, parfois appelé centre de gravité.

3.2.2 La méthode de Buchsbaum et al

Buchsbaum et al ont travaillé dans [111] sur le problème de la recherche dans une plage [112] (rechercher un ensemble d'éléments S dans un ensemble S' plus grand). Ils ont introduit le produit vectoriel des arbres "tree cross-product", leur but est de faire un pré-traitement sur des arbres et des hypergraphes [113] afin de rendre le traitement des requêtes plus efficace. Leur résultat est appliqué dans la visualisation des graphes, l'analyse des logiciels et la recherche de motifs.

Buchsbaum et al. améliorent la méthode de Amir et al [83] en augmentant les structures de données Trie et Trie inversé. La méthode consiste à ajouter des arcs entre les feuilles des deux arbres qui ont les mêmes étiquettes. Ensuite, ils utilisent la méthode sur les graphes pour faire l'intersection des nœuds trouvés dans le Trie et le Trie inversé dans l'étape 3 de Amir et al.

La complexité temporelle et spatiale atteinte par Buchsbaum et al. est comme suit : Pour un texte T de taille n et un mot requête d'une longueur m, l'espace mémoire utilisé est de l'ordre de $O(n \log n)$. Le temps de pré-traitement est de $O(n \log n)$. Le temps de la recherche avec $k = 1$ erreur est de $O(m \log \log n + occrs)$ ($occrs$ représente le nombre d'occurrences de la solution).

3.3 L'algorithme de D.Belazzougui

D.Belazzougui dans [15] a proposé une nouvelle structure de données qui permet de résoudre le problème de la recherche approchée pour $k = 1$ erreur dans un temps proportionnel à la longueur m du mot requête $O(m + occrs)$ ($occrs$ le nombre d'occurrences de la solution), et dans un espace mémoire optimal où l'index occupe $O(n(lg(n)loglog(n))^2)$ avec n la taille de texte.

Pour arriver à ce résultat, il utilise plusieurs outils combinés entre eux : les fonctions de hachage parfait minimal (mphf), les tableaux de hachage dynamique, le Trie et le Trie inversé, les tableaux de bits, et la structure de données préfixe-sum.

Étant donnée une collection S de d motifs (clés) de longueur n caractères. On note le mot requête par q et sa longueur par $m = |q|$. On met σ comme la taille de l'alphabet Σ, donc $\sigma = |\Sigma|$).

La solution simple (naïve) pour répondre aux requêtes approchées sur le mot q est de chercher exhaustivement pour tous les mots possibles qui peuvent être obtenues par l'opération de distance d'édition avec une seule erreur sur le mot requête q. Si un dictionnaire exact est utilisé donc cela prend un temps de $O(m)$ pour répondre à chaque requête candidats pour les différentes combinaisons des mots générés avec les 3 opérations de distance d'édition, donc cela donne au total un temps de $O(m^2\sigma)$ pour répondre à tous les mots candidats.

Afin de réduire le temps des requêtes, D.Bellazougui a réduit le nombre de mots

candidats pour l'insertion et la substitution de $O(m\sigma)$ à juste $O(m)$, par l'utilisation d'une structure de données "dictionnaire des listes de substitution" qui donne un caractère candidat pour chaque position de l'insertion ou la substitution dans le mot q. Ainsi, au lieu de tester pour chaque position σ caractères, cette structure nous donne une liste de caractères qui lorsqu'on les teste donnent tous des solutions.

Il effectue un pré-traitement pour le mot q, avec un temps $O(m)$, pour pouvoir vérifier (avec la comparaison exacte) chaque mot candidat dans un temps de $O(1)$ au lieu $O(m)$.

L'idée utilisée pour améliorer le temps de comparaison d'une manière déterministe se base sur le fait que l'on peut comparer deux blocs mémoires dans un temps seulement $O(1)$, tel que chaque bloc est de taille w (un mot mémoire), cela permet de comparer deux morceaux de chaîne de caractères tels que chaque morceau contient u caractères où $ub = O(w)$ et b est le nombre de bits pour un seul caractère. Ceci implique que les mots de moins de u caractères peuvent être comparés dans un temps O(1).

Pour améliorer le temps de comparaison pour les mots longs, l'idée est de calculer les signatures de tous les préfixes et suffixes des mots dans le dictionnaire dont les longueurs sont multiples de u. Ces signatures occupent moins que w bits, et l'espace total utilisé par les signatures est ainsi du même ordre que l'espace occupé par le mot. La comparaison d'un mot candidat comprend la comparaison des signatures des préfixes et des suffixes de q, avec les signatures des préfixes et des suffixes de mot s de dictionnaire. Le nombre de comparaisons est constant, impliquant un nombre de blocs de moins de $|q|/u$.

Afin d'obtenir des signatures déterministes (non-collision) pour les préfixes et les suffixes, on utilise un Trie et un Trie inversé construit sur l'ensemble des mots du dictionnaire. Comme cela, on peut trouver des signatures déterministes pour n'importe quels préfixe et suffixe de chaque mot du dictionnaire.

3.3.1 La construction de la structure de données

La structure de données utilise les composants suivants :

3.3.1.1 Un Trie et un Trie inversé

Étant donnée une collection de mots S de taille total n caractères (nb bits), on veut construire un Trie sur S qui occupe un espace de $O(n)$, et qui peut être traversé dans un temps de $O(|q|)$ pour un mot requête q.

Pour la requête de mot q, on a besoin que le Trie renvoie un numéro identifiant unique pour chaque nœud traversé pendant la recherche de motif q. Cet identifiant doit occuper un espace de $O(w)$ bits. Dans chaque nœud on ajoute un identifiant unique (un nombre), la racine à comme indice 0, ensuite, on donne des indices au fils gauche, ensuite, les fils droite.

Ce problème peut être résolu par l'utilisation d'une structure de données décrite dans [114]. La structure de données peut renvoyer un nombre entier unique dans l'intervalle $[1, Nb\_nd]$ pour chaque nœud traversé, où Nb_nd est le nombre total des nœuds.

On a besoin de deux Tries :

- Un Trie Tr construit sur l'ensemble des mots S. Le rôle principal de ce Trie est qu'il permet de comparer les préfixes des clefs dans S avec les préfixes de n'importe quel mot requête q, et cela juste en comparant les identifiants retournés par le Trie pour les deux mots. C'est la partie essentielle de l'algorithme qui permet d'obtenir un temps de requête borné en $O(1)$ pour chaque mot candidat.
- Un Trie inversé $\overline{Tr}$ construit sur l'ensemble $\bar{S}$, afin de comparer les suffixes des clefs de S avec les suffixes de n'importe quel mot q, et cela juste on comparant les identifiants retournés par le Trie inversé pour les deux mots.

On note que le parcours du Trie, ou du Trie renversé pour un motif q retourne au plus $|q|$ identifiants dans l'intervalle $[1, Nb\_nd]$ correspondant aux nœuds traversés.

3.3.1.2 Un dictionnaire exact de mot avec une fonction du hachage parfait minimal

Étant donné un ensemble S de d mots de taille totale n. En utilisant la fonction du hachage parfait minimal, on peut construire un dictionnaire (index du dictionnaire) qui occupe un espace optimal afin de répondre à des requêtes exactes dans un temps proportionnelle à la longueur de mot requête. Le dictionnaire occupe $O(nb)$ bits et la recherche prend $O(m)$, où m est la longueur de mot requête.

La 1re étape de construction du dictionnaire consiste à construire la fonction du hachage parfait minimal $mphf$ de l'ensemble S.

La $mphf$ mappe chaque mot de S vers un numéro distinct dans l'intervalle $[0, d-1]$.

On stocke les mots l'un après l'autre dans un tableau T consécutif dans l'ordre donné par $mphf$. Le premier caractère du mot s_i mappé par $mphf$ dans le tableau T est donné par la formule suivante :

$$Pos(s_i) = \sum_{j=0}^{j<i} |s_j|$$

Dans le but de trouver la position d'un mot dans T, on utilise la structure de données prefix-sum qui donne pour n'importe quelle position i la somme des longueurs de tous les mots mappés dans T par $mphf$ avant la position i.

La construction de la $mphf$ **:** soit un nombre premier P tel que $P > n \times d^2$, et $P > 2^b$ (n : le nombre total des caractères de l'ensemble S, d : le nombre de motifs de S, b : le nombre de bits d'un seul caractère). Avant de construire la fonction de hachage parfait minimal $mphf$, on doit mapper l'ensemble de l'alphabet de l'ensemble S vers des valeurs entières, chaque caractère c va être mappé à une valeur unique v avec $v \in [0, P-1]$.

La première étape dans la construction est de mapper l'ensemble S de d mots aux valeurs du hachage distinctes dans l'intervalle $[0, P-1]$. Pour cela, on utilise une fonction de hachage h paramétrée avec un nombre entier t aléatoirement choisi tel que $t \in [0, P-1]$.

On calcule la valeur de hachage pour un mot s en utilisant la formule suivante :

$$h(s) = (s[1] \otimes t) \oplus (s[2] \otimes t^2) \oplus ... \oplus (s[m] \otimes t^m)$$

où l'addition et la multiplication sont fait modulo P, (les caractères d'une chaine s sont considérés comme des nombres entiers dans l'intervalle $[0, P-1]$).

Une fois qu'on a calculé les valeurs de hachage liées à tous les mots de S, on vérifie si on a des collisions entre les valeurs du hachage générées pour les clefs (les mots) de l'ensemble S. Si c'est le cas, on répète le calcul de l'ensemble des valeurs de hachage en utilisant une nouvelle valeur t aléatoirement choisie. Ce traitement est répété jusqu'à obtenir un ensemble sans collision. Le temps total prévu pour générer les valeurs de hachage **sans collisions** est $O(n)$.

Une fois que l'on a mappé tous les mots de S aux nombres distincts, on utilise ces nombres comme clefs pour construire la $mphf$. On utilise une fonction de hachage qui mappe ces d valeurs vers un tableau de taille de d éléments.

Comment stocker les mots dans le dictionnaire : Le détail final dans la construction du dictionnaire exact basé sur $mphf$, est comment stocker les mots dans le dictionnaire.

Les mots courts et longs sont traités différemment. Les mots courts (de longueur $m \leq w$) sont stockés tels quels sans modification dans le dictionnaire.

Pour un mot long s de taille $m > w$, on stocke un mot modifié s' de longueur $3m$ caractères. On divise le mot s' en trois parties consécutives $s_1^{'}$, $s_l^{'}$ et $s_r^{'}$, chacun de longueur de mb bits.

Le mot s_1' va contenir une copie du mot s. Il reste les deux parties s_l' et s_r', on va expliquer comment les composer.

À cet effet, premièrement on met une valeur $u = \lceil log(n)/b \rceil$ qui représente un bloc (on peut le considérer comme un mot mémoire).

On considère le mot s_l' et s_r' comme des tableaux de $m' = \lfloor m/u \rfloor$ éléments avec ub bits pour chacun (noter que $log(n) \leq ub < log(n) + b$), on ignore les bits de remplissage (le

dernier $mb - m'ub$ bits). Donc chaque tableau va être décomposé à un nombre de cases m', tel que m' représente le nombre de blocs ou les parties dans le mot s, (si on considère que $u = 4$ caractères et on a un mot de 12 caractères, alors on aura 3 partie, $m' = 3$). Les éléments du tableau de s'_l et s'_r sont numérotés en commençant par 1.

On met $s'_l[i]$ à la valeur $L[ui]$ (l'identifiant du nœud atteint à l'étape $u \times i$ en traversant le Trie Tr pour le mot s), et on met $s'_r[i]$ la valeur $R[ui]$ (l'identifiant du nœud atteint à l'étape $u \times i$ en traversant le Trie inversé $\overline{Tr}$ pour le mot $\bar{s}$) ($i \in [1..m']$). Cela signifie que dans chaque case on met l'identifiant correspond au longueur de préfixe du mot $i \times u$ (exemple : soit un mot de longueur 12 et $u = 4$, donc lorsque $i = 1$, on met l'identifiant des 4 premiers caractères de mot, ensuite, $i = 2$, donc la partie avec 8 caractères, $i = 3$ donc la partie avec 12 caractères).

Bien évidemment, on stocke les mots (les mots courts non modifiés, et les mots longs qui sont modifiés) dans un tableau contigu dans l'ordre donné par la $mphf$, et on utilise la structure de données prefix-sum pour stocker l'emplacement de chaque mot.

Utiliser la structure de données prefix-sum : Tous les mots du dictionnaire sont rangés dans un tableau contigu l'un après l'autre selon l'ordre donné par $mphf$. Chaque mot court i occupe juste sa taille qui est $|m_i|$, et chaque mot long j occupe trois fois sa taille $3 \times |m_j|$.

Noter que les mots courts et les mots longs peuvent avoir des longueurs différentes. Cela veut dire que le tableau qui stocke ces mots n'est pas divisé en blocs de même taille. Le problème est comment ranger ces mots dans le tableau, et le plus important comment les retrouver par la suite ? Pour cela, on utilise une structure de données supplémentaire appelée **prefix-sum** dans le but de garder la position donnée au mot par $mphf$ et retrouver sa position dans la mémoire.

On utilise un tableau intermédiaire Tab_1 avant de créer le tableau prefix-sum. Pour chaque mot on récupère sa position depuis le tableau de hachage parfait minimal calculé précédemment, et on met sa taille dans la case correspondante à l'indice donnée par $mphf$ dans Tab_1.

Comme cela, la position du mot s_i dans la mémoire est après tous les mots (s_j, avec $j \in [1..i-1]$) précédents, donc la somme de toutes les tailles des mots précédents dans le tableau. Donc $Pos(s_i) = \sum_{j=0}^{j<i} |s_j|$.

On construit le tableau prefixe-sum qui va contenir pour chaque case de Tab_1 la somme de toutes les tailles précédentes.

Lorsqu'on cherche un mot q, on calcule son hachage parfait minimal pour trouver la position dans le tableau prefix-sum, ensuite, on récupère sa position dans la mémoire. La taille de mot dans la mémoire est la différence entre les deux cases successives de tableau prefix-sum, si l'indice de mot est i donc sa taille est le contenu de la case $i + 1$ moins le

contenu de la case i.

3.3.1.3 Le dictionnaire des listes de substitution

Le but de cette structure de données est de réduire le nombre de possibilité des mots candidat pour l'insertion et la substitution de $O(m\sigma)$ à juste $O(m)$ avec m est la taille de mot requête q, donc on élimine toutes les combinaisons des caractères possibles de la solution naïve qui est σ de chaque position.

Soit un ensemble $S^{'} = \{x_1, x_2, ..., x_{d'}\}$ de d' clés et de n' caractères au total. Soit la collection non vide $L = \{l_1, l_2, ..., l_{d'}\}$ où chaque l_i est une liste d'éléments (caractères) qu'on l'associe avec une clef x_i de S'. Le nombre total d'éléments stocker dans toutes les listes est n' où chaque élément est de taille b bits.

Le but est de construire une structure de données de taille $O(n'b)$ bits qui supporte les opérations suivantes dans un temps constant :

- $list\_element(x, i)$, cette opération retourne l'élément numéro i (les éléments de la liste sont numérotés à partir de 1) de la liste associer avec la clef x si $x \in S'$. L'élément retourné est non défini si $x \notin S'$.
- $size\_of(x)$, cette opération retourne le nombre d'éléments dans la liste associée avec x si $x \in S'$. Le résultat de l'opération est non défini si $x \notin S'$.

La solution est basée sur la structure de données prefix-som et le hachage parfait minimal, et elle supporte les deux opérations précédentes dans un temps constant. La solution a une certaine similarité avec la solution du dictionnaire exact avec hachage parfait minimal décrite précédemment.

L'idée est de stocker tous les caractères possibles qui peuvent donner des solutions approchées pour l'insertion et/ou la substitution dans une position d'erreur i. Soit un mot x du dictionnaire qui a un préfixe u et un suffixe v et un caractère au milieu c, donc $x = ucv$. Si on suppose qu'on a un mot requête $q = u\phi v$, et ϕ la position de l'erreur (on prend le cas de la substitution). Le mot $x = ucv$ est une solution approchée, on substitue le caractère c par ϕ. Ainsi, on a juste un seul caractère possible pour cette position. Si dans le dictionnaire on a un autre mot de la même forme $y = uc_2v$, alors ce mot aussi représente une 2^e solution (on substitue le caractère c_2 par ϕ), dans ce cas on a deux caractères qui donnent deux solutions. Il y a des cas où on a juste une solution, et des cas où on trouve plusieurs solutions. Ces caractères vont être stockés dans un tableau contigu, l'ordre est donnée par une fonction de hachage parfait minimal, les caractères qui donnent des solutions à la même requête dans la même position (comme l'exemple précédent c et c_2) sont rangés dans la même place et donc il vont représenter une liste de solutions. Pour trouver ces caractères lors de la recherche, on utilise la structure de données prefix-sum afin de trouver la position dans la mémoire donc le début et la fin de chaque liste de caractères qui donne des solutions pour la même position.

La première étape consiste à construire la fonction de hachage parfait minimal sur l'ensemble $S^{'}$. On note par x_i l'élément mappé avec $mphf$ de $S^{'}$ vers la position i. On stocke les éléments de chaque liste d'une façon contiguë dans un tableau T'. La position d'élément numéro j de la liste qui est associée à la clef x_i est donnée par :

$$Pos(i,j) = (\sum_{k=0}^{k<i} |x_k|) + j$$

Le parcours de la liste liée à une clef q, commence par le calcul de $mphf$ qui donne un numéro i. Ensuite, on interroge la structure de données prefix-sum qui donne la position dans la mémoire pos_i. L'élément numéro j de la liste associée avec la clef q est dans la position $T'[pos_i + j]$.

Pour avoir le nombre d'éléments qui est dans la liste associée avec la clef q, on interroge simplement la structure de données prefix-sum avec les numéros i et $i+1$ pour nous donner pos_i et pos_{i+1}. Le nombre d'éléments dans la liste associée avec q est simplement $T'[pos_{i+1}] - T'[pos_i]$.

On détaille maintenant la construction de cette structure de données. On traite successivement chaque mot s de l'ensemble S (soit m la longueur de mot s). On stocke deux tableaux temporaires de nombres entiers $L[0..m]$ et $R[0..m]$ pour chaque mot, les nombres sont dans l'intervalle $[0, n]$ avec n le nombre total de tous les caractères du dictionnaire.

On met $L[0] = R[0] = 0$. En premier on parcourt le Trie Tr pour le mot s, et on stocke dans la case $L[i]$ l'identifiant du nœud atteint à l'étape i (les étapes sont numérotées à partir de 1).

On fait la même chose pour générer les éléments $R[1..m]$. On parcourt le Trie renversé $\overline{Tr}$ pour le mot $\bar{s}$ (l'inverse du mot s) et on stocke l'identifiant du nœud atteint à l'étape i dans la case $R[i]$.

Pour chaque mot s de longueur m pour lequel on a calculé les tableaux L et R, on ajoute le caractère $s[i]$ à la liste correspondante à la paire $(L[i-1], R[m-i])$ pour chaque i, tel que $1 \leq i \leq m$.

Donc on aura pour chaque mot une liste $\{(L[0], R[m-1], s[1]), (L[1], R[m-2], s[2]), ..., (L[m-1], R[0], s[m])\}$.

Ensuite, pour chaque paire $(L[i], R[j])$ on calcule le hachage parfait minimal. Chaque caractère $s[i]$ est stocké dans la position donnée par $mphf$ de $(L[i-1], R[m-i])$.

En fait, on insère les éléments dans le dictionnaire qui est implémenté en utilisant une table de hachage dynamique temporaire et des listes liées.

Si deux ou plusieurs mots ont la même valeur paire $(L[i], R[j])$ cela vaut dire qu'ils ont le même préfixe et le même suffixe, juste le caractère à la position i qui est différent (comme on a expliqué dans un exemple précédent pour c et c_2), donc leurs valeurs de hachages sont les mêmes, donc tous les caractères liés à la paire $(L[i], R[j])$ vont être stockés dans la même position (la même liste).

Par la suite, on peut construire le dictionnaire des listes de substitution depuis la table de hachage temporaire et les listes liées en utilisant la méthode décrite précédemment dans la construction de dictionnaire exact (avec la structure prefix-sum).

On va créer un seul tableau contigu, et on stocke les listes des éléments (des caractères) liste après liste, il y a des listes qui contiennent juste un seul élément, et d'autres qui en contiennent plusieurs. L'ordre de ces listes est donné par $mphf$ (l'ordre des listes dans le tableau de hachage dynamique temporaire).

À la fin, on construit le tableau prefix-sum, pour cela on utilise un tableau temporaire qui va contenir dans chaque case k, la taille de la liste numéro k, ensuite, on construit le tableau prefix-sum, chaque case k va indiquer le début de la liste k dans la mémoire (la somme de toutes les tailles des listes précédentes sachant que chaque élément occupe b bits).

3.3.2 La recherche du mot requête

Étant donné un mot requête q de longueur m.

En premier, on calcule $\bar{q}$ qui est le mot inverse de mot q.

Ensuite, on calcule les tableaux $L[0..m]$ et $R[0..m]$. Pour cela, on met $L[0] = R[0] = 0$. Ensuite, on parcourt le Trie Tr pour le mot q et on met l'identifiant de nœud atteint dans l'étape i dans la case $L[i]$. Si la recherche s'arrête à l'étape i alors on place une valeur spéciale $\perp$ dans les cases qui reste $L[j]$ pour $j \in [i+1, m]$. De même, on remplit le tableau R en parcourant le Trie inversé $\bar{Tr}$ avec le mot $\bar{q}$ et on met l'identifiant de nœud atteint à l'étape i dans la case $R[i]$. Si la recherche s'arrête à l'étape i on place une valeur spéciale $\perp$ dans les cases qui reste $R[j]$ pour $j \in [i+1, m]$.

On prépare aussi trois tableaux en plus, notés par $A_t[0, m+1]$, $F[0..m]$ et $G[1..m+1]$:

1. Le tableau A_t stocke toutes les puissances de t jusqu'à t^{m+1}. En premier on met $A_t[0] = 1$, ensuite, on met $A_t[i] = A_t[i-1] \otimes t$ pour chaque i dans l'intervalle $[1, m+1]$.
2. Pour générer le tableau F, on met en premier $F[0] = 0$, ensuite, on met $F[i] = F[i-1] \oplus (q[i] \otimes A_t[i])$ pour chaque i dans l'intervalle $[1..m]$. (F : tableau qui calcule les valeurs du hachage pour les préfixes de q).
3. Pour générer le tableau G, en premier on met $G[m+1] = 0$, ensuite, on met $G[i] = (G[i+1] \oplus q[i]) \otimes t$ pour chaque i dans l'intervalle $[1..m]$. (G : tableau qui calcule les valeurs du hachage pour les suffixe de mot q).

On a quatre types de mots dans le dictionnaire qui pourrait correspondre au mot requête : un mot qui est à une distance 0 (exact), et les mots du dictionnaire qui peuvent être obtenus par l'application de l'un des trois types d'erreurs de distance d'édition (suppression, insertion, et substitution).

3.3.2.1 La recherche exacte

La recherche exacte pour les mots avec une distance d'édition 0. On fait simplement un accès au dictionnaire exact pour le mot requête q. Pour cela, on calcule la valeur du hachage de mot q.
On sait que $h(q) = (q[1] \otimes A_t[1]) \oplus (q[2] \otimes A_t[2]) \cdots \oplus (q[k] \otimes A_t[k]) = F[k] = G[1]$. La valeur du hachage est déjà pré-calculée, lorsque on a préparé les deux tableaux F et G. Ensuite, avec cette valeur de hachage on calcule la valeur de hachage parfait minimal $mphf$ qui donne la position i dans le tableau prefix-sum. On récupère la position d'accès au dictionnaire exact (le tableau contigu qui stocke les mots du dictionnaire). La position se trouve dans la case de tableau prefix-sum numéro i. La taille de mot dans la mémoire est calculée par le contenu du tableau prefix-sum $i + 1$ moins le contenu de la case à la position i.

Si le mot trouvé est un mot court (sa longueur est $< w$), soit ce mot court s. Donc on le compare directement avec le mot requête q , la comparaison prend un temps de $O(1)$, car la vérification d'un seul mot mémoire w ce fait d'un seul coup. Si les deux mots sont égaux ($s = q$) alors on retourne s comme un résultat exact.

Dans le cas contraire, on a un mot long, sa longueur est au moins $3w$. Il faut se rappeler que dans la phase de construction, si le mot du dictionnaire s est un mot court on le stocke tel quel, et si sa longueur est $> w$, on le traite et on le stocke dans un nouveau mot s' qui occupe un espace de 3 fois sa taille. Dans ce cas on a un mot s' qui contient trois parties s'_1, s'_l et s'_r, le mot est stocké dans la première partie s'_1.

On peut comparer directement q avec s'_1 dans un temps $O(m)$ et si on trouve qu'ils sont égaux on retourne s'_1 comme résultat exact. Mais le but est de faire cette comparaison dans un temps de $O(1)$. Pour cela, on calcule le nombre de blocs dans le mot requête q donc $l'_q = \lfloor (m-1)/u \rfloor$. Ensuite, on vérifie les conditions suivantes :

- La longueur de s' est $3m$. Car s' contient trois parties égaux. Cette vérification prend un temps de $O(1)$.
- $s'_l[l'_q] = L[u \times l'_q]$, vérifier si l'identifiant de mot s'_1 (pour les caractères qui constituent l'_q blocs) est le même que l'identifiant de mot q pour le même nombre de blocs. Les deux identifiants sont récupérés du Trie, et donc on a une égalité si et seulement si les caractères sont égaux. Cette opération prend un temps de $O(1)$.
- $q[u \times l'_q + 1..m] = s'_1[u \times l'_q + 1..m]$, tester les caractères qui restent qui ne sont pas dans les blocs $u \times l'_q$, le nombre de ces caractères est moins qu'un bloc u, donc la comparaison est en $O(1)$.

Le pré-traitement des tableaux A_t, F, G, L, R est en $O(m)$ chacun. Si on considère qu'on a que la recherche exacte seule, le temps de recherche est en $O(m)$.

Le temps de comparaison est en $O(1)$ car on ne compte pas le temps de pré-traitement qui est fait spécialement pour qu'on puisse faire la comparaison des différents mots candidats dans la recherche approchée en $O(1)$.

3.3.2.2 la recherche approchée

L'erreur de type insertion : on commence par l'erreur de type insertion, les mots obtenus par l'insertion d'un seul caractère dans q.

On va faire $m+1$ étapes pour $i \in [0, m]$. À chaque étape i on fait un accès au dictionnaire des listes de substitution pour trouver la liste associée avec la paire $(L[i], R[m-i])$ si et seulement si $L[i] \neq \perp$ et $R[m-i] \neq \perp$ (on suppose qu'on a inséré un caractère après la position i).

Pour trouver cette liste il suffit juste de calculer le hachage parfait minimal de la paire $(L[i], R[m-i])$ pour avoir une position dans le tableau prefix-sum qui nous donne la position de la liste dans la mémoire (et on déduit aussi sa taille).

Pour vérifier la validité de la liste retournée, on doit seulement vérifier son premier élément. Si le premier élément est valide, alors on conclut que la liste existe vraiment et que tous les éléments restants sont également valides.

Soit c le premier élément (caractère) de la liste associée avec la paire $(L[i], R[m-i])$. On doit faire un accès au dictionnaire pour vérifier l'existence de mot $q' = q[1..i]\, c\, q[i+1..m+1]$. Les étapes sont les mêmes que celles de la recherche exacte.

On a $h(q') = F[i] \oplus (c \oplus G[i+1]) \otimes A_t[i+1]$. On utilise cette valeur de hachage $h(q')$ afin de calculer le $mphf$, ensuite, on utilise la position retournée par $mphf$ pour accéder au dictionnaire à travers le tableau prefix-sum.

Si le mot retourné est un mot court (donc s), on peut directement le comparer à q' dans un temps de $O(1)$, et on dit qu'on a trouvé une solution dans le cas où ils sont égaux.

Si c'est un mot long (donc s' de longueur au moins $3w$), on divise s' en trois parties égaux $(s'_1), (s'_l)$ et (s'_r). Ce dont on a besoin maintenant est de pouvoir comparer les mots s'_1 et q', on peut le faire d'une façon naïve dans un temps $O(m)$. Cependant, on peut comparer s'_1 et q' dans un temps de $O(1)$.

Pour pouvoir comparer s'_1 et q' dans un temps constant, on utilise les parties s'_l et s'_r qui sont considérées comme des tableaux d'éléments de longueur $u \times b \geq log(n)$ bits chacun. $u = \lceil log(n)/b \rceil$, u représente un bloc qu'on peut vérifier d'un seul coup (un bloc de u caractères représente un mot mémoire w).

On initialise la variable l'_q par le nombre de blocs qu'on a dans le préfixe $q'[0..i]$ ($l'_q = \lfloor i/u \rfloor$), et de même, on initialise r'_q par le nombre de blocs qu'on dans le suffixe $q'[m-i..m]$ ($r'_q = \lfloor (m-i)/u \rfloor$).

On retourne un match si et seulement si toutes les conditions suivantes sont réunies :

- La longueur de s' est $3(m+1)$, donc les tailles du mot requête et s' sont les mêmes. La taille du mot requête est m, on met $+1$ car on a une insertion.
- $l'_q = 0$, dans le cas où l'insertion est au début donc on a que la partie suffixe après l'insertion,

  ou

$s'_l[l'_q] = L[u \times l'_q]$ [1]. Ce test est utilisé afin de vérifier si le préfixe avant la position de l'insertion ($u \times l'_q$ caractères de s'_1) et le préfixe de q' sont les mêmes. Ce test prend un temps de $O(1)$. Il faut se rappeler que s'_l est un tableau qui contient les identifiants des parties multiples de u de s'_1 dans le Trie, et L aussi contient tous les identifiants de mot requête pour chaque position, mais qui ne sont pas multiples de u.

On peut facilement voir que $s'_l[l'_q] = L[u \times l'_q]$ est vrai si et seulement si l'identifiant retourné du Trie Tr est identique pour le premier préfixe ($u \times l'_q$ caractères) de s'_1 et le premier préfixe $u \times l'_q$ de q, cela est vrai seulement si leurs caractères de ces deux parties sont les mêmes.

- $q[u \times l'_q + 1..i] = s'_1[u \times l'_q + 1..i]$. Ce test se fait dans un temps de $O(1)$ car on compare deux mots qui ont une longueur au max $(u-1)b = O(w)$ bits. Ce test est fait pour couvrir la partie des caractères qui reste jusqu'à la position i et qui ne rentre pas dans les blocs $u \times l'_q$.

  Exemple : si on a un préfixe de taille 10 et $u = 4$, dans ce cas $l'_q = 10/4 = 2$, donc on a 2 blocs qui couvrent 8 caractères ($u \times l'_q = 4 \times 2$), ces blocs on les teste par $s'_l[l'_q] = L[u \times l'_q]$, et il reste 2 caractères, donc on les teste séparément.
- $s'_1[i+1] = c$. Ce test prend clairement un temps de O(1). (le test de caractère de l'insertion).
- $q[i+1..m-u \times r'_q] = s'_1[i+2..m+1-u \times r'_q]$. Ce test prend un temps de O(1) car on compare deux mots de taille $O(w)$ bits. Le test des caractères qui ne rentre pas dans les blocs de suffixe de $[m-i..i]$.
- $r'_q = 0$ si l'insertion est à la fin, donc on a que le préfixe,

  ou

  $s'_r[r'_q] = R[u.r'_q]$. Ce test est fait pour vérifier si les derniers $u \times r'_q$ caractères de s'_1 et q sont les mêmes. Ce teste prend un temps de $O(1)$. $s'_r[r'_q] = R[u \times r'_q]$ est vrai si et seulement si l'identifiant retourné du Trie inversé $\overline{Tr}$ est identique pour les derniers $u \times r'_q$ caractères (suffixe) de s'_1 et les derniers $u \times r'_q$ caractères de q, et cela est vrai seulement si leurs caractères sont les mêmes.

Noter qu'à chaque étape i, on a besoin de vérifier seulement le mot obtenu en insérant à la position i le premier caractère de la liste retournée par le dictionnaire des listes de substitution.

Si on a une égalité, on peut continuer à récupérer les mots obtenus en insérant les caractères restants de la liste à la position i, et on est sûr d'avoir une égalité pour les mots obtenus.

La récupération de chaque élément supplémentaire de la liste prend un temps de O(1).

1. D.Belazzougui dans son article [15], à fait une petite erreur d'indice dans la comparaison, il compare $s'_l[l'_q] = L[l'_q]$ au lieu de $s'_l[l'_q] = L[u \times l'_q]$ (donc il a mis $L[l'_q]$ au lieu de $L[u.l'_q]$). Car dans le tableau s'_l on a les identifiants multiples de u (donc $i \times u$), alors que dans le tableau L on a les identifiants de chaque position i.

La vérification pour les deux types d'erreurs (la substitution et la suppression) prend aussi un temps de $O(m)$. La procédure pour vérifier ces erreurs est semblable à la procédure pour vérifier l'insertion. L'explication est décrite dans le paragraphe suivant. La récupération des mots valides pour chaque erreur prend un temps supplémentaire de O(1) par élément. Ainsi le temps total de la requête est de $O(m + occrs)$.

L'erreur de type substitution : Le cas de la substitution est très semblable à celui de l'insertion. En premier, on récupère une liste de caractères depuis le dictionnaire des listes de substitution pour chaque paire $(L[i-1], R[m-i])$ pour une position donnée $i \in [1, m]$, avec $L[i-1] \neq \perp$ et $R[k-i] \neq \perp$.

Soit c le premier élément de la liste associé avec $(L[i-1], R[m-i])$, et soit $q' = q[1..i-1]\, c\, q[i+1..m]$. On a $h(q') = F[i-1] \oplus (c \oplus G[i+1]) \otimes A_t[i]$.

On fait un accès au dictionnaire en utilisant $h(q')$. Si le mot retourné est un mot court (donc s), on le compare directement avec q' en $O(1)$. Autrement (on a un mot long s'), on décompose s' en trois parties égaux s'_1,s'_l et s'_r. Et on compare s'_1 avec q'. Pour cela, on initialise les deux variables $l'_q = \lfloor (i-1)/u \rfloor$, et $r'_q = \lfloor (m-i)/u \rfloor$. On retourne une égalité si et seulement si les conditions suivantes sont réunies :

- La longueur de s' est $3m$.
- $l'_q = 0 \vee s'_l[l'_q] = L[u \times l'_q]$.
- $q[u \times l'_q + 1..i-1] = s'_1[u \times l'_q + 1..i-1]$.
- $s'_1[i] = c$.
- $q[i+1..m - u \times r'_q] = s'_1[i+1..m - u \times r'_q]$.
- $r'_q = 0 \vee s'_r[r'_q] = R[u \times r'_q]$.

Il est clair que la vérification de chaque condition prend un temps de $O(1)$. Pour plus de détail sur ces conditions, voir la partie précédente (type d'erreur insertion).

De même dans le cas de l'insertion, on doit faire la vérification seulement pour le premier élément de la liste. Si on n'a pas une égalité pour le premier élément, on conclut que la liste n'existe pas et on arrête immédiatement. Autrement, on retourne le premier élément et on continue à récupérer les éléments restants de la liste sans les vérifier dans un temps $O(1)$ par élément, et on retourne chaque élément comme une égalité (on fait la substitution dans la position i pour chaque élément).

L'erreur de type suppression : Les mots obtenus en supprimant un caractère de q. Dans ce cas, on essaye d'accéder au dictionnaire exact avec chaque mot candidat possible qui peut être obtenu en supprimant un des caractères de q à la position i. Cela revient à faire une recherche exacte pour chaque mot candidat. On a exactement m mots candidats, qui signifie qu'on doit passer seulement un temps de $O(1)$ pour chaque accès pour vérifier s'il y a une égalité ou non.

Si on souhaite accéder au dictionnaire pour le mot q' obtenu en supprimant le caractère numéro i de q ($q' = q[1..i-1]q[i+1..m]$), on fait les étapes suivantes :

En premier, on vérifie que $L[i-1] \neq \perp$ et $R[m-i] \neq \perp$. Si ce n'est pas le cas, on conclut immédiatement qu'on a pas une égalité pour le mot q'.

Dans le cas contraire, on calcule la valeur de hachage $h(q') = F[i-1] \oplus G[i+1] \otimes At[i-1]$ dans un temps constant. Ensuite, on calcule la valeur $mphf$. En utilisant la position retournée par $mphf$, on fait la requête au dictionnaire exact, qui va retourner le mot s si c'est un mot court, ou s' dans le cas contraire.

Si le mot est un mot court, donc on peut faire la comparaison directement de s et q' dans un temps de $O(1)$ (avant de comparer les mots, on compare leur longueur) et on retourne une solution si les deux mots sont égaux.

Si le mot est long, on divise s' en trois parties égaux s'_1, s'_l et s'_r. On retourne une solution si $s'_1 = q'$. Pour faire cette vérification on met $l'_q = \lfloor (i-1)/u \rfloor$ et $r'_q = \lfloor (m-i)/u \rfloor$, et on vérifie toutes les conditions suivantes :

- La longueur de s' est $3(m-1)$.
- $l'_q = 0 \vee s'_l[l'_q] = L[u \times l'_q]$.
- $q[u \times l'_q + 1..i-1] = s'_1[u \times l'_q + 1..i-1]$.
- $q[i+1..m-u \times r'_q] = s'_1[i..m-1-u \times r'_q]$.
- $r'_q = 0 \vee s'_r[r'_q] = R[u \times r'_q]$.

3.3.2.3 Le comptage du nombre des solutions d'une requête

Le comptage du nombre d'occurrences de la solution peut être fait dans un temps de $O(m)$. Ceci est fait par la somme des mots solutions de chaque type d'erreur. Pour la suppression, on peut avoir au maximum m mots solutions, et la vérification de chaque mot candidat prend un temps de $O(1)$. Pour l'insertion et la substitution on a besoin de $m+1$ et m étapes respectivement. Dans chaque étape on a besoin seulement de vérifier le premier élément de la liste retournée par le dictionnaire des listes de substitution, et si cet élément est valide, on ajoute la taille de la liste (obtenue par l'utilisation de l'opération $size\_of$) au nombre total des mots solutions $occrs$.

3.3.3 La recherche approchée dans un texte indexé avec une distance d'édition 1

La structure de données de la recherche approchée dans le dictionnaire avec une distance d'édition "1" peut-être utilisée pour avoir une solution pour la recherche approchée dans le texte. Pour cela on combine cette solution avec celle décrite dans [102]. Donc on construit l'index de [102].

On sélectionne toutes les sous-chaines de texte de longueur jusqu'à $log(n)loglog(n)$ caractères, et on les stocke dans la structure de données (dictionnaire exact + dictionnaire des listes de substitution) qui va stocker au plus $n(log(n)loglog(n))$ chaines de caractères. Les chaines qui apparaissent plus qu'une seule fois dans le texte sont stockées seulement

une fois dans la structure de données. On associe avec chaque chaine (dans le dictionnaire exact) en tant que données satellites[1] dans le dictionnaire un pointeur qui pointe vers un vecteur qui stocke tous les positions où la chaine apparaît dans le texte.

Chaque pointeur de vecteur occupe $log(n)$ bits, et chaque chaine occupe au maximum $log(n)loglog(n)$ caractères, cela rend l'espace total utilisé par le dictionnaire $O(n(log(n)loglog(n))^2b)$ bits.

On peut stocker tous les vecteurs qui contiennent les positions des chaines dans un seul tableau contigu de $O(nlog(n)loglog(n))$ pointeurs, qui occupe un espace total de $O(nlog(n)^2loglog(n))$ bits. On fait la somme de l'espace utilisé par tous les composants de la structure de données , on obtient un espace total de $O(n(log(n)loglog(n))^2b)$ bits.

Pour faire une requête avec une chaine de caractères q, on vérifie simplement si $|q| > log(n)loglog(n)$ dans ce cas on utilise la structure de données de [102] pour répondre à la requête. Dans le cas contraire, on utilise cette structure de données pour répondre à la requête.

3.4 La méthode de Amir et al.

Amir *et al.* [83] proposent une solution pour résoudre le problème de la recherche approchée avec $k = 1$ erreur dans un dictionnaire et dans un texte avec l'utilisation du Trie et du Trie inversé. Avec une seule erreur on a deux parties exactes dans le mot requête, ces deux parties sont vérifiées avec les deux Tries, ensuite une étape de vérification est faite afin de trouver les solutions approchées. Ils obtiennent un résultat dans un temps de pré-traitement de $O(n\,log^2 n)$ où n est la taille du texte dans le problème de l'indexation, et la taille de dictionnaire dans le problème de la recherche dans un dictionnaire. Pour le temps de la requête, ils obtiennent un temps de recherche de $O(m\,log^2 n\,log\,log\,n + occrs)$ dans l'indexation de texte où m est la longueur de mot requête et $occrs$ est le nombre des occurrences de la solution. Pour la recherche dans un dictionnaire ils obtiennent un temps de $O(m\,log^3 d\,log\,log\,d + occrs)$ avec n est la taille de texte et d est la taille du dictionnaire.

Nous commençons par expliquer la construction de la structure de données bidirectionnelle (Trie et Trie inversé), ensuite, nous donnons l'algorithme de la recherche pour un mot requête de longueur m.

3.4.1 La construction de la structure de données

La construction de la structure de données se fait comme suit :

1. Construire le Trie du dictionnaire et enregistrer les numéros des mots dans les feuilles.

1. Données satellites : des données supplémentaires qui ne sont pas une partie de la structure de données.

2. Construire le Trie inversé du dictionnaire et enregistrer les numéros des mots dans les feuilles.
3. Dans chaque structure de données, relier toutes les feuilles de la gauche vers la droite, pour obtenir une liste de numéros de mots.
4. Enregistrer dans chaque nœud le pointeur de la feuille la plus à gauche et la plus à droite, donc le début et la fin de la liste des sous-listes qui contiennent tous les enfants sortant de ce nœud.

La construction de chaque Trie prend $O(n)$, où n est le nombre total de caractères dans le dictionnaire.

3.4.2 La recherche de mot requête

Soit un mot requête $q = c_1\, c_2 \ldots c_m$ de longueur m.
La étapes pour rechercher toutes les solutions approchées sont comme suit :

Pour $i = 1, ..., m$ **faire :**
i représente la position de l'erreur(ϕ) $q' = c_1\, c_2 \ldots c_{i-1}\, \phi\, c_{i+1}\, c_{i+2} \ldots c_m$.

1. Trouver le nœud v, qui représente l'emplacement de $\{c_1, c_2, ..., c_{i-1}\}$ dans le Trie, si ce nœud existe.
2. Trouver le nœud w, qui représente l'emplacement de $\{c_{i+1}, c_{i+2}, ..., c_m\}$ dans le Trie inversé, si ce nœud existe.
3. Si v et w existent, alors trouver l'intersection entre les deux sous-listes enracinés par les deux nœuds, (l'intersection des numéros qui sont stockés dans les feuilles des deux nœuds v et w).

Comment faire l'intersection : Pour une solution efficace au problème de l'intersection entre les deux ensembles des nombres qui sont dans les feuilles de deux nœuds v_{err} et w, Amir *et al.* utilisent la technique expliqué dans [115].

Chapitre 4

La recherche approchée dans un dictionnaire en utilisant des tables de hachage

4.1 Introduction

Le problème de la recherche approchée dans les dictionnaires est largement étudié. Le problème est défini comme suit : Étant donné (comme entrée donnée à l'avance) un dictionnaire $D = \{x_1, x_2, \dots x_d\}$ avec d mots de longueur totale n sur un alphabet Σ de taille σ [1] et un seuil k. On veut construire une structure de données sur D d'une manière à être en mesure de répondre aux requêtes suivantes : étant donné un mot requête q de longueur m, retourner tous les mots du dictionnaire à une distance d'édition au plus k du mot requête q.

Dans ce chapitre, nous nous intéressons aux algorithmes pratiques pour la recherche approchée des motifs dans un dictionnaire avec un nombre d'erreurs d'édition $k \geq 2$.

Nous avons proposé une méthode qui utilise une structure de données basée sur le hachage avec sondage linéaire et des signatures de hachage associées afin d'optimiser le temps de recherche. Pour le dictionnaire exact, nous utilisons une structure de données basée sur plusieurs tables de hachage selon la longueur des mots.

Dans notre travail nous utilisons l'idée de dictionnaire des listes de substitutions qui a été proposée dans [15]. Pour plus de détail voir dans le chapitre état de l'art, la sous-section 3.3.1.3.

Notre solution pratique utilise un espace mémoire de $O(n \log \sigma)$ bits, et un temps de requête proche de $O(m + occrs)$ et plus exactement en $O(m \left\lceil \frac{m \log \sigma}{w} \right\rceil)$ où w représente la taille d'un mot mémoire, et $occrs$ est le nombre d'occurrences trouvés. Le temps de construction est de $O(n)$. Nous montrons que la performance de notre solution est en $O(m \left\lceil \frac{m \log \sigma}{w} \right\rceil)$ sous des hypothèses qui sont susceptibles d'être vérifiées en pratique. Nous

1. Pour des raisons pratiques, on suppose que l'alphabet est le domaine entier $[1..\sigma]$.

vérifions expérimentalement que notre solution est compétitive avec les solutions précédemment proposées et que nos hypothèses sont réalistes (elles sont généralement vraies en pratique).

Le reste de ce chapitre est organisé comme suit : Dans la section 4.2, nous détaillons les étapes de la création de notre structure de données. Dans la section 4.3, nous expliquons comment faire la recherche exacte et approchée avec k=1 erreur. Dans la section 4.4, nous donnons un récapitulatif des performances de notre structure de données. Dans la section 4.5, nous expliquons comment étendre notre algorithme pour fonctionner avec deux erreurs ou plus. La section 4.6 est dédiée aux tests et aux expérimentations. Dans la section 4.7, nous expliquons comment adapter notre méthode dans l'indexation de texte. La dernière section 6.11 conclut ce chapitre.

4.2 La structure de données

Dans notre travail, nous utilisons seulement deux (2) structures de données afin de minimiser le nombre d'accès mémoire.

1. Un dictionnaire exact basé sur le hachage.
2. Un dictionnaire des listes de substitutions implémenté avec une structure de données basée sur le hachage avec sondage linéaire et des signatures de hachage.

Dans le but d'optimiser l'espace mémoire, nous compressons ces deux structures de données, en utilisant pour cela une autre structure de données (le vecteur des bits). La version non compressée est progressive, donc on peut rajouter des nouveaux mots au dictionnaire (sous réserve de ne pas dépasser la capacité maximale).

Dans notre travail, nous utilisons le hachage avec sondage linéaire [48] pour implémenter les deux dictionnaires (index) exact et approché. La raison pour laquelle nous utilisons ce type de hachage est qu'il a été démontré qu'il est parmi les méthodes de hachage les plus pratiques. La bonne performance est essentiellement due a la bonne localité des accès mémoire [116].

Tous nos tableaux de hachage avec sondage linéaire sont paramétrés avec un facteur de chargement (LF : LoadFactor) $\alpha < 1$. Pour plus de simplicité, nous utilisons le même paramètre pour tous les tableaux de hachage.

Dans notre travail, nous utilisons une fonction de hachage polynomiale, la fonction de hachage de Rabin-Karp [90]) modulo un nombre premier P. Les valeurs de hachage sont calculées modulo $P = 2^{32} - 5$, le plus grand nombre premier qui est plus petit que 2^{32}. La fonction de hachage est très simple, on utilise un nombre t choisi aléatoirement dans

l'intervalle $[1..P-1]$. La valeur de hachage pour une chaîne de caractère x est calculée par la formule suivante : $h(x) = \sum_{i=1}^{m} x_i \cdot t^i$ [1].

La fonction de hachage polynomiale a comme propriété intéressante d'être incrémentale. On peut dans un temps $O(m)$ pré-traiter la chaîne x, en calculant un certain nombre de vecteurs d'entiers en temps $O(m)$, de telle sorte que le calcul de la valeur de hachage de n'importe quelle chaîne à une distance 1 de x, prend un temps constant (en utilisant un nombre constant d'additions et de multiplications modulo le nombre premier P). Voir les détails dans l'état de l'art dans la sous-section 3.3.2, et dans ce chapitre section 4.3.

Notre algorithme permet de traiter deux erreurs d'édition ou plus ($k \geq 2$). Dans les prochaines sous-sections, on donne plus de détails sur notre structure de données.

4.2.1 Dictionnaire exact

Le dictionnaire exact (basé sur le hachage avec sondage linéaire) est utilisé pour localiser les occurrences exactes et permet de vérifier si le motif candidat est dans le dictionnaire ou pas.

Dans notre travail, nous utilisons une seule structure de données pour la vérification exacte des mots, cette structure de données étant constituée de plusieurs tables de hachage selon les longueurs des mots du dictionnaire.

L'insertion d'un mot de longueur m dans le dictionnaire exact prend un temps moyen de $O(m)$, et la recherche prend aussi un temps moyen de $O(m)$ si on applique la méthode naïve. Avec l'utilisation du *bit-parallelism*, la vérification peut être améliorée en un temps de $O(\lceil \frac{m \log \sigma}{w} \rceil)$.

Nous notons que le nombre t de longueurs de mots distinctes (les mots avec une longueur $l_1, l_2, \ldots l_t$) dans le dictionnaire ne peut excéder $2\sqrt{n}$. Ceci peut être aisément prouvé comme suit. En premier lieu considérons les longueurs de mots excédant $\sqrt{n}+1$. On note que le nombre total des mots de longueur au moins égales à $\sqrt{n}$ ne peut dépasser $\sqrt{n}$, sinon, leur longueur totales serait au moins $\sqrt{n} \cdot (\sqrt{n}+1) > n$. Donc le nombre de longueurs distinctes supérieur à $\sqrt{n}$ ne peut dépasser $\sqrt{n}$, puisque qu'il ne peut y a voir plus de mots (de longueurs au moins $\sqrt{n}$) que de longueurs distinctes. Comme le nombre de longueurs distinctes au plus $\sqrt{n}$ est au maximum égal à $\sqrt{n}$, nous déduisons que le nombre maximal de longueurs distinctes est au plus égal à $2\sqrt{n}$.

Afin d'implémenter le dictionnaire exact, on pourrait partitionner les mots du dictionnaire en (au plus) $2\sqrt{n}$ groupes, contenant chacun des mots de même longueur, et on utilise ensuite une table de hachage séparé pour chaque groupe.

1. Il y a une petite erreur dans notre article [16], nous avons mis m à la place de i et donc la formule suivante $h(x) = \sum_{i=1}^{m} x_m \cdot r^m$ est fausse.

Le détail de notre schéma : Pour mettre en place le dictionnaire exact, au lieu de stocker $2\sqrt{n}$ [1] différentes tables de hachage pour les $2\sqrt{n}$ groupes des longueurs de mots distincts, nous utilisons un seuil β et on stocke $\beta - 1$ tables de hachages.

Tous les mots de longueur $i < \beta$ sont stockés dans la table de hachage numéro $i - 2$ [2] qui contient $\lceil n_i/\alpha \rceil$ cases, où chaque case est de longueur i caractères.

Le mot de longueur i est stocké dans le table numéro $i - 2$ car on considère que le mot le plus petit est de longueur 2. La table numéro 0 va stocker les mots de longueur 2 ($i = 2$,donc $i - 2 = 2 - 2 = 0$), et les mots de longueur 3 vont être stockés dans la table numéro 1, jusqu'à ce qu'on arrive à l'avant-dernière table dont le numéro est $(\beta - 3)$. Tous les mots de longueur $i \geq \beta$ sont stockés dans une table de hachage (la dernière table dont le numéro est $(\beta - 2)$) qui stocke des pointeurs vers les mots originaux au lieu des mots eux-mêmes.

Dans notre implémentation, nous avons utilisé $\beta = 16$ caractères. Lorsque nous avons fait des calculs sur l'optimisation de l'espace mémoire, nous avons trouvé qu'il est préférable de stocker tous les mots de longueur $i \geq 16$ dans un seul tableau de pointeur. Pour plus de détails et d'explications, voir la sous-section 4.2.1.1.

Tous les $\beta - 1$ tableaux sont gérés par le hachage avec sondage linéaire. Chaque tableau $i - 2$ à une taille multiple de la longueur i des mots qu'il va stocker. Chaque élément de la table est un bloc de longueur i permettant de stocker les mots de longueur i. Le numéro de bloc pour stocker un mot donné est calculé par une fonction de hachage. La taille de tableau T en terme d'octets est donc $(\texttt{NbMots\_long\_i/LF}) \times i$. On arrondit ensuite le résultat de la division à l'entier immédiatement supérieur. Donc la formule finale est : $T = \left\lceil \frac{\texttt{NbMots\_long\_i}}{\texttt{LF}} \right\rceil \times i$. La taille de la table en terme de blocs est $\texttt{Tnb} = \left\lceil \frac{\texttt{NbMots\_long\_i}}{\texttt{LF}} \right\rceil$.

La dernière table (celle contenant les pointeurs vers des mots de longueur $i \geq 16$), est constituées de cases de taille w, et sa taille est $T = \left\lceil \frac{\texttt{Tab\_NbMots}[\beta-2]}{\texttt{LF}} \right\rceil$.

La structure de données est illustrée dans la figure 4.1.

La construction de la structure de données est donnée en résumé dans l'algorithme suivant :

Algorithme de construction de la structure de données exacte :
Entrée : fichier qui contenant les mots de dictionnaire.
Sortie : structures de données composant le dictionnaire exact.

---

1. Dans notre article [16] il y a une petite erreur dans la section 3.7. Nous avons écrit $\sqrt{d}$ (d le nombre total des mots) au lieu de $2\sqrt{n}$ (n la somme de toutes les longueurs des d mots de dictionnaire). De même pour la section 3.1, on doit mettre $2\sqrt{n}$ au lieu de $\sqrt{n}$.
2. Il y a une petite erreur d'indice dans notre article [16], dans la section 3.7, nous avons écrit la table numéro i au lieu du numéro $i - 2$.

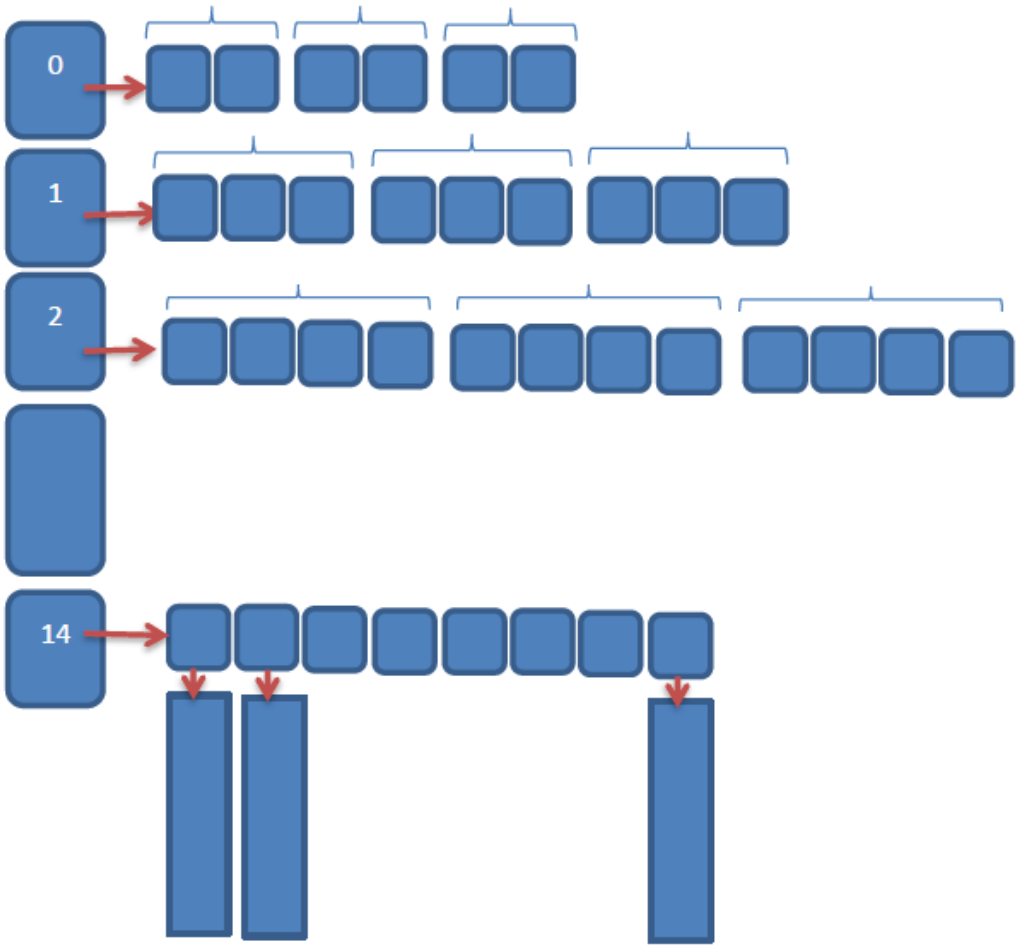

FIGURE 4.1 – La structure de données du dictionnaire exact. Cette structure comporte plusieurs tableaux pour stocker les mots selon leurs longueurs, le mot d'une longueur m va être stocké sans son marqueur de fin de chaine dans le tableau $m-2$. Le dernier tableau stocke les mots de longueur ≥ 16 avec leurs marque de fin. Tous tableaux sont gérés par le hachage avec sondage linéaire.

1. Mettre le nombre des mots de longueur i dans la case $i-2$ du tableau `Tab_NbMots`, et le nombre des mots de longueur $i \geq \beta$ dans la case $\beta - 2$.
2. Créer un tableau de pointeurs `Tab_exact` de $\beta - 2$ cases.
3. Pour chaque case i ($i \in [2..\beta-3]$) de `Tab_exact`, créer un tableau de caractères de taille $T = \lceil \frac{\texttt{Tab\_NbMots}[i-2]}{\texttt{LF}} \rceil \times i$.
4. Créer un tableau de pointeurs de taille $T = \lceil \frac{\texttt{Tab\_NbMots}[\beta-2]}{\texttt{LF}} \rceil$, et le relier avec la dernière case $\texttt{Tab\_exact}[\beta - 2]$.
5. Pour chaque mot x de longueur i dans le dictionnaire :
   - Calculer la valeur de hachage de x donc $h(x)$.
   - Si $i < \beta$ insérer x dans le tableau $\texttt{Tab\_exact}[i-2]$, à la position $\texttt{pos} = h(x) \bmod (\lceil \frac{\texttt{Tab\_NbMots}[i-2]}{\texttt{LF}} \rceil)$ sans son marqueur de fin ($\backslash 0$).
   - Si $i \geq \beta$, stocker le mot x dans la mémoire avec son marqueur de fin, et garder

son adresse dans le tableau pointé par `Tab_exact`$[\beta-2]$ à la position `pos` $= h(x) \bmod (\lceil \frac{\texttt{Tab\_NbMots}[\beta-2]}{\texttt{LF}} \rceil)$.

4.2.1.1 Explication du paramètre $\beta = 16$

Le choix du seuil β utilisé pour guider le choix du stockage des mots du dictionnaire (soit dans des tables de hachage stockant des caractères ou alors dans une table stockant des pointeurs vers les mots dans la mémoire) dépend de l'espace mémoire occupé par chaque structure. Le but étant de faire une comparaison afin de choisir le paramètres β qui minimise l'utilisation de l'espace mémoire.

Dans une table de caractères, les mots sont stockés à l'intérieur de la table sans le marqueur de fin (\0). Par contre dans la table des pointeurs, les mots sont stockés dans la mémoire (avec leur marqueur de fin) et on garde leurs adresses dans la table des pointeurs.

Afin d'éviter toute confusion, on appelle *longueur de mot*, le nombre des caractères du mot sans compter le marqueur de fin du mot.

Dans une table de hachage stockant des caractères, chaque mot x de longueur m occupe un espace mémoire de $\lceil m/\texttt{LF} \rceil$. Lorsqu'on utilise une table de pointeurs, chaque mot x occupe un espace de taille $m+1$ (la longueur du mot plus le marqueur de fin) auquel on ajoute la taille de pointeur w (un mot mémoire) divisé par le taux de chargement :$(m+1) + \lceil w/\texttt{LF} \rceil$. Dans notre cas, on prend $\texttt{LF} = 0,7$.

En utilisant ces deux informations, on peut trouver quelles structures utiliser, selon l'espace mémoire occupée pour le stockage des mots du dictionnaire.

On nomme la première structure (Table de caractères) **Tab_char**, et la deuxième structure (table de pointeurs vers les mots) **Tab_pointeur**.

Les calculs pour un seul mot montre que la structure **Tab_char** occupe moins d'espace mémoire pour les mots de longueurs 2 à 15. Lorsque la longueur des mots est supérieure à 16, la structure **Tab_pointeur** donne de meilleurs résultats.

On va faire la même vérification pour les deux structures en utilisant des groupes de mots de 10, 100, et 1000 mots pour toutes les longueurs, voir la figure 4.2.

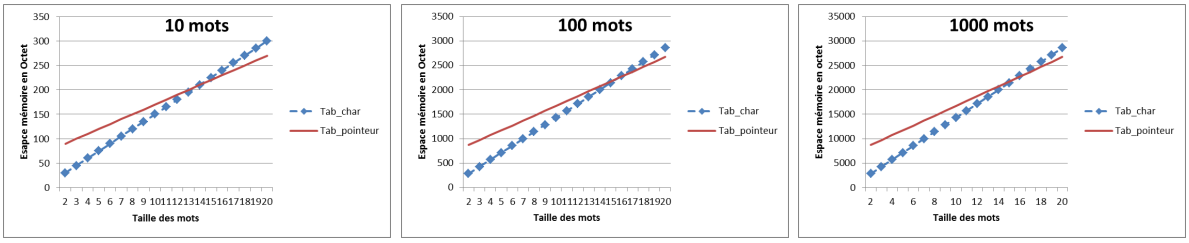

FIGURE 4.2 – Comparaison de l'occupation de l'espace mémoire par les deux structures de données avec différents nombres de mots.

On remarque dans la figure 4.2, qu'avec juste 10 mots, la structure `Tab_pointeur` commence à donner des meilleurs résultats lorsque la longueur des mots devient 15, mais il y a pas une grand différence avec la longueur 16. Lorsque le nombre des mots est 100, la structure `Tab_pointeur` donne de meilleurs résultats lorsque la longueur des mots devient 16, et c'est la même chose avec le nombre des mots égale à 1000. Il est clair que la structure `Tab_char` occupe moins d'espace mémoire pour les mots de longueur 2 à 15.

On fixe donc le seuil β à 16 car la structure avec pointeurs `Tab_pointeur` donne de meilleurs résultats à partir de ce seuil.

En fin de compte, pour les mots de longueur 2 à 15 on utilise des tables de caractères, et pour les mots de longueur ≥ 16, on stocke des pointeurs vers les mots stockés dans la mémoire.

4.2.2 Dictionnaire des listes de substitutions

Pour pouvoir faire une recherche approchée, nous avons besoin d'ajouter une autre structure de données nommée *dictionnaire des listes de substitutions*.

L'idée est de faire un pré-traitement, pour stocker pour chaque position donnée, la liste des caractères dont la substitution à cette position conduit à un mot existant dans le dictionnaire.

Donc, au lieu de tester pour chaque position tous les σ caractères possibles, la structure nous donne une liste de caractères à insérer ou à substituer dans une position donnée du mot requête q. Cette liste contient généralement un seul caractère si on a juste une seule solution (ce qui est le cas pour la majorité des requêtes).

Pour implémenter le dictionnaire des listes de substitutions nous utilisons une table de hachage avec sondage linéaire, dans laquelle chaque élément est un caractère. le calcul des valeurs de hachage se base sur les mots du dictionnaire dans lesquels on substitue un caractère spécial dans la position de l'erreur. Pour chaque mot x_i de longueur m, et pour chaque position $j \in [1..m]$, on insère le caractère $x_i[j]$ dans la table de hachage à la position $i = h(x_{i,j}) \bmod T$, où $x_{i,j} = x_i[1..j-1]\phi x_i[j+1..m]$ et T est la taille de la table de hachage.

Tous les caractères qui appartiennent à la même liste de substitutions sont mappés vers la même position dans la table de hachage. En d'autres termes, deux caractères c_1 et c_2 seront mappés vers une même position $i = h(p\phi v)$ dans la table de hachage si et seulement si il existe une paire de chaînes de caractères (p, v), tels que les deux mots (pc_1v) et (pc_2v) existent dans le dictionnaire.

Diminuer les candidats avec des signatures de r bits : Nous proposons une deuxième méthode d'implémentation du dictionnaire de listes de substitutions dans le but de diminuer le nombre des collisions.

On ajoute une signature de hachage de taille r bits avec chaque caractère, pour permettre ainsi de filtrer une grande partie des caractères qui sont en collision. La signature de hachage est calculée sur la base de la valeur de hachage de la chaîne de caractère $x_i[1..j-1]\phi x_i[j+1..m]$, et sa longueur est seulement r bits (pour un petit nombre r).

Quand un caractère c est inséré dans la liste de substitutions, on stocke avec lui sa signature correspondante. Lors de la recherche, on considère qu'un caractère appartient à la liste de substitutions si et seulement si sa signature est la même que celle stockée dans la liste de substitutions.

Détail de la mise en place de la structure de données : Comme il a été expliqué précédemment, nous avons deux variantes pour l'implémentation de la table de hachage des caractères de substitution : celle dans laquelle chaque entrée est juste un caractère, et l'autre dans laquelle chaque entrée est un caractère et une signature de r bits.

On code un caractère en utilisant un octet ($8bits$), et on code la signature de hachage r en utilisant 4 bits ($r = 4$). Le codage de la signature avec 4 bits simplifie l'implémentation.

Dans la deuxième variante, la table de hachage est divisé en blocs de 3 octets. Le premier octet stocke le premier caractère, le troisième octet stocke le deuxième caractère, et le deuxième octet central stocke les 4 bits de chaque caractère. Les 4 bits de poids fort (celle à gauche) de l'octet central pour le premier caractère, et les 4 bits de poids faible (celle à droite) pour le deuxième caractère.

La table de hachage est constituée avec des blocs de 3 octets (soit 2 cases, deux caractères avec leurs signatures). Voir la figure 4.3.

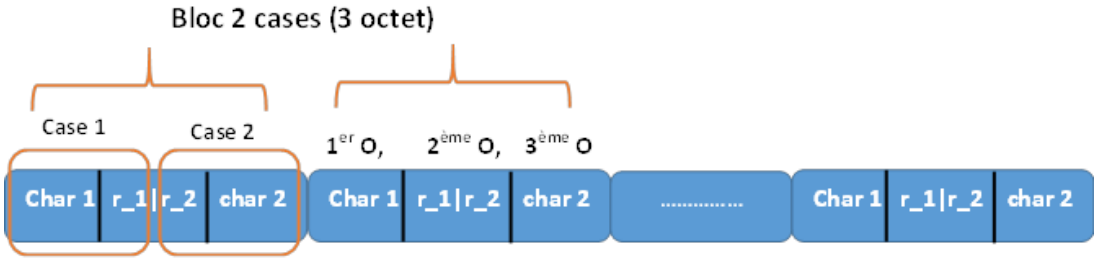

FIGURE 4.3 – La deuxième variante du dictionnaire des listes de substitutions.

Si le nombre des caractères est impair, donc il reste un caractère seul, on lui donne aussi un bloc de 3 octets. Au final, le tableau à une taille T qui est multiple de 3. La taille de la table est donc $T = \left\lceil \frac{n+\frac{n}{2}}{\text{LF}} \right\rceil$, si $T \bmod 3 = 2$ alors T devient $T+1$, et si $T \bmod 3 = 1$ alors T devient $T+2$.

La signature de hachage de $r = 4$ bits est constituée à partir de la valeur de hachage $h(x)$. Dans notre cas, on prend simplement les 4 bits de poids faible de la valeur de hachage $h(x)$. Nous avons fait ce choix, car, quelle que soit la taille de la valeur de

hachage (petite ou grande), généralement les bits de poids faible ne sont pas nuls, contrairement aux bits de poids fort, si la valeur de hachage est petite cela signifie qu'ils sont nuls.

Le calcul de la valeur de hachage pour les mots candidats générés depuis le mot x_i se fait en $O(1)$. Pour cela, on fait le même pré-traitement utilisé par [15]. On prépare trois tableaux :

1. $A_t[0, m+1]$ pour stocker toutes les puissances de t jusqu'à t^{m+1} où m est la longueur de mot x_i, et t est une valeur aléatoire choisi entre 1 et P, comme il est expliqué dans le début de cette section.
2. $F[0..m]$ un tableau qui stocke les valeurs de hachage pour les préfixes de mot x_i.
3. $G[1..m+1]$ un tableau qui stocke les valeurs de hachage pour les suffixes de mot x_i.

Pour plus de détail sur ces trois tableaux, voir dans le chapitre état de l'art la sous-section 3.3.2.

Pour le mot x_i de longueur m, pour chaque $j \in [1..m]$, on substitue le caractère $x_i[j]$ avec un caractère spécial ϕ où $x_{i,j} = x_i[1..j-1]\phi x_i[j+1..m]$. La valeur de hachage de mot candidat $x_{i,j}$ est donnée en $O(1)$ par $h(x_{i,j}) = F[j-1] \oplus (\phi \oplus G[j+1]) \otimes A_t[j]$. Cette valeur va permettre de stocker le caractère $x_i[j]$ dans la structure de données.

Nous donnons l'algorithme qui résume la construction de notre structure de donnée **le dictionnaire des listes de substitutions**.

Algorithme de construction du dictionnaire des listes de substitutions (DLS) :
Entrée : un fichier qui contient les mots du dictionnaire.
Sortie : la structure de données (DLS) pour la recherche approchée.

1. Calculer le nombre total des caractères des mots du dictionnaire n.
2. Créer une table de hachage avec sondage linéaire :
   - La première variante : la taille de la table est simplement $T = \lceil \frac{n}{\text{LF}} \rceil$.
   - La deuxième variante : la taille de la table est $T = \lceil \frac{n+\frac{n}{2}}{\text{LF}} \rceil$, si $T \bmod 3 = 2$ alors T devient $T+1$, et si $T \bmod 3 = 1$ alors T devient $T+2$.
3. Préparer le tableau $A_t[0, m'+1]$, pour stocker toutes les puissances de t jusqu'à $t^{m'+1}$ (m' le plus long mot du dictionnaire).
4. Pour chaque mot x_i de longueur m ($i \in [1..d]$ et d le nombre des mots du dictionnaire) : préparer les deux tableaux $F[0..m], G[1..m+1]$ qui stockent les valeurs de hachage pour les préfixes et les suffixes de mot x_i, ensuite pour chaque position $j \in [1..m]$:

(a) Substituer le caractère $x_i[j]$ par un caractère spécial : $x_{i,j} = x_i[1..j-1]\phi x_i[j+1..m]$.

(b) Calculer la valeur de hachage de $x_{i,j}$ en $O(1)$ par la formule $h(x_{i,j}) = F[j-1] \oplus (\phi \oplus G[j+1]) \otimes A_t[j]$.

(c) Trouver la position dans le tableau $\texttt{pos} = h(x_{i,j}) \bmod T$.

(d) Stocker le caractère $x_i[j]$

La première variante :

- Si la case numéro pos est vide, alors insérer $x_i[j]$.
- Sinon, avancer dans la table case par case jusqu'à ce qu'on trouve une case vide, $\texttt{pos} = \texttt{pos} + 1 \bmod T$.

La deuxième variante :

- Si $\texttt{pos} \bmod 3 = 0$, alors stocker le caractère dans cette position, et stocker les 4 bits de la signature dans $\texttt{pos} + 1$ dans la partie poids fort (partie gauche).
- Si $\texttt{pos} \bmod 3 = 2$, alors stocker le caractère dans cette position, et stocker les 4 bits dans l'octet précédent dans les 4 bits de poids faible qui est à droite dans $\texttt{pos} - 1$.
- Si $\texttt{pos} \bmod 3 = 1$, on passe à l'octet suivant ($\texttt{pos} = \texttt{pos} + 1$), et on applique le cas de $\texttt{pos} \bmod 3 = 2$.
- Si la case est non vide, avancer vers la case suivante, si on est dans $\texttt{pos} \bmod 3 = 0$, alors $\texttt{pos} = \texttt{pos} + 2$, et si $\texttt{pos} \bmod 3 = 2$, alors $\texttt{pos} = \texttt{pos} + 1 \bmod T$.

4.2.3 La compression de notre structure de données

Dans notre travail, toutes les tables ont le même facteur de chargement $\alpha = 0,7$, ce qui signifie que pour chaque table (16 en tout) on a 30% de cases vides. Le but est de trouver une méthode permettant de réduire l'espace mémoire utilisé par notre structure de données tout en gardant le même temps d'accès au données. L'inconvénient du compactage de la structure de données est qu'elle devient non incrémentale.

Nous utilisons la compression basée sur un tableau de bits. Le résultat du compactage donne deux vecteurs : un table de hachage compact de n cases, et un vecteur de bits de $n' = \lceil \frac{n}{\alpha} \rceil$ bits.

La compression est faite de la manière suivante : on parcourt toutes les cases de la table de hachage, et on génère le vecteur de bits. On met dans le vecteur de bits un 0 pour chaque case vide de la table de hachage originale, et un 1 pour chaque case non-vide. Durant le parcours, on rajoute toutes les cases non-vides dans la table de hachage compact (qui était initialement vide).

Afin de prendre en charge les requêtes dans la table de hachage compactée, on indexe le vecteur de bits pour supporter les opérations de rangs [42, 44] (Le rang d'une position i dans un vecteur de bits b est défini comme étant le nombre de 1 dans $b[1..i]$).

L'idée est très simple, une position dans le tableau de bits est la même que celle dans la table de hachage originale. On marque les cases non-vides avec 1, donc la position de ce même élément dans la table compactée est le nombre des 1 dans le tableau de bits depuis le premier bit jusqu'à cette position i.

Compter le nombre de 1 depuis le début jusqu'à une position i donnée à chaque fois augmente le temps de la requête. Afin d'y remédier, nous utilisons une méthode qui permet de donner le nombre des 1 pour n'importe quelle position dans un temps constant. On implémente un simple algorithme de rang qui augmente l'espace original du vecteur de bits d'un facteur de $1 + 1/\delta$, tel que δ est une petite valeur constante. On décompose le vecteur de bits en blocs de δ mots mémoires ($w \times \delta$ bits), et on sauvegarde le rang calculé jusqu'au début de chaque bloc. L'espace final obtenu est de $N(1 + 1/\delta)$ bits pour un vecteur de bits originalement constitué de N bits.

Dans notre travail, nous assumons un mot mémoire w de 32 bits, et nous prenons $\delta = 4$. Chaque bloc a une taille de 4 mots mémoire (32×4 bits), auquel, on ajoute une case mémoire de taille w (si on pose $N = 4$, et on applique la formule $N(1 + 1/\delta)$, alors $4(1 + 1/4) = 4 + 4/4 = 4 + 1$). Donc pour chaque bloc de 4 mots mémoire, on ajoute un mot mémoire pour stocker le nombre des 1 (voir la figure 4.4). La requête de rang sur le vecteur de bits retourne la position dans la table de hachage compacte. On compte le nombre des 1 consécutifs dans le vecteur de bits pour avoir le nombre des cases non-vides, et donc la position dans la table de hachage compacte. Le nombre de comptage partiel i va stocker tous les nombres des 1 dans les $(i - 1)\delta$ premiers mots. Au lieu de stocker les comptes partiels dans un vecteur distinct, on les stocke entrelacées avec le vecteur de bits.

Afin de supporter le comptage efficace des 1 jusqu'à la position i, on utilise le vecteur de bits pour obtenir le comptage des 1 jusqu'au mot numéro $\lfloor \frac{i}{\delta} \rfloor$ en utilisant les comptes partiels, puis compter le nombre des 1 jusqu'à $32\delta - 1$ bits, qui couvrent au plus δ mots en utilisant l'opération `PopCount` [117] [1], voir la figure 4.4. Ainsi, l'espace total sera de $n' + n'/\delta = \lceil n/\alpha \rceil (1 + \delta)$ bits, et le temps sera de $O(\delta) = O(1)$.

On note que toutes nos méthodes non-compactes sont entièrement incrémentales, dans le sens où l'ajout d'un élément au dictionnaire est possible et prend à peu près le même temps que le temps de construction total divisé par le nombre total des mots. En revanche, les structures compactes ne sont pas incrémentales, en raison des étapes de constructions et du résultat final du compactage.

Notre implémentation suppose des entiers et des pointeurs de 32 bits, mais il est trivial

1. Même si le temps d'exécution n'est pas constant dans la théorie, dans la pratique, elle est très rapide et peut être considérée comme constante. Notons également que les processeurs récents possèdent souvent une implémentation matérielles de cette instruction.

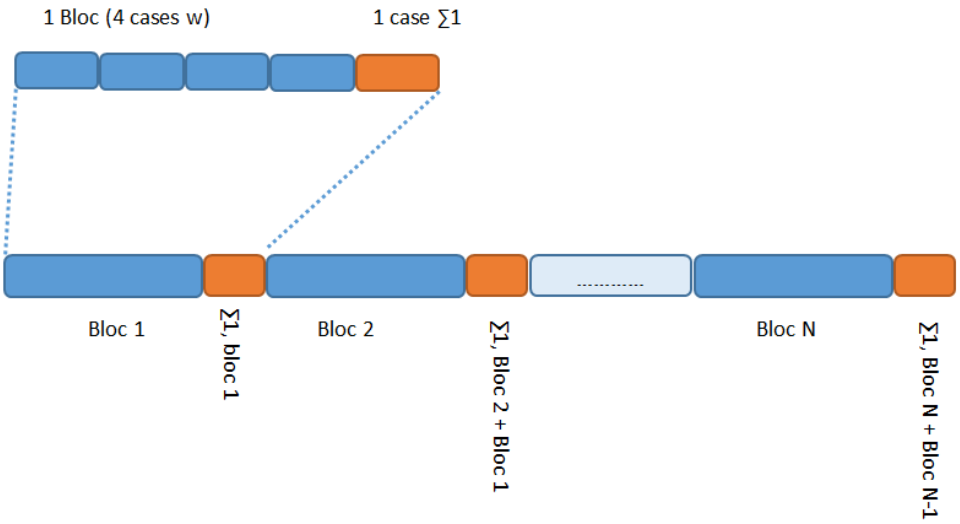

FIGURE 4.4 – Le vecteur de bits et les cases qui contiennent les compteurs des nombres des 1 pour tous les blocs précédent.

de changer le code pour fonctionner avec 64 bits (et donc gérer de plus grands dictionnaires) sans encourir beaucoup de dépassement d'espace mémoire ou de temps. Le seul élément qui utilise plus d'espace mémoire est le dictionnaire exact pour les grands mots qui sont stockés dans une table de pointeurs (les mots de longueur ≥ 16). Tous les autres composants n'utilisent pas les pointeurs, et donc leur utilisation de l'espace ne devraient pas être affectés.

Remarque: Pour plus de performance avec 64 bits, et pour optimiser l'espace mémoire, le seuil β va être égale à 32. On utilise les mêmes étapes expliquées dans la sous-section 4.2.1.1 pour le trouver. Cela implique le changement de nombre de tableaux utilisé dans le dictionnaire exact, on va utiliser 30 tableaux de caractères afin d'optimiser l'espace mémoire pour les mots qui ont une longueur ≥ 16 (la nouvelle condition est $i \geq 32$).

4.3 La vérification des occurrences

Soit un mot requête q de longueur m. Pour vérifier les occurrences potentielles, nous utilisons une stratégie similaire à Boytsov [89]. Au lieu de calculer la distance d'édition entre le mot requête et le mot candidat, nous construisons directement une chaîne modifiée, en appliquant l'opération d'édition candidate, ensuite, nous faisons une comparaison directe entre la chaîne obtenue et le mot candidat.

Avant de commencer les vérifications des mots candidats, on prépare trois tableaux A_t, G, F comme décrit dans la partie **dictionnaire des listes de substitutions** dans la sous-section 4.2.2 :

– $A_t[0..m'+1]$, pour stocker toutes les puissances de t jusqu'à $t^{m'+1}$, m' est la longueur de mot le plus long supposé être utilisé comme un mot requête. On utilise un nombre

suffisamment grand dans le but de calculer ce tableau une seule fois, et on le réutilise pour tous les mots requête, car ces valeurs sont indépendantes des mots requête.

- $F[0..m]$, un tableau qui stocke les valeurs de hachage pour les préfixes du mot requête q.
- $G[1..m+1]$ pour stocker les valeurs de hachage pour les suffixes de q. Ces deux tableaux sont calculés pour chaque mot requête en $O(m)$.

On commence par expliquer comment vérifier l'existence d'un mot requête dans le dictionnaire exact (une recherche exacte). Par la suite, on expliquera les étapes de la recherche approchée avec une seule erreur pour les trois types d'erreurs d'édition.

4.3.1 La recherche exacte

La recherche exacte est très simple, il suffit juste de trouver la position dans la structure de données à l'aide de la fonction de hachage et ensuite de faire une simple comparaison. Voici les étapes à faire pour une vérification dans le dictionnaire exact :

Algorithme de la recherche exacte :
Entrée : un mot requête q de longueur m, le dictionnaire exact (la structure de données exacte).
Sortie : booléen (existe ou non).

1. Calculer la valeur de hachage du mot requête q en $O(1)$, la valeur de hachage est déjà pré-calculée $h(q) = F[m] = G[1]$.
2. Calculer la position avec la valeur de hachage modulo la taille de la table qui stocke les mots de longueur m, $\texttt{pos} = h(q) \bmod T$.
   - Si $m < \beta$, alors on utilise la taille de la table en nombre de blocs (le tableau $Tab\_exact[m-2]$), $T = \lceil(\texttt{Tab\_NbMots}[m-2]/\texttt{LF})\rceil$.
   - Si $m \geq \beta$, on utilise le tableau de pointeur $Tab\_exact[\beta-2]$. La taille est en nombre de cases mémoire de taille w, donc $T = \lceil(\texttt{Tab\_NbMots}[\beta-2]/\texttt{LF})\rceil$.
3. Faire une comparaison directe entre le mot requête q et le mot stocké. Si la case à la position `pos` est vide, alors, on dit qu'on n'a pas de résultat. Dans le cas contraire, on teste tous les mots jusqu'à la première case vide.

4.3.2 La recherche approchée

Dans la recherche approchée, on construit des mots candidats en appliquant l'une des trois opérations de distance d'édition dans une position donnée dans un temps constant, ensuite, on fait une vérification dans le dictionnaire exact. On récupère les caractères depuis le **dictionnaire des listes de substitutions**, qui donnent des mots candidats qui peuvent être une solution, afin de les utiliser dans la substitution et l'insertion. Pour la suppression, il suffit juste de vérifier l'existence de mot candidat dans le dictionnaire exact.

4.3.2.1 La substitution

On commence par la création d'un mot copie $\texttt{sub}_q$ de notre mot requête q de la même longueur m. Ceci permettra de générer des mots candidats en $O(1)$, et de substituer les caractères récupérés depuis le dictionnaire des listes de substitutions dans une position donnée i.

Algorithme de la recherche approchée (substitution)
Entrée : un mot requête q de longueur m, le dictionnaire exact et le dictionnaire des listes de substitutions.
Sortie : liste des mots solutions approchées.

1. Pour chaque position i dans $\texttt{sub}_q$ (le mot copie du mot requête) avec $i \in [1..m]$:
2. Calculer la valeur de hachage pour la position i en $O(1)$, avec la formule suivante : $h(\texttt{sub}_q) = F[i-1] \oplus (\phi \oplus G[i+1]) \otimes A_t[i]$.
3. Récupérer une liste de caractère `list_chars` depuis le dictionnaire des listes de substitutions :
   (a) Trouver la position dans la table de hachage qui stocke les caractères de substitutions $\texttt{pos} = h(\texttt{sub}_q i) \bmod T$.
   (b) La première variante de la structure de données des listes de substitutions :
   - La taille de la table $T = \lceil \frac{n}{\texttt{LF}} \rceil$.
   - Si la case `pos` est vide, alors on retourne une liste vide.
   - Sinon, on récupère tous les caractères de cette position jusqu'à ce qu'on arrive à une case vide. L'avancement se fait case par case $\texttt{pos} = \texttt{pos} + 1 \bmod T$.

   (c) La deuxième variante (avec la signature de 4 bits) :
   - La taille de tableau $T = \lceil \frac{n+\frac{n}{2}}{\texttt{LF}} \rceil$, si $T \bmod 3 = 2$ alors T devient $T+1$, et si $T \bmod 3 = 1$ alors T devient $T+2$.
   - Si la case est vide, alors retourner une liste `list_chars` vide.

- Sinon, pour tous les caractères de cette position jusqu'à une case vide, vérifier si la signature de $(h(\texttt{sub}_q))$ est la même que celle stockée dans la table de hachage. Si c'est le cas, alors ajouter le caractère à la liste `list_chars`. Si `pos` mod $3 = 0$, alors les 4 bits de la signature sont dans `pos` $+ 1$ dans la partie poids fort (partie gauche). Si `pos` mod $3 = 2$, alors les 4 bits sont dans l'octet précédent `pos` $- 1$ dans les 4 bits de poids faible (à droite). Si `pos` mod $3 = 1$ alors on passe à l'octet suivant (`pos` $+ 1$).
- Pour avancer vers les cases suivantes, si `pos` mod $3 = 0$, alors on fait `pos` $=$ `pos` $+ 2$, et si on est dans le cas `pos` mod $3 = 2$, alors on fait (`pos` $=$ `pos` $+ 1$ mod T).

4. Pour chaque caractère j dans la liste `list_chars` de taille L ($j \in [1..L]$) :
   - Substituer le caractère `list_chars`$[j]$ dans la position i du mot requête $\texttt{sub}_q[i] = list\_chars[j]$, pour construire le mot candidat $\texttt{sub}_q = \texttt{sub}_q[1..i-1]\texttt{list\_chars}[j]\texttt{sub}_q[i+1..m]$.
   - Vérifier l'existence du mot candidat dans le dictionnaire exact. La valeur de hachage se calcule en $O(1)$ avec la formule $h(\texttt{sub}_q) = F[i-1] \oplus (\texttt{list\_chars}[j] \oplus G[i+1]) \otimes A_t[i]$.
5. Passer à la position suivante dans le mot requête $(i+1)$. Avant cela, on doit remettre le caractère originel de la position i en faisant $\texttt{sub}_q[i] = q[i]$.

On remarque que la génération des mots candidats après la récupération de la liste des caractères prend un temps constant par candidat : il suffit juste de substituer les caractères de la liste dans la position i et à la fin remettre le caractère original avant de passer à la position suivante.

Dans les deux variantes de la structure de données, l'obtention de la liste de substitutions pour une position donnée, peut nécessiter la traversée de nombreuses cases dans la table de hachage avant d'atteindre une case vide. Pour borner le temps de la requête dans le pire cas, nous avons implémenté la stratégie suivante : à chaque fois qu'une requête pour une liste de substitutions traverse plus que σ cases dans la table de hachage, on arrête le parcours des cases de la table de hachage, et on crée σ chaînes candidates parmi tous les Σ caractères possibles [1].

4.3.2.2 L'insertion

Dans cette partie, le traitement est le même qu'auparavant avec juste quelques petits changements dans les indices. Nous en décrivons les détails dans ce qui suit.

1. Il faut noter que lorsque qu'on traverse σ cases, cela ne veut pas dire qu'on va collecter σ caractères candidats distincts. Le nombre peut même être beaucoup plus petit lorsque les signatures de hachage sont utilisées

On commence par créer un mot copie $\texttt{ins}_q$ avec une longueur $m+1$ afin de nous permettre de générer des mots candidats en $O(1)$ avec insertion dans une position donnée i. On copie le mot q dans $\texttt{ins}_q$ à partir de la deuxième position, on laisse la première position vide, donc $\texttt{ins}_q[2..m+1] = q[1..m]$.

Algorithme de la recherche approchée (insertion)
Entrée : un mot requête q de longueur m, le dictionnaire exact, le dictionnaire des listes de substitutions.
Sortie : liste des mots solutions approchées.

1. Pour chaque position dans le mot copie du mot requête $\texttt{sub}_q$ $i \in [1..m+1]$:
2. Calculer la valeur de hachage du mot $\texttt{sub}_q$ après une insertion du caractère spécial ϕ à la position i, $h(\texttt{ins}_q) = F[i] \oplus (\phi \oplus G[i+1]) \otimes A_t[i+1]$.
3. Calculer la position $\texttt{pos} = h(\texttt{ins}_q i) \bmod T$, et ensuite, récupérer une liste de caractères (`list_chars`) si elle existe.
4. Construire les mots candidats par la substitution des caractères de la liste `list_chars` à la position i (où on a inséré le caractère ϕ) : $\texttt{ins}_q = \texttt{ins}_q[1..i]\texttt{list\_chars}[j]\texttt{ins}_q[i+1..m+1]$. Ensuite, faire la vérification dans le dictionnaire exact, la valeur de hachage étant calculée par la formule $h(\texttt{ins}_q) = F[i] \oplus (\texttt{list\_chars}[j] \oplus G[i+1]) \otimes A_t[i+1])$.
5. Passer à la position d'erreur suivante $(i+1)$. Décaler le caractère qui est à la position $i+1$ vers la position i du mot $\texttt{ins}_q$ $(\texttt{ins}_q[i] = \texttt{ins}_q[i+1])$.

4.3.2.3 La suppression

Dans la suppression, nous n'avons pas une liste de caractères à insérer ou à substituer. Il suffit juste de faire une suppression dans la position i et de calculer la valeur de hachage pour ensuite faire une vérification dans le dictionnaire exact.

Algorithme de recherche approchée (suppression)
Entrée : un mot requête q de longueur m, le dictionnaire exact, le dictionnaire des listes de substitutions.
Sortie : liste des mots solutions approchées.

1. On crée un mot $\texttt{del}_q$ de longueur $m-1$.
2. Construire les m mots candidats :

i. Construire le premier mot candidat de $i = 1$. On copie le mot q à partir de sa deuxième position, donc $\texttt{del}_q[1..m-1] = q[2..m]$.

ii. Construire les mots candidats pour les autres positions $i \in [2..m]$. Il suffit juste de faire $\texttt{del}_q[i-1] = q[i]$.

iii. Construire le dernier mot candidat, on ajoute $\texttt{del}_q[i-1] = q[i-1]$.

3. Pour chaque mot candidat $\texttt{del}_q = q[1..i-1]q[i+1..m]$ ($i \in [1..m]$), calculer la valeur de hachage avec la formule suivante : $h(\texttt{del}_q) = F[i-1] \oplus G[i+1] \otimes A_t[i-1]$, ensuite, faire la vérification dans le dictionnaire exact.

4.3.3 La recherche dans la structure compacte

Avec la structure compacte, on utilise exactement les mêmes étapes précédentes, sauf qu'au lieu d'accéder directement au tableau de hachage compact, on passe par le tableau de vecteur de bits pour trouver la position de l'élément dans le tableau compact.

On commence par trouver la position dans le tableau de bits $\texttt{pos}_{\texttt{bv}} = h(q') \bmod T$. Cette position $\texttt{pos}_{\texttt{bv}}$ va nous donner la position réel $\texttt{pos}_{\texttt{htc}}$ dans la table de hachage compacte, pour cela on récupère le nombre de 1 depuis le début du tableau de bits jusqu'à cette position $\left(\texttt{pos}_{\texttt{htc}} = \sum_{i=0}^{i=\texttt{pos}_{\texttt{bv}}} 1\right)$. Comme on a expliqué dans la section 4.2.3, cette opération ce fait dans un temps constant, à cause des cases mémoire ajoutées dans chaque δ cases qui contient la somme des 1.

On récupère directement le nombre d'éléments de la liste de substitutions depuis le vecteur de bits qui est le nombre de 1 successifs depuis la position trouvée dans le vecteur de bits. Pour cela, on avance case par case depuis la position $\texttt{pos}_{\texttt{bv}}$ pour vérifier si on est arrivé à une case vide, et on calcule le nombre d'éléments de 1.

4.4 Récapitulatif des performances de notre structure de données

Nous pouvons maintenant définir deux théorèmes pour récapituler les performances de notre structure de données en se basant sur deux hypothèses formalisées avec les définitions suivantes :

Définition 1 *Une fonction de hachage pour une chaîne de caractères est dite incrémentale, si après quelques pré-traitements faits sur une chaîne de caractères x en entrée, le résultat du hachage de n'importe quelle chaîne de caractères à une distance 1 de x prend un temps constant.*

Définition 2 *Une fonction de hachage mappant les éléments de l'ensemble U vers l'ensemble V est dite complètement aléatoire, si elle se comporte comme si elle a été choisie aléatoirement parmi l'ensemble de toutes les fonctions possibles de U dans V.*

L'espace mémoire utilisé lors de la construction et l'espace mémoire final (espace utilisé par notre structure de données) sont résumés par le théorème suivant :

Théorème 1 *Le dictionnaire présenté dans cette section occupe $O(n \log \sigma)$ bits (où n est la taille totale des chaînes dans le dictionnaire). L'espace utilisé durant la construction est également de $O(n \log \sigma)$ bits. Si le compactage de la table de hachage est utilisé, le dictionnaire final occupe au plus un espace mémoire de $n(2 \log \sigma + O(1))$ bits.*

Preuve 1 *Les mots du dictionnaire donnés comme entrée occupent un espace de $n \log \sigma$ bits. Le dictionnaire des listes de substitutions a $n/\alpha = O(n)$ cases de taille $\log \sigma + r$ bits chacune dont n seulement sont occupées. Le dictionnaire exact a (d) cases (rappelons que d est le nombre total des mots dont la taille totale est n) qui occupent un espace total de $(n/\alpha) \log \sigma = O(n \log \sigma)$ bits. L'espace mémoire total utilisé par l'algorithme de construction est clairement en $O(n \log \sigma)$ bits, car il lit uniquement les entrées, et écrit les sorties qui occupent un espace total de $O(n \log \sigma)$ bits. Quand le compactage est utilisé, l'espace du dictionnaire exact est réduit à $n \log \sigma + d(1 + 1/\alpha) + o(d) = n \log \sigma + O(d)$ bits, et l'espace utilisé par le dictionnaire des listes de substitutions est réduit à $n(1 + r + \log \sigma) + o(n) = n \log \sigma + O(n)$ bits, pour un espace total de $n(2 \log \sigma + O(1))$ bits.*

La performance de la construction de la structure de données et le temps de requête sont résumés dans le théorème suivant :

Théorème 2 *En supposant un hachage complètement aléatoire et incrémental, et que chaque liste de substitutions contient un seul élément, et étant donné le mot requête de longueur m, alors le temps de requête moyen est égal à $O(m \lceil \frac{m \log \sigma}{w} \rceil)$. Et étant donné un dictionnaire avec une longueur des mots totale égale à n, alors le temps de la construction de la structure de données est $O(n)$ en moyenne.*

Preuve 2 *En supposant un hachage incrémental, et étant donné un mot de longueur m. Le calcul de la valeur de hachage pour chercher une liste de caractères (dans le dictionnaire des listes de substitutions) pour insertion à une position i donnée dans le mot requête prend un temps amorti de $O(1)$. Cela donne comme résultat, un temps total de $O(m)$ pour les $O(m)$ positions du mot requête. L'hypothèse qui dit que chaque liste de substitutions est de taille 1, signifie que tous les n mots ($x_{i,j} = x_i[1..j-1]\phi x_i[j+1..m_i]$ avec $i \in [1..n]$) utilisés comme clés pour l'insertion des caractères dans le dictionnaire des listes de substitutions vont être tous distincts. De plus, il est bien connu que le coût d'insertion d'un élément dans une table de hachage avec sondage linéaire est de $O(1)$ en moyenne*

si on suppose l'utilisation d'un hachage aléatoire [48]. Donc il est évident que l'insertion de chaque élément de la liste de substitutions prend un temps constant en moyenne. Cela donne un temps total moyen de $O(n)$ pour insérer tous les n éléments (caractères) dans le dictionnaire des listes de substitutions pour tous les d mots du dictionnaire. Par conséquent, le temps total moyen pour la construction de la version non-compactée du dictionnaire des listes de substitutions est de $O(n)$ [1]*. Avec des arguments similaires, le temps de construction du dictionnaire exact est estimé à $O(n)$, car l'insertion d'un mot de longueur m requiert un nombre constant d'accès au tableau de hachage, et chaque accès à un coût de $O(m)$. En faisant la somme de tous les temps d'insertion pour tous les d mots du dictionnaire, le temps total moyen d'insertion devient $O(n)$. La génération de la version compacte du dictionnaire des listes de substitutions depuis le dictionnaire non-compact prend trivialement $O(n)$. En fin de compte, le temps de construction de la version compacte et non-compacte du dictionnaire présentées dans ce travail est de $O(n)$ en moyenne.*

Maintenant, on va prouver que le temps de recherche moyen pour un mot requête de longueur m est $O(m \lceil \frac{m \log \sigma}{w} \rceil)$. Rappelons que l'extraction des éléments d'une liste de substitutions requiert l'accès à des cases consécutives dans la table de hachage, en démarrant par la position i jusqu'à arriver à une case vide $i + k$ et en collectant ensuite tous les k caractères des cases parcourues. On distingue deux cas :

1. *Le cas où la liste de substitutions n'existe pas, la position initiale i va suivre une distribution uniforme sur toutes les cases de la table de hachage. Alors le nombre total des cases parcourues va suivre la borne d'une recherche infructueuse dans une table de hachage avec sondage linéaire, une borne connue comme étant constante en moyenne selon [48].*
2. *Dans le cas d'une requête pour une liste de substitutions existante, le nombre de cases parcourues est également constant en moyenne. Le nombre de cases parcourues dans ce cas donnés par le temps moyen d'une recherche fructueuse dans une table de hachage, qui est connu également comme étant constant [48].*

Considérons maintenant que la table de hachage est décomposée en un ensemble de cases non-vides continues, et un ensemble de cases vides. De fait que les cases vides sont une fraction constante de toutes les cases de tableau, on conclut que choisir une position de départ d'une manière uniforme et aléatoire à partir des cases non-vides va générer un nombre constant de cases parcourues avant d'atteindre une case vide. Comme chaque

1. Pour voir pourquoi l'hypothèse que chaque liste de substitutions de taille 1 est utile pour assurer un temps d'insertion moyen constant pour les éléments de la liste de substitutions, on peut considérer la liste de substitutions de taille maximale de σ. Vu que tous les éléments de cette liste vont être mappés au même endroit dans la table de hachage, l'insertion de chaque élément doit parcourir tous les éléments qui sont déjà insérés dans la même liste de substitutions avant d'atteindre une case vide, résultant en un temps d'insertion $\Omega(\sigma)$.

position non-vide représente une liste de substitutions, on conclut que le nombre moyen de caractères candidats collectés lors de la recherche d'une liste de substitutions existante est aussi constant. Cela implique que le nombre total des mots candidats générés depuis la recherche de toutes les listes de substitutions est égal à $O(m)$ en moyenne. Le temps de vérification dans le dictionnaire exact pour un mot candidat est égal à $O(\lceil \frac{m \log \sigma}{w} \rceil)$, qui est le temps nécessaire pour comparer le mot candidat avec le mot du dictionnaire.

L'explication est que le dictionnaire exact est représenté en utilisant la table de hachage avec sondage linéaire, dans laquelle une requête parcourt un nombre de cases constant en moyenne. À chaque fois, nous avons besoin d'un temps de $O(\lceil \frac{m \log \sigma}{w} \rceil)$ pour comparer le mot du dictionnaire avec le mot candidat. Ceci prouve que le temps d'exécution final d'une requête est de $O(m \lceil \frac{m \log \sigma}{w} \rceil)$ en moyenne.

Dans la pratique, nous nous attendons à ce que la plupart des listes de substitutions contiennent un seul élément (nous l'avons vérifié précisément sur les jeux de données testés). Comme il est fait dans la majorité des travaux pratiques, nous avons préféré d'utiliser les fonctions de hachage les plus rapides qui sont efficaces en pratique au lieu d'utiliser celles qui sont moins rapides avec de meilleures garanties théoriques [1].

4.5 Extension à deux erreurs ou plus

Notre algorithme peut être étendu pour fonctionner avec deux erreurs ou plus. Pour deux erreurs, on stocke deux **dictionnaire de listes de substitutions**. Le premier dictionnaire est utilisé pour une erreur (le niveau 1), pour chaque mot x dans le dictionnaire de longueur m, et pour toutes les positions $i \in [1..m]$, on place un caractère spécial (joker) ϕ dans la position i, donc $(x' = u\phi v)$. On calcule la valeur de hachage du mot x' donc $(h(x'))$, et ensuite, on stocke le caractère $x[i]$ dans le dictionnaire des listes de substitutions. Au total, on a m caractères à insérer. Ce dictionnaire est donc identique à celui utilisé plus haut pour le cas d'une seule erreur.

Le deuxième dictionnaire (le niveau 2) va stocker pour chaque mot et chaque paire de positions distinctes, le caractère de la première position. Pour chaque mot x de longueur m, et toute paire de positons (i, j) tels que $i \in [1..m-1]$ et $j \in [i+1..m-1]$, on place deux caractères spéciaux ϕ dans les positions i et j, donc $x'' = u\phi v\phi w$ (u, v, w sont des sous-chaines), et on calcule la valeur de hachage de $h(x'')$ afin de stocker le caractère original de la position i dans ce deuxième dictionnaire.

1. Le lecteur peut se référer à [118] pour avoir plus de détails sur les raisons pour lesquelles les fonctions de hachage simples se comportent dans la pratique suivant le principe de l'aléatoire pure (elles se comportent en pratique comme si elles étaient entièrement aléatoires).

Au moment de la requête, on interroge d'abord le dictionnaire des listes de substitutions de niveau 2 avec le mot requête $q'' = u\phi v\phi w$ pour récupérer la liste des caractères de substitutions $L_2 = \{c_1, c_2, ..., c_l\}$, grâce à la valeur de hachage $h(q'')$, qui nous donne la position dans la table de hachage de niveau 2. Ensuite, pour chaque caractère c_i récupéré de cette liste, on le substitue dans le premier joker du mot q'' (donc $q' = uc_iv\phi w$). Ensuite, on passe au dictionnaire des listes de substitutions de niveau 1 pour trouver tous les caractères à substituer au deuxième joker à l'aide de la valeur de hachage $h(q')$. On obtient alors une liste $L_1 = \{e_1, e_2, ..., e_f\}$, et on substitue alors les caractères de la liste (tous les éléments e_j) dans la deuxième position, pour constituer les mots candidats de la forme $q = uc_ive_jw$. Enfin, on interroge le dictionnaire exact pour les mots résultants (avec la substitution des 2 caractères trouvés dans le premier et deuxième dictionnaires de substitutions).

Exemple :
Soit le mot `ALABAMA`. Pour illustrer cet exemple, on considère juste une seule combinaison de 2 positions (les autres combinaisons peuvent être traitées de la même façon). Dans la phase de construction de la structure de données, on fait ce qui suit :

1. On stocke le mot `ALABAMA` dans le dictionnaire exact.
2. On stocke une liste de substitutions associée à `ALABA`ϕ`A` dans le dictionnaire des listes de substitutions de niveau 1.
3. On stocke une liste de substitutions associée à `A`ϕ`ABA`ϕ`A` dans le dictionnaire de niveau 2.

Au moment de la requête, on fait les étapes suivantes :

1. On interroge le deuxième dictionnaire des listes de substitutions avec `A`ϕ`ABA`ϕ`A`, pour obtenir un ensemble de caractères ($L_2 = \{c_1, c_2, ..., c_l\}$) qui pourraient être substitués au premier joker. Dans notre exemple, nous avons un seul élément dans la liste $L_2 : \{c = L\}$.
2. Pour chaque caractère c_i dans cet ensemble, on interroge le niveau 1 avec `A` c_i `ABA`ϕ`A`, pour obtenir la liste de substitutions de la deuxième position. Nous avons un seul élément dans la première liste L, et donc on va interroger le dictionnaire des listes de substitutions avec `ALABA`ϕ`A`. On trouve un seul élément aussi, le caractère M.
3. On fait la substitution pour le deuxième joker, et donc on trouve le mot `ALABAMA`, on vérifie l'existence dans le dictionnaire exact. Le mot est trouvé dans le dictionnaire et est donc ajouté à la liste des solutions approchée avec 2 erreurs pour tous les mots de la forme `A`ϕ`ABA`ϕ`A`.

Concernant l'utilisation de l'espace mémoire de ce schéma étendu à 2 erreurs, on note que le mot `ALABAMA` étant de longueur $\ell = 6$, va générer $\frac{\ell(\ell-1)}{2} = \frac{6\times5}{2} = 15$ caractères pour

les listes de substitutions.

Dans le cas général, on peut donner les théorèmes suivants qui résument la performance de notre structure de données pour 2 erreurs. On commence d'abord par l'utilisation de l'espace mémoire.

Théorème 3 *Le dictionnaire présenté dans cette section occupe* $O(N \log \sigma)$ *bits où :* $N = \sum_{i=1}^{d} n_i(n_i - 1)/2$ *et où* n_i *est la longueur du mot numéro i du dictionnaire. L'utilisation de l'espace mémoire lors de la construction est de* $O(N \log \sigma)$ *bits. Si le compactage est utilisé, alors le dictionnaire (l'index) final occupe un espace mémoire au plus de* $(N + 2n)(\log \sigma + O(1))$ *bits.*

Preuve 3 *On peut utiliser les mêmes arguments que pour le théorème 1. La seule différence étant que l'on a maintenant le dictionnaire des listes de substitutions de niveau* 2 *dans lequel on insère* N *éléments. La borne spatiale se calcule facilement.*

Ensuite, analysant le temps de la requête on obtient le théorème suivant :

Théorème 4 *En supposant un hachage totalement aléatoire et incrémental, et que chaque liste de substitutions contient un seul élément, avec un mot requête donné de longueur* m*, alors le temps de requête moyen est de* $O(m^2 \times \lceil \frac{m \log \sigma}{w} \rceil)$.

Étant donné un dictionnaire de d *mots, tels que* $N = \sum_{i=1}^{d} n_i(n_i - 1)/2$ *où* n_i *est la longueur du mot numéro i du dictionnaire, le temps de construction de la structure de données est de* $O(N)$ *en moyenne.*

Preuve 4 *De même que pour la borne spatiale, la borne temporelle peut être également prouvée en utilisant les mêmes arguments utilisés pour prouver le théorème 2. Le coût des requêtes et des insertions dans le dictionnaire des listes de substitutions de niveau* 2 *peut être délimité de la même manière que celui du dictionnaire de niveau* 1*, sauf qu'on remplace le terme* m *par* m^2 *pour les requêtes, car chaque requête fait* $O(m^2)$ *accès en plus au dictionnaire de niveau* 2*. On remplace également le terme* n *par* N *dans le temps de la construction, puisqu'on insère* N *éléments dans le dictionnaire des listes de substitutions de niveau* 2*.*

Nous avons seulement implémenté l'algorithme pour deux erreurs, mais notre dictionnaire (la structure de données) peut être étendu pour gérer $K > 2$ erreurs. Pour cela, on utilise des dictionnaires de listes de substitutions jusqu'à K niveaux. Le nombre de caractères spéciaux substitués dans chaque mot est égal à k, et le caractère stocké dans le niveau K est le premier caractère (parmi les K caractères à substituer par ϕ). Pour chaque niveau, on doit faire toutes les combinaisons possibles pour remplacer les k caractères par ϕ. Un mot de longueur n_i, ajoute $\binom{n_i}{k} \times \log \sigma$ bits au niveau k pour le dictionnaire $(1 \leq k \leq K)$.

4.6 Expérimentation

Notre implémentation est modulaire. Nous avons fait notre expérimentation avec les deux jeux de données suivants : l'ensemble des titres de Wikipédia **WikiTitle**, qui a environ 1,8 millions de mots, et le dictionnaire Anglais qui comporte environ 213 milles mots [1]. Ces deux ensembles de données sont également utilisés dans les expérimentations de Karch et al [88]. Ils ont également utilisé deux autres jeux de données plus petits : Mobydick et Town.

Nous avons expérimenté avec **Mobydick**, mais étant donné la petite taille de ce jeu de données (37 mille mots), le résultat n'a pas donné un bon aperçu des performances (la structure de données totale tient dans le cache du processeur). Nous n'avons pas fait d'expérimentations sur **Town** car il n'était pas disponible et de toute façon, le résultat n'aurait pas été très différent de **Mobydick** étant donné que le fichier ne contient que 47 mille mots.

Nous avons implémenté la structure de données, les algorithmes de construction et de recherche en **langage C** utilisant le compilateur GNU GCC version 4.4.1. Les tests ont été effectués avec un processeur Intel E8400 3.0 Gigahertz Core 2 duo, exécutant Windows 7. Nous avons utilisé un seul cœur de processeur.

Nous avons comparé nos résultats avec ceux de Karch et al. Les raisons pour lesquelles nous avons choisi Karch et al. est que ces derniers ont trouvé que leurs résultats étaient meilleurs que ceux des approches concurrentes. Comme l'implémentation de Karch et al. n'est pas accessible au public, nous avons juste pris les résultats tels que publiés dans l'article de Karch et al. Nous notons que le matériel utilisé par Karch et al. est comparable au nôtre, bien qu'ils ne soient pas identiques [2]. Ainsi, la comparaison des temps de requête n'est pas tout à fait exacte. Toutefois, étant donné que la capacité des deux machines est très proche, nous pouvons considérer la comparaison précise jusqu'à une marge d'erreur entre 10% et 20%. D'autre part, l'utilisation de l'espace mémoire est directement comparable, car elle ne dépend pas du matériel utilisé.

Nous n'avons pas comparé avec les résultats de Boytsov. Bien que le code source soit disponible, nous n'avons pas réussi à le faire fonctionner. Nous aurions pu comparer aux résultats de Boytsov [89] tels qu'indiqués dans son article, mais il n'y a pas de tableaux indiquant la performance des algorithmes dans son papier. La seule alternative aurait été de deviner les résultats des performances depuis les figures, mais cela aurait donné des résultats trop approximatifs. Néanmoins, d'après les similitudes avec notre méthode, nous nous attendons à ce que la méthode de Boytsov ait les mêmes bornes en termes d'espace mémoire et de temps de requête. Cependant, sa méthode est inférieure à la nôtre dans les aspects suivants :

1. Nous remercions Dennis Luxen pour nous avoir fournit les jeux de données.
2. Leurs expérimentations sont effectuées sur un Intel Xeon X5550 CPU avec 2.67 GHZ, qui est une machine avec une performance très proche de la nôtre.

Méthode	Temps Constr. (secondes)	Espace (Mo)	Temps de requête (μs)
Karch et al	3.45	8.55	8
Non comp. (LF 0.7)	0.4	5.72	27.6
Non comp. + Sign. (LF 0.7)	0.44	7.15	6.24
Compacte (LF 0.7)	0.43	4.53	31.6
Compacte (LF 0.3)	0.47	5.19	5.154
Comp. + Sign. (LF 0.7)	0.48	5.53	8.9
Comp. + Sign. (LF 0.3)	0.625	6.77	4.55

TABLE 4.1 – Comparaison des méthodes existantes sur le dictionnaire Anglais (En) pour une erreur.

1. La structure de données est très lente à construire [89], beaucoup plus lente que les autres compétiteurs étudiés dans son article. Notre structure de donnés est plus rapide que celle de Karch et al. qui est la plus compétitive. Le temps de construction de sa structure peut aller jusqu'à 1 heure pour les grands jeux de données. En revanche, notre temps de construction est de moins d'une minute pour tous les ensembles de données étudiés.
2. La construction de notre structure de données nécessite $O(n \log \sigma)$ bits contre $O(n \log n)$ bits pour l'implémentation de Boytsov. Ceci est très important pour les périphériques mobiles et embarqués où l'espace mémoire est un problème.
3. L'utilisation du hachage parfait minimal conduit à utiliser un espace mémoire élevé au moment de la construction, même si l'espace final est petit. La méthode de hachage parfait minimal exige habituellement au moins 12 octets par entrée, même si l'espace final est inférieur à 3 bits. En revanche, l'espace de construction de notre variante non-compacte est exactement le même que l'espace final.
4. La méthode n'est pas incrémentale. L'insertion de nouveaux mots nécessite la reconstruction de toute la structure de données.

Pour réaliser nos tests, nous avons choisi au hasard 1000 mots de notre dictionnaire et nous avons appliqué une opération d'édition choisie aléatoirement sur chacun d'eux, et ensuite, interrogé le dictionnaire. Pour obtenir le temps de requête final, nous divisons le temps par 1000. Nous avons répété la procédure 20 fois, et nous avons pris la moyenne des résultats obtenus.

Nous avons expérimenté avec différents facteurs de chargement (le paramètre LF ou α) allant de $0,3$ jusqu'à $0,7$. Les résultats sont résumés dans les tableaux 4.1 , 4.2 , 4.3 , 4.4. Les résultats pour les versions compactées sur le dictionnaire anglais sont présentés dans la figure 4.5. Pour Karch et al. juste un seul point est montré (dans le graphe d'une erreur) parce que le schéma pour une erreur n'admet qu'une seule variante (il ne permet pas de faire varier le temps de réponse en fonction du changement de la taille du dictionnaire).

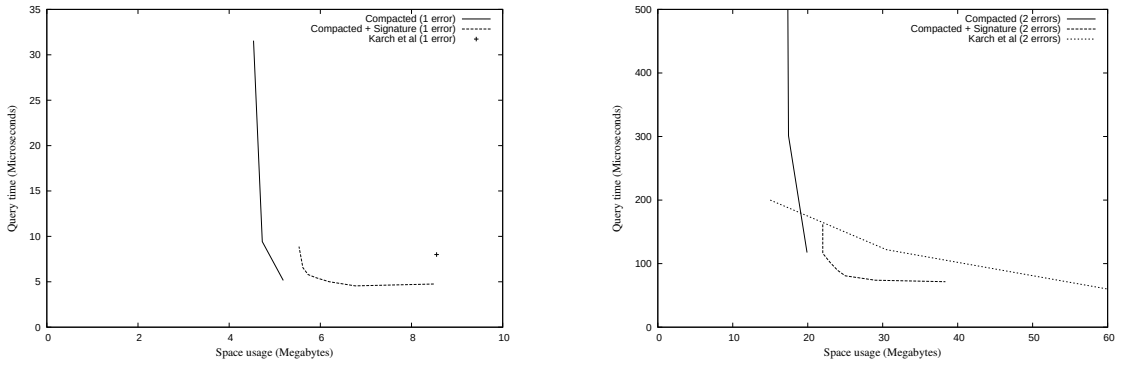

FIGURE 4.5 – L'utilisation de l'espace mémoire en fonction du temps de requête sur le dictionnaire Anglais (En) avec une (1) et deux (2) erreurs (versions compactes).

Méthode	Temps Constr. (secondes)	Espace (Mo)	Temps de requête (μs)
Karch et al	32.29	55.84	34
Non comp. + Sign. (LF 0.7)	4.61	54.44	8.44
Compacte (LF 0.5)	4.72	36.27	16.6
Comp. + Sign. (LF 0.7)	4.91	42.3	13.5
Comp. + Sign. (LF 0.3)	5.17	47.5	8.28

TABLE 4.2 – Comparaison des méthodes existantes sur le dictionnaire WikiTitle (Wi) pour une erreur.

Méthode	Temps Constr. (seconds)	Espace (Mo)	Temps de requête (μs)
Karch et al	12.596	30.49	122
Non comp. + Sign. (LF 0.7)	2.612	27.87	117
Compacte (LF 0.7)	2.54	16.6	2309
Comp. + Sign. (LF 0.7)	2.802	22.8	162
Comp. + Sign. (LF 0.4)	2.95	24.44	89.27
Comp. + Sign. (LF 0.2)	3.1	29.08	73.89

TABLE 4.3 – Comparaison des méthodes existantes sur le dictionnaire Anglais (En) pour deux erreurs.

Méthode	Temps Constr. (seconds)	Espace (Mo)	Temps de requête (μs)
Karch et al	107.289	170.79	502
Non comp. + Sign. (LF 0.7)	21.29	203.82	503
Compacte (LF 0.5)	21.36	127.87	1993
Comp. + Sign. (0.7)	22.86	162.57	1271

TABLE 4.4 – Comparaison des méthodes existantes sur le dictionnaire WikiTitle (Wi) pour deux erreur.

Facteur de chargement (LF)	Temps d'insertion moyen en μs	
	1 erreur	2 erreurs
0.7-0.75	2.42	13.34
0.75-0.80	2.44	13.83
0.80-0.85	2.52	16.66
0.85-0.90	3.52	30.11
0.90-0.95	5.56	43.74

TABLE 4.5 – Le temps d'insertion dans l'index avec le jeu de données WikiTitle pour 1 et 2 erreurs.

D'après les résultats, nous pouvons faire les observations suivantes :

1. Tous nos algorithmes sont beaucoup plus rapides dans la construction que l'algorithme de référence. Ils sont généralement entre 5 et 10 fois plus rapide.
2. L'algorithme pour une erreur est très robuste. Il est 4 fois plus rapide que le résultat de Karch et al. sur les très grands jeux de données (les titres de Wikipédia "WikiTitle").
3. Le compactage donne un compromis très intéressant entre l'espace mémoire et le temps.
4. Le temps de requête de notre méthode pour deux erreurs n'est pas si bon, surtout sur les très grands jeux de données. Toutefois, il fournit toujours un compromis pertinent espace/temps. Même avec le plus grand jeu de données, nous pouvons obtenir un avantage très significatif en terme d'espace mémoire au détriment d'un temps de requête plus important.

Les variantes non-compactes de notre index supportent naturellement l'insertion de nouveaux mots. Nous avons fait quelques expérimentations pour appuyer notre affirmation selon laquelle notre index prend en charge efficacement l'insertion de nouveaux éléments. Pour cela, nous avons construit d'abord notre index sur une fraction de $0,7$ des mots dans le jeu de donnée WikiTitle avec un facteur de chargement de $0,7$. Nous avons ensuite inséré une fraction de $0,25$ de l'ensemble de données original, résultant en un facteur de chargement final de $0,95$.

Nous avons mesuré le temps d'insertion [1] pour des fractions de $0,05$ successives du fichier original. Les résultats sont présentés dans le tableau 4.5. Comme nous pouvons le voir dans le tableau, l'insertion ne prend que quelques microsecondes pour le dictionnaire pour une erreur, et de quelques dizaines de microsecondes dans le dictionnaire pour deux d'erreurs.

Nous avons vérifié d'une façon expérimentale l'hypothèse que la plupart des listes de substitutions (pour une erreur) contiennent juste un seul élément. À partir du tableau 4.6,

1. Nous avons fait des expérimentations sur une machine différente avec des performances légèrement plus faible (Intel Xeon avec 2.27 GHz, 2 Gigaoctets de RAM, sous Windows XP).

nous pouvons voir que plus de 96% des caractères sont dans des listes de substitutions de taille 1.

Taille de liste	Pourcentage Anglais (En)	Pourcentage WikiTitle (Wi)
1	98.89%	96.16%
2	0.71%	1.63%
3	0.19%	0.52%
4	0.09%	0.29%
5	0.05%	0.19%
≥ 6	0.08%	1.20%

TABLE 4.6 – Pourcentage des éléments dans les listes de substitutions pour une taille donnée pour les deux jeux de données.

4.7 L'application de notre méthode dans l'indexation du texte

Notre solution est adaptable à la recherche des occurrences des mots dans un texte sans perte de performances. Rappelons qu'un dictionnaire $D = \{w_1, w_2, ..., w_d\}$ avec d mots, et $w_i \neq w_j$ $(i \neq j)$. Un texte 'T' peut être vu comme étant un ensemble de mots, où chaque mot peut avoir plusieurs occurrences, $T = \{w_1, w_2, ..., w_n\}$, et il se peut que $w_i = w_j$ $(i \neq j)$. L'index de Texte 'T' est un ensemble de mots où un mot peut avoir plusieurs occurrences (des positions). On peut écrire la formule 'T' comme suit : $T' = \{l_1, l_2, ..., l_d\}$ où chaque mot peut avoir plusieurs occurrences $l_i = \{pos_1, pos_2, ..., pos_{nb}\}$.

Afin d'adapter notre solution pour la recherche des mots dans un texte, il suffit d'ajouter dans la structure de données de dictionnaire exact, pour chaque mot un champ qui pointe sur une liste finie de ses positions.

La recherche des facteurs dans le texte :
Pour la recherche de n'importe quel facteur dans le texte, on peut remplacer notre dictionnaire exact avec un arbre des suffixes pour la vérification des mots candidats, le dictionnaires des listes de substitutions des deux niveaux restent les mêmes. On insère le caractère de substitution seulement s'il n'existe pas dans le dictionnaire des listes de substitutions.

4.8 Conclusion

Dans ce chapitre, nous avons présenté un algorithme robuste et efficace pour la recherche approchée pour $k \geq 2$ dans un dictionnaire en utilisant les tables de hachages.

Les algorithmes que nous avons proposés sont faciles à implémenter et très efficaces dans la pratique.

L'algorithme pour une erreur utilise grossièrement deux fois la taille du dictionnaire exact. Le temps de la requête est attendu comme étant linéaire, en se basant sur certaines hypothèses qui sont presque toujours rencontrées en pratique. L'algorithme pour deux erreurs à un espace d'utilisation linéaire, et en pratique, il est plus lent que le premier. Toutefois, Il atteint un compromis pertinent entre temps/espace.

Les versions non-compactes de nos structures sont incrémentales (avec un temps d'insertion, au pire quelques microsecondes).

Dans le chapitre suivant, on va explorer une autre méthode pour résoudre le problème de la recherche approchée. L'idée se base sur le concept de filtrage. Le but est de trouver les parties exactes rapidement de la requête en utilisant le Trie, et de localiser la position d'erreur afin de trouver les solutions possibles rapidement.

Chapitre 5

Un algorithme rapide de recherche approchée de motifs dans un dictionnaire basé sur le Trie et le Trie inversé.

5.1 Introduction

Dans ce chapitre, nous proposons une solution basée sur le concept de filtrage [3, 59]. L'algorithme parcourt tous les mots du dictionnaire afin de localiser rapidement les candidats possibles qui peuvent être des solutions.

Chaque mot approché contient quelques sous-chaînes exactes. Nous cherchons ces pièces exactes en utilisant une structure de données bidirectionnelle (un Trie et un Trie inversé). Nous utilisons une méthode de vérification pour obtenir les mots candidats et retourner toutes les occurrences qui sont au plus différentes du mot requête par une seule erreur.

Ce chapitre est organisé comme suit : Dans la section 5.2, nous détaillons les étapes de notre nouvelle approche (de la recherche approchée) qui réduit le nombre de transitions sortantes testées dans chaque nœud dans le Trie. Dans la section 5.3, nous proposons une méthode qui utilise le Trie et le Trie inversé, et qui améliore la méthode de Amir et al [83], grâce à l'utilisation d'heuristique. Dans la section 5.4, nous expliquons notre troisième méthode, qui est une combinaison des deux méthodes précédentes. La section 5.5 porte sur les tests et les expérimentations. Dans la section 5.6, nous expliquons comment adapter notre méthode dans l'indexation de texte. Dans la dernière section 6.11 nous donnons une conclusion de ce chapitre.

5.2 Une nouvelle approche pour réduire le nombre de transitions sortantes testées dans chaque nœud

5.2.1 L'idée de base

Soit $D = \{x_1, x_2, ..., x_d\}$ un dictionnaire défini sur l'alphabet Σ, et soit P un mot requête de longueur $|P| = m$, $P = c_1c_2 \cdots c_m$. Supposons l'existence dans P d'une seule erreur ; alors P peut s'écrire $P = P_1\phi P_2$ où $P_1 = c_1c_2 \ldots c_{i-1}$, ϕ : représente l'erreur à la position i, et $P_2 = c_{i+1}c_{i+2} \ldots c_m$.

Si on suppose qu'on a une solution approchée $P' \in D$ et $P' = P_1\, a\, P_2,\ a \in \Sigma$ où P_1 et P_2 sont deux fragments exacts, alors P_1a et aP_2 doivent exister respectivement dans le Trie et dans le Trie inversé.

Nous pouvons conclure de cette observation que, seul le chemin sortant marqué par la lettre a doit être vérifié pour trouver l'autre pièce exacte P_2 dans le Trie.

Nous devons obtenir toutes les lettres $a \in \Sigma$ qui satisfont cette propriété $P' = P_1aP_2$.

L'idée revient à faire une recherche exacte de P_1 dans le Trie (recherche de gauche vers la droite), et une recherche exacte pour P_2 dans le Trie inversé (recherche de droite à gauche) pour le mot requête, puis faire une intersection entre toutes les transitions sortantes des deux nœuds qui représente la position de l'erreur, pour avoir les caractères qui pourraient conduire aux solutions.

La dernière étape consiste à vérifier si P_1 et P_2 appartiennent au même mot, pour s'assurer que la réponse n'est ni de la forme P_1aZ, ni de la forme QaP_2 où Q, Z sont deux sous-chaînes différentes de P_1, P_2 respectivement.

Dans la prochaine sous-section, nous expliquons et nous détaillons notre méthode qui s'appuient sur le Trie et le Trie inversé afin de réduire le nombre des transitions sortantes dans chaque nœud, et vérifier que quelques chemins seulement, ceux qui peuvent conduire à des solutions (voir la figure 5.2).

5.2.2 Principe de l'algorithme

Soit $P = P_1\phi P_2$ le mot requête où P_1 et P_2 sont deux fragments exacts, et ϕ un caractère joker qui représente la position de l'erreur.

Notons v_{err} le nœud du Trie de la position de l'erreur (où la recherche s'arrête). De façon similaire, notons w le nœud où la recherche de P_2 s'arrête dans le Trie inversé.

Soit $A = \{a_1, a_2, ..., a_{L1}\}$, et $B = \{b_1, b_2, ..., b_{L2}\}$ les ensembles des caractères qui étiquettent les transitions sortantes de v_{err} et w respectivement.

Afin de trouver les caractères communs, on effectue une intersection entre les deux ensembles :

$C = A \cap B = \{c_1, c_2, ..., c_{L3}\}$.
Si une solution approchée existe, cela signifie qu'il existe un caractère c_k avec $k \in [1..L3]$, et le mot $P' = P_1 c_k P_2,\ P' \in D$ est une solution.

Après l'opération de l'intersection, l'ensemble résultat C contiendra 2 types d'éléments :

1. Les caractères qui mènent à des solutions : dans ce cas, on a $P_1 c_k$, et $c_k P_2$ et les deux sous-chaînes (P_1, P_2) sont dans le même mot.
2. Les caractères qui ne mènent pas aux solutions : lorsqu'on a $P_1 c_k$ et $c_k P_2$ ne sont pas dans le même mot, autrement dit $P_1 c_k Z$ et $Q c_k P_2$ sont deux mots différents.

En raison de ces deux cas, nous devons vérifier dans le Trie que $P_1 c_k P_2$ est dans le même mot.

Remarque :
Sachant $(P_1 c_k)$ est déjà trouvé dans le Trie, il reste juste à tester l'existence de (P_2) dans le Trie sur le même chemin étiqueté par (c_k). De même, nous pouvons faire cette vérification dans le Trie inversé, c'est-à-dire tester l'existence de P_1 sur le chemin étiqueté par c_k qui sort de P_2.

Pour réduire le nombre de comparaisons, nous vérifions la plus petite chaîne d'entre P_1 et P_2. En effet, si $(|P_1| > |P_2|)$ on vérifie P_2 dans le Trie, autrement, on vérifie P_1 dans le Trie inversé.

Pour trouver toutes les solutions, nous devons déterminer toutes les positions possibles de l'erreur.

5.2.3 Les positions possibles de l'erreur

Pour un motif donné $P = P_1 \phi P_2$, $|P| = |P_1| + 1 + |P_2| = m$. La recherche exacte de (P) dans le Trie donne (P_1) ; la position d'erreur est automatiquement trouvée lorsqu'on ne peut pas avancer dans le Trie.

Puisque $k = 1$, seuls les cas suivants sont possibles (voire Fig. 5.1) :

a. Il n'est pas possible de trouver une autre erreur après $|P_1| + 1$, sinon nous comptabilisons deux erreurs. Donc, Lorsque nous cherchons P_2 $(|P_2| = m - |P_1| - 1)$ dans le Trie inversé, si nous trouvons une erreur avant la longueur $|P_2|$, par conséquent, il n'existe aucune solution. Soit q le fragment (la sous-chaîne) trouvé dans le Trie inversé avant la position d'erreur w (qui satisfait $|P_2|$), avec $|q| < |P_2|$. Cela signifie que nous avons $|P_2| - |q|$ positions à ne pas vérifier.

b. Soit $V = \{v_1, v_2, ..., v_k, v_{err}\}$ l'ensemble des nœuds sur le chemin qui mène de la racine à v_{err}. Tous les $(v_i \in V)$ peuvent être une position d'erreur.

Lorsque nous appliquons une distance d'édition, un autre chemin (différent de celui de la recherche exacte) peut conduire à une solution, voire Fig. 5.1.

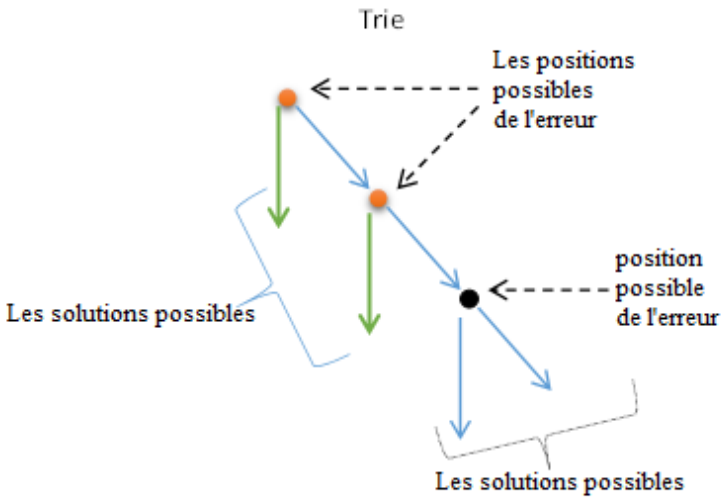

FIGURE 5.1 – Toutes les positions possibles de l'erreur dans le Trie et le Trie inversé. Toutes les transitions sortantes de ces nœuds mènent à des solutions possibles.

Comme expliqué précédemment, dans chaque position possible de l'erreur, seulement quelques chemins seront examinés (voire Fig. 5.2).

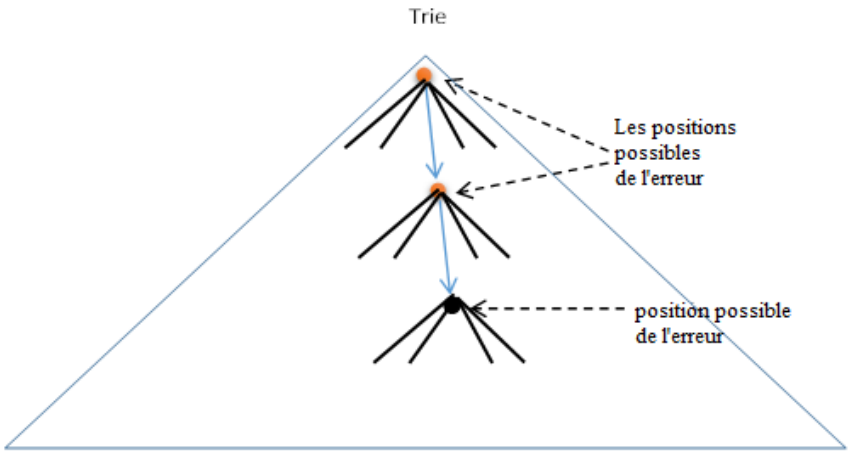

FIGURE 5.2 – Quelques transitions sortantes à partir du nœud erreur seront vérifiées.

5.2.4 L'algorithme de recherche

Algorithme de la recherche approchée TRT_CI

Entrée : un mot requête q de longueur m, les structures de données Trie et Trie inversé.
Sortie : liste des mots solutions approchées.

a. Faire une recherche exacte dans l'arbre Trie, jusqu'à ce que l'on arrive à la position de l'erreur. Soit v_{err} le nœud qui représente cette erreur. Soit P_1 le préfixe avant cette position.

b. Faire une recherche pour le suffixe P_2 dans le Trie inversé, jusqu'à ce que la longueur de ce suffixe soit consommé, ($|P_2| = m - |P_1| - 1$). Soit w le nœud dans lequel la recherche s'arrête dans le Trie inversé.

c. Si on ne peut pas trouver P_2, l'algorithme s'arrête (il n'y a pas de solution).

d. Sinon, on a deux nœuds v_{err} et w ; on calcule une intersection entre les deux ensembles des caractères des transitions sortantes de ces deux nœuds. Soit C l'ensemble des caractères résultant.

e. Pour chaque caractère $c_k \in C$, continuer une recherche exacte en suivant la transition sortante étiquetée par c_k comme suit :
   - Si ($|P_1| > |P_2|$) alors continuer la recherche exacte de P_2 dans le Trie.
   - Sinon continuer la recherche exacte de P_1 dans le Trie inversé.

f. Pour chaque $v_i \in V$ où $V = \{v_1, v_2, ..., v_k\}$ sont les nœuds sur le chemin qui mène de la racine à v_{err}, répétez les étapes : b., c., d. et e.

5.2.5 Des cas particuliers

Deux cas particuliers doivent être traités séparément : le cas où P_1 ou bien P_2 sont vides, et le cas où l'on trouve une solution exacte pour le motif recherché P.

- Le cas où l'une des deux sous-chaînes est vide (préfixe ou suffixe) :
  Soit $P = P_1\phi P_2$ et $P_1 = \epsilon$ ou $P_2 = \epsilon$, donc, on a $P = P_1\phi$ ou bien $P = \phi P_2$. Dans ce cas, l'intersection est faite entre un nœud interne et le nœud racine. Exécuter une intersection dans ce cas serait inefficace. Pour éviter ce calcul, nous appliquons une distance d'édition (que la suppression et la substitution) à la position finale de la sous-chaîne non-vide.

- Le cas où l'on trouve une solution exacte : Lorsqu'on effectue une recherche exacte sur le mot requête dans le Trie et dans le Trie inversé et aucune erreur n'est rencontrée.

  Soient n_1 et n_2 les nœuds dans lesquels la recherche exacte se termine dans le Trie et le Trie inversé, respectivement. Nous traitons ce cas comme suit :

– La première étape : dans les deux nœuds n_1 et n_2, seule l'erreur d'insertion est examinée.

– La deuxième étape : après la première étape, on aura $P = P_1\phi P_2$, où $|P_1| \geq 1$ et $|P_2| \geq 1$, tous les nœuds sur le chemin qui mène de la racine à n_1 (ou n_2) seront considérés comme des nœuds de l'erreur.

5.2.6 Le cas de l'insertion et la suppression

La suppression : Soit un mot requête $P = P_1\phi P_2$ avec $|P| = m$. Soit v_{err} et w deux nœuds qui représentent la profondeur de $|P_1|$ et $|P_2|$ respectivement. Pour faire une recherche avec une suppression, il suffit juste de sauter un seul caractère (la position de l'erreur) dans le mot requête. Cela revient à faire une recherche exacte de mot $P' = P_1\,P_2$.

Lorsque on applique l'algorithme 5.2.4 pour faire la recherche des solutions avec substitution, en même temps, on cherche les solutions avec une suppression comme suit : Si $|P_1| > |P_2|$, alors on continue à vérifier P_2 dans le Trie à partir du nœud v_{err}. Sinon, on continue à vérifier P_1 dans le Trie inversé à partir du nœud w.

Le cas de l'insertion : Soit le mot requête $P = P_1\,P_2$ d'une longueur m. Soit le pattern $P' = P_1\phi P_2$ qui représente une solution approchée avec une insertion, avec $|P'| = m + 1$, et ϕ est la position de l'erreur de type insertion.

Pour trouver toutes les solutions approchées avec une insertion, on fait une recherche avec substitution avec le même algorithme 5.2.4 sur le mot requête $P' = P_1\phi P_2$. On cherche la première partie $|P_1|$ dans le Trie pour déterminer le nœud v_{err}, ensuite, on cherche P_2 dans le Trie inversé ($|P_2| = m - |P_1|$) pour déterminer le nœud w, ensuite, on fait une intersection entre les caractères des fils des deux nœuds afin de continuer la vérification de l'un des parties (P_1 ou P_2) en suivant les caractères en commun.

5.2.7 L'intersection

Chaque nœud à au maximum σ fils (le nombre total des caractères de l'alphabet). Le but est de faire une intersection entre deux ensembles des caractères (S_1 et S_2) des transitions sortantes de deux nœuds.

Chaque ensemble de l'alphabet est ordonné et représenté par un tableau de bits (si le caractère existe donc 1, et 0 sinon).

Si $|S_1| < |S_2|$, alors on vérifie l'existence des éléments de S_1 dans S_2. Comme les caractères sont ordonnés, alors on a un accès direct à la bonne case (le bon bit) dans le tableau de bits. Cela donne comme complexité $O(|S_1|)$ avec $|S_1| < |S_2|$.

5.2.8 Évaluation de la complexité

Théorème 5 *Étant donné un dictionnaire D équiprobable de d mots, et un mot requête q d'une longueur m. La complexité en pire cas est en $O(\sigma m^2)$. La complexité temporelle moyenne est $O(m^2)$ si $m \geq 2\frac{\log d}{\log \sigma}$ (elle est en $O(m)$ pour chaque position). La complexité est en $O(m + m.occrs)$ si $m \geq 2(\frac{logd}{log\sigma} + \frac{logm}{log\sigma})$, $occrs$ représente le nombre des occurrences de la solution.*

Preuve 5 *Étant donné un dictionnaire équiprobable D de d mots, et un mot requête $q = P_1 c P_2$ d'une longueur m. La complexité temporelle en pire cas est de $O(\sigma m^2)$:*
Dans chaque position $i \in [1..m]$ on teste tous les σ caractères possibles (pour insertion ou substitution) : nous avons m positions, ce qui donne σm branches du Trie à explorer. Pour chaque branche nous avons un mot candidat que l'on doit vérifier dans le Trie en un temps $O(m)$ en descendant depuis cette branche en faisant une recherche exacte pour tous les caractères restant du mot. Donc, le temps total est de $O(\sigma m^2)$ dans le pire cas.

1. *Le cas en $O(m^2)$. L'algorithme cherche P_1 dans le Trie, et P_2 dans le Trie inversé, ensuite après l'étape de l'intersection, l'algorithme doit vérifier la partie la plus courte entre P_1 et P_2. Si on suppose que le résultat de l'intersection est non nul. Pour $m \geq 2\frac{\log d}{\log \sigma}$, la taille de l'une des deux parties P_1 ou P_2 est au moins égale à $m' \geq \frac{1}{2}m$. Soit $P' = P_1$ si $P_1 > P_2$ et $P' = P_2$ sinon. Nous avons donc $|P'| \geq m'$. En supposant que $m' \geq \frac{\log d}{\log \sigma}$ et donc $|P'| \geq \frac{\log d}{\log \sigma}$. Dans ce qui suit on suppose que $P' = P_1$, le cas $P' = P_2$ étant symétrique. La probabilité pour que les m' premiers caractères d'un mot du dictionnaire soit égaux aux m' premiers caractères de P_1 est donc inférieure à $1/\sigma^{\frac{\log d}{\log \sigma}} = 1/d$, et donc le nombre de mots candidats en moyenne est de $O(d \cdot (1/d)) = O(1)$. Comme la recherche exacte sur les caractères restant prend un temps $O(m)$, le temps moyen est de $O(m)$. Donc, pour les m positions la complexité devient $O(m^2)$.*
2. *Le cas en $O(m + m \times occrs)$. On suppose maintenant que $m \geq 2(\frac{\log d}{\log \sigma} + \frac{\log m}{\log \sigma})$. On raisonne de la même façon que précédemment avec pour seul changement la valeur de m' qui est maintenant égale à $m/2 = \frac{\log d}{\log \sigma} + \frac{\log m}{\log \sigma}$. La probabilité pour qu'un mot du dictionnaire soit candidat devient $1/\sigma^{m'} = 1/(m \times d)$ et donc le nombre de mots candidats pour chaque position devient en moyenne égal à $d/(m \times d) = 1/m$. Donc le temps moyen pour vérifier chaque candidat devient $O(m \times (1/m)) = O(1)$, qui donne un temps de $O(m)$ pour l'ensemble des positions. On ajoute un temps de $m \times occrs$ pour les autres occurrences de la solution si elles existent (car chaque solution est en $O(m)$).*
   La complexité moyenne finale est donc : $O(m + m \times occrs)$.

Nous nous référons à notre méthode par **TRT_CI**, qui est un acronyme pour la phrase en Anglais : (Trie and reverse Trie Characters Intersection), qui ce traduit par Trie et Trie inversé Caractères Intersection.

5.3 Une amélioration de la méthode de Amir et al.

Dans la méthode précédente TRT_CI nous avons expliqué comment localiser la position de l'erreur, et ensuite, comment réduire le nombre de transitions afin de tester uniquement celles qui mènent à des solutions possibles.

Soit $P = P_1 \phi P_2$, d'une façon très simple, l'erreur divise le mot en deux parties P_1 et P_2. Le but est de s'assurer que ces deux parties appartiennent au même mot dans le dictionnaire.

L'idée simple suivante peut être utilisée : supposant que l'on puisse déterminer le numéro du mot auquel les parties P_1 et P_2 appartiennent, cela veut dire que le motif qui a la forme $P = P_1 \phi P_2$ représente une solution approchée avec $k = 1$ erreur si P_1 et P_2 appartiennent au même mot.

Pour faire cela, supposons que dans le Trie, au moment de sa construction, on stocke les numéros des mots dans les feuilles. Ensuite, on utilise le Trie pour chercher P_1, et on récupère tous les numéros qui sont stockés dans les fils du nœud qui est à la fin de P_1. On fait le même traitement pour P_2 dans le Trie inversé. On obtient deux ensembles qui contiennent les numéros des mots. On fait une intersection pour vérifier si P_1 et P_2 appartiennent au même mot, et obtenir la solution.

Cette idée été utilisée par Amir et al. [83]. Dans leur travail, ils font un pré-traitement afin de remédier au problème de l'intersection. Ils obtiennent un résultat qui un temps de pré-traitement en $O(n\, log^2 n)$, et un temps de recherche en $O(m\, log^3 d\, log\, log\, d + occr)$ pour le dictionnaire, et $O(m\, log^2 n\, log\, log\, n + occr)$ pour l'indexation de texte, où n est la taille du texte et d est la taille du dictionnaire, et $occr$ est le nombre des occurrences de la solution.

Amir et al. dans [83] font toutes les combinaisons de la position de l'erreur dans le mot requête. Chaque fois, ils modifient la position de l'erreur et essaye de trouver des nœuds dans le Trie et dans le Trie inversé, pour faire une intersection entre les numéros des mots stockés dans les feuilles des deux structures de données. Partant de cette observation, nous notons les points suivants :

1. Comme nous avons expliqué dans la sous-section 5.2.3, nous avons juste certaines positions possibles à considérer comme position de l'erreur. Donc si à la position i il y a une erreur, nous allons perdre du temps pour vérifier les autres positions dans l'intervalle $[i + 1..m]$ (m la longueur de mot requête), parce que nous aurons plus d'une seule erreur.
2. Si l'erreur est à la fin de mot, dans ce cas, l'intersection se fait avec tous les mots du dictionnaire, et cela implique un temps de traitement élevé.
3. Si nous utilisons un Trie compact, nous ne pouvons pas trouver un nœud à la position i, si cette position est au milieu d'une arête qui a de nombreux caractères.

Pour améliorer la méthode de Amir et al [83], nous utilisons les mêmes heuristiques

utilisées dans la première méthode *TRT_CI* pour trouver les positions possibles de l'erreur et pour éviter de faire une intersection avec le nœud racine comme nous l'avons expliqué dans la sous-section 5.2.5.

Dans ce que suit nous présentons la structure de données et notre algorithme de recherche pour améliorer la méthode de Aimr et al.

5.3.1 La construction de la structure de données

Notre algorithme de recherche est basé sur la même structure de données que celle de Amir et al. [83]. La construction de la structure de données est comme suit :

Algorithme de construction de la structure de données TRT_WNI
Entrée : fichier contenant les mots du dictionnaire.
Sortie : structure de données TRT_WNI.

1. Construire le Trie du dictionnaire et stocker les numéros des mots dans les feuilles.
2. Construire le Trie inversé et stocker les numéros des mots dans les feuilles.
3. Dans chaque structure de données, relier toutes les feuilles de la gauche vers la droite, pour obtenir une liste des numéros de mots.
4. Stocker dans chaque nœud interne le pointeur de la feuille de son fils le plus à gauche et son fils le plus à droite, donc le début et la fin de la sous-liste qui contienne tous les fils sortant de ce nœud.

5.3.2 L'algorithme de recherche

Les étapes de l'algorithme ressemble à celles de notre méthode *TRT_CI* expliqué dans la sous-section 5.2.4. La différence avec *TRT_CI* est dans l'étape de la vérification de l'appartenance de P_1 et P_2 au même mot.

Algorithme de la recherche approchée TRT_WNI
Entrée : un mot requête q de longueur m, la structure de données TRT_WNI.
Sortie : liste des mots solutions approchées.

a. Faire une recherche exacte dans le Trie, jusqu'à ce qu'on arrive à la position de l'erreur, on trouve le nœud v_{err} qui représente cette erreur. Soit P_1 le préfixe avant la position de l'erreur.

b. Faire une recherche pour le suffixe P_2 dans le Trie inversé, jusqu'à ce que la longueur de ce suffixe soit consommée, ($|P_2| = m - |P_1| - 1$). Soit w le nœud dans lequel la recherche s'arrête dans le Trie inversé.

c. Si on ne peut pas trouver P_2, l'algorithme s'arrête (il n'y a pas de solution).

d. Sinon, on a deux nœuds v_{err} et w, on fait une intersection entre les deux ensembles des numéros des mots stockés dans les feuilles sortantes de ces deux nœuds.

e. Pour chaque $v_i \in V$ où $V = \{v_1, v_2, ..., v_k\}$ sont les nœuds sur le chemin qui mènent de la racine à v_{err}, répétez les étapes : b. c. et d.

5.3.3 Un vecteur de bits simple pour faire l'intersection

Dans notre travail on utilise une méthode simple pour faire l'intersection sans prétraitement. Nous utilisons un vecteur de bits de longueur d, initialisé à 0, où d est le nombre total des mots du dictionnaire.

On prend le premier ensemble des numéros des mots du Trie et on marque leurs positions dans le vecteur de bits par 1. Ensuite, on prend le deuxième ensemble des numéros des mots du Trie inversé, et pour chaque numéro de mot, on vérifie si dans sa position dans le vecteur de bits est marqué 1. Si oui, donc cette position est le numéro de mot qui représente une solution.

Dans le pire des cas, cette opération peut se faire dans une complexité temporelle de $O(n)$, mais si le dictionnaire est trié, nous pouvons la faire dans $O(n/w)$ où w est un mot mémoire dans le modèle RAM et $w = log(n)$ avec n le nombre total de caractères, car il suffit juste de faire l'opération (AND), où chaque mot mémoire prend un temps de $O(1)$.

Nous nous référons à notre deuxième méthode par *TRT_ WNI* qui est un acronyme pour la phrase en Anglais : (Trie and Reverse Trie Words Number Intersection) qui signifie Trie et Trie inversé, l'intersection des numéros des mots.

5.4 Une méthode hybride : une combinaison des deux méthodes précédentes

Afin d'améliorer la seconde méthode *TRT_ WNI* (voir la section 5.3), nous combinons les deux techniques, le filtrage des transitions sortantes de la méthode *TRT_ CI*, et l'intersection des numéros de mots de la méthode *TRT_ WNI*.

Nous utilisons la première approche pour trouver les nœuds de l'erreur, et pour choisir seulement les nœuds qui sont à la fin des premières transitions sortantes depuis le nœud

erreur (v_{err}/w), ces transitions peuvent probablement conduire à des solutions. Nous appliquons la deuxième approche pour effectuer une intersection entre les numéros des mots stockés dans les feuilles sortantes des nœuds choisis dans la première étape.

La structure de données est exactement la même, celle définie dans la méthode précédente, voir la sous-section 5.3.1. Les étapes de l'algorithme de recherche de cette troisième méthode sont comme suit :

Algorithme de la recherche approchée TRT_CWNI
Entrée : un mot requête q de longueur m, la structure de données TRT_WNI.
Sortie : liste des mots solutions approchées.

1. Obtenir les deux nœuds de l'erreur v_{err} et w :
   (a) Faire une recherche exacte dans le Trie, jusqu'à ce qu'on arrive à la position de l'erreur, on trouve le nœud v_{err} qui représente cette erreur. Soit P_1 le préfixe avant la position de l'erreur.
   (b) Faire une recherche pour le suffixe P_2 dans le Trie inversé, jusqu'à ce que la longueur de ce suffixe est consommée, ($|P_2| = m - |P_1| - 1$). Soit w le nœud dans lequel la recherche s'arrête dans le Trie inversé.
   (c) Si on ne peut pas trouver P_2, on arrête l'algorithme de recherche (il n'y a pas de solution).
2. À partir des deux nœuds (v_{err} et w), obtenir tous les nœuds obtenus par les transitions sortantes qui peuvent conduire à des solutions. Soit v'_{err} l'ensemble résultants du Trie, $v'_{err} = \{v_{err1}, v_{err2}, ..., v_{errL1}\}$, chaque $v_{err\,i}$ conduit à une solution possible. Et de même pour le Trie inversé, on a $w' = \{w_1, w_2, ..., w_{L2}\}$, donc on a deux ensembles v'_{err} et w'.
3. Faire une intersection entre les numéros de mots stockés dans les feuilles obtenus de chaque nœud dans les deux ensembles. v'_{err} et w'.
4. Pour chaque $v_i \in V$ où $V = \{v_1, v_2, ..., v_k\}$ sont les nœuds sur le chemin qui mènent de la racine à v_{err}, répéter les étapes : 1b, 1c, 2, et 3.

La troisième méthode est appelée *TRT_CWNI* pour la phrase en Anglais : (Trie and reverse Trie, Characters and Words Number Intersection) qui signifie Trie et Trie inversé, l'intersection des caractères et l'intersection des numéros des mots.

5.5 Les expérimentations et les analyses

Les expérimentations ont été faites avec les deux ensembles de données suivants : une version extraite du dictionnaire des titres de wikipédia (Wikititle) qui a environ 1,2 millions de mots, et le dictionnaire Anglais qui contient environ deux cent treize mille mots.

L'implémentation est faite dans le langage C, en utilisant le compilateur GNU C (GCC), la version 4.7.1. Les tests ont été effectués sur Windows 8.1 Pro 64 bits, avec un processeur Intel Core i7-2670QM 2,20 GHz et 6 Gigabyte de RAM. Nous avons utilisé un seul coeur.

Le code source de la méthode *TRT_CI* est disponible sur :
https://github.com/chegrane/TrieRTrie/tree/master/TrieRTrie_char_inter.
Les méthodes de Amir et al., *TRT_WNI*, et *TRT_CWNI* sont disponibles sur :
https://github.com/chegrane/TrieRTrie.
Les deux ensembles de données sont disponibles sur :
https://github.com/chegrane/TrieRTrie/tree/master/dataset.

Toutes nos implémentations sont distribuées sous la licence publique générale limitée GNU (GNU LGPL) (Anglais : GNU Lesser General Public License).

Nous avons comparé nos résultats aux travaux de Karch et al [88], et Chegrane et Belazzougui [16], car leurs résultats sont très compétitifs par rapport à d'autres approches qui ont une implémentation pratique.

Nous comparons, aussi, nos résultats avec un travail pratique et récent de Aleksander Cisłak et Szymon Grabowski [1], qui traite la distance de Hamming seulement et qui donne des résultats intéressants.

Pour réaliser nos tests, nous avons choisis 100 mots d'une façon aléatoire de notre fichier du dictionnaire. Nous avons appliqué une opération d'édition choisie aléatoirement. À la fin, nous avons effectué une recherche approchée sur ces mots requête. Nous avons répété la procédure 10 fois, et nous avons calculé la moyenne des résultats. Les résultats sont résumés dans le tableau 5.1.

Méthode	Anglais	Wikititle
Karch et al	8	34
Chegrane et Belazzougui	4,55	8,28
TRT_CI	0,77	0,78

TABLE 5.1 – Comparaison entre des méthodes existantes et notre méthode *TRT_CI* sur les ensembles de données WikiTitle, et Anglais. (Le temps d'exécution est en (μs)).

Dans le tableau 5.1, on voit clairement que la méthode *TRT_CI* est plus efficace en terme de temps d'exécution comparée aux deux autres méthodes testées dans ce travail.

En fait, notre méthode *TRT_CI*, est d'environ 6 fois plus rapide que celle de [16], dans le dictionnaire Anglais, et environ 10 fois plus rapide dans le dictionnaire WikiTitle. Elle est aussi d'environ 10 fois plus rapide que celle de [88] dans l'ensemble de données Anglais, et environ 43 fois plus rapide dans le dictionnaire WikiTitle.

Les expérimentations sont effectuées sur des différentes tailles du dictionnaire. Nous avons fait varier le nombre de mots dans le dictionnaire, pour observer l'influence de ce paramètre sur le temps d'exécution, voir la figure 5.3, (Pour Karch et al. juste un seul point est montré, parce que nous n'avons qu'un seul résultat, les résultats de Karch et al. ne permettent pas de faire varier le temps de réponse en fonction du changement de la taille du dictionnaire).

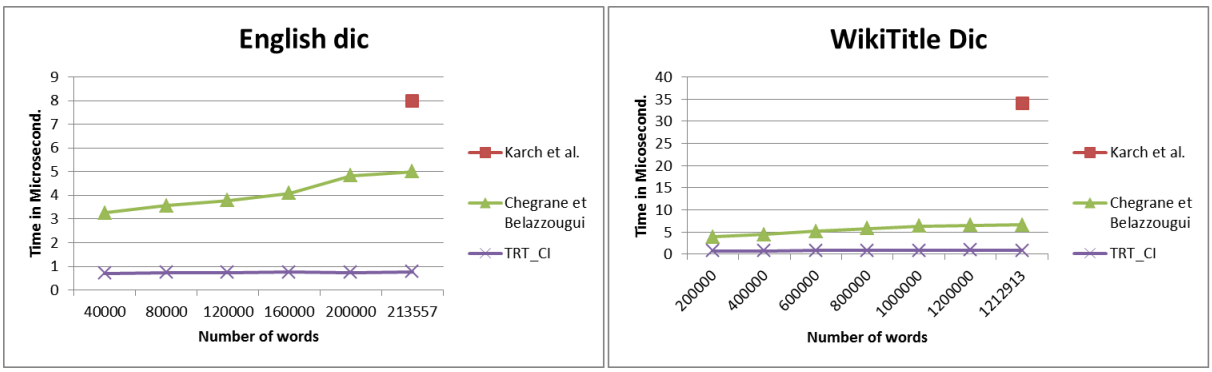

FIGURE 5.3 – Temps d'exécution en fonction du nombre de mots dans les dictionnaires Anglais et WikiTitle.

Dans la figure 5.3, on voit que contrairement aux méthodes [88] et [16], l'augmentation de la taille du dictionnaire n'affecte pas le temps d'exécution de notre méthode *TRT_CI*. Dans la méthode *TRT_CI*, le temps est presque constant (environ 0,77 μs), car elle ne dépend que du résultat de l'intersection sur les caractères communs des transitions sortantes des deux nœuds de l'erreur dans le Trie et dans le Trie inversé, et dans le pire cas, on a $|\Sigma| = \sigma$ chemins à vérifier.

En pratique, l'intersection donne un ensemble très petit, beaucoup moins que le nombre de caractères de l'alphabet σ. De plus la vérification s'arrête immédiatement si une erreur est rencontrée, car on ne peut pas avoir deux erreurs. La recherche exacte des deux chaînes P_1 et P_2 se fait une seule fois dans le Trie et le Trie inversé respectivement. Tous les nœuds traversés dans la recherche exacte de P_1 et P_2 sont stockés dans deux vecteurs, pour nous permettre d'effectuer une intersection entre les caractères qui sont sur les transitions sortantes des deux nœuds (qui sont dans ces vecteurs directement). Après le filtrage des transitions sortantes, la vérification de chaque chemin s'arrête directement si une erreur est rencontrée, autrement cette voie conduira à une solution.

Aleksander Cisłak et Szymon Grabowski dans leur travail [1], partitionnent chaque mot de dictionnaire d_i en $k+1$ morceaux $d = \{p_1, p_2, ..., p_{k+1}\}$. Chaque morceau joue

le rôle d'une clé pour stocker le mot d'origine dans une table de hachage (avec chaînage externe). Au total, il y a $k+1$ listes de mots à stocker dans la table de hachage pour chaque mot d_i. Au lieu de stocker la totalité du mot, ils stockent seulement les parties qui sont différentes du morceau qui joue le rôle d'une clé, et ils stockent comme indice le numéro du morceau qui manque dans le mot stocké. Les auteurs de [1], se sont basés sur le cas de $k = 1$, où ils ne stockent que le préfixe et le suffixe du mot d_i dans la table de hachage. Ils ne stockent que le préfixe et le suffixe de mot dans la table de hachage, ils traitent les listes de mots sans leurs préfixes en premier, ensuite, les listes de mots sans leurs suffixes, et ajoutent, dans chaque liste, la position où le dernier morceau commence. Le travail de [1] concerne seulement la distance de Hamming.

La recherche de mot requête se fait comme suit : décomposer le mot requête q en $k+1$ morceaux. Pour chaque morceau p_i, chercher la liste l_i correspondante depuis la table de hachage (pour $k = 1$, on a seulement le préfixe et le suffixe). Pour chaque mot m_j (le mot d'origine moins le morceau p_i) stocker dans la liste, si $|m_j| = |q| - |p_i|$, alors vérifier si $dist\_Hamming(m_j, q-p_i) \leq k$, ($q-p_i$ signifie qu'on enlève le morceau p_i du mot requête q). Si la distance de Hamming est satisfaite, alors les morceaux (m_j et p_i) sont combinés dans un seul mot pour le présenter comme une solution approchée.

Les résultats expérimentaux de [1] ont été obtenus sur une machine équipée du processeur Intel i5-3230M cadencé à 2,6 GHz et 8 Go de mémoire DDR3, et le code C++ a été compilé avec la version clang 3.4-1 et exécuter sur Ubuntu 14.04 OS.

Cette configuration est proche de celle utilisée dans ce chapitre. Nous n'avons pas testé leur code sur notre machine par manque de temps (et aussi, nous n'avons pas leur code source). Nous allons utiliser le graphe (voir la figure 5.4) de leurs résultats afin de faire la comparaison.

On peut remarquer depuis la figure 5.4 (les résultats de [1]) les points suivants :

1. Le graphe montre que les méthode de [1] donnent de meilleurs résultats comparés à ceux de notre méthode de recherche approchée avec hachage (chapitre 4 précédent) et ceux de Boytsov. Mais cette comparaison n'est pas totalement correcte, car la méthode de [1] ne traite qu'un seul type d'erreur (la substitution). Le temps de calcul augmente lorsqu'on traite les trois types d'erreurs (intuitivement, le temps de traitement pour trois erreurs est de 3 fois plus). Donc, par conséquence, on ne peut pas dire si les résultats de [1] sont meilleurs ou pas.

2. Notre méthode de recherche approchée avec hachage (chapitre 4 précédent) donne de meilleurs résultats par rapport à ceux de Boytsov. Les deux méthodes traitent les 3 types d'erreurs de distance d'édition, donc ils sont directement comparables.

3. Le temps de calcul de la méthode sans compression de [1], qui donne le meilleur résultat dans le graphe est proche de $1\mu s$. Le résultat de notre méthode TRT_CI est d'environ 0,77 μs donc proche de 1 μs aussi. Sauf que la méthode de [1] traite

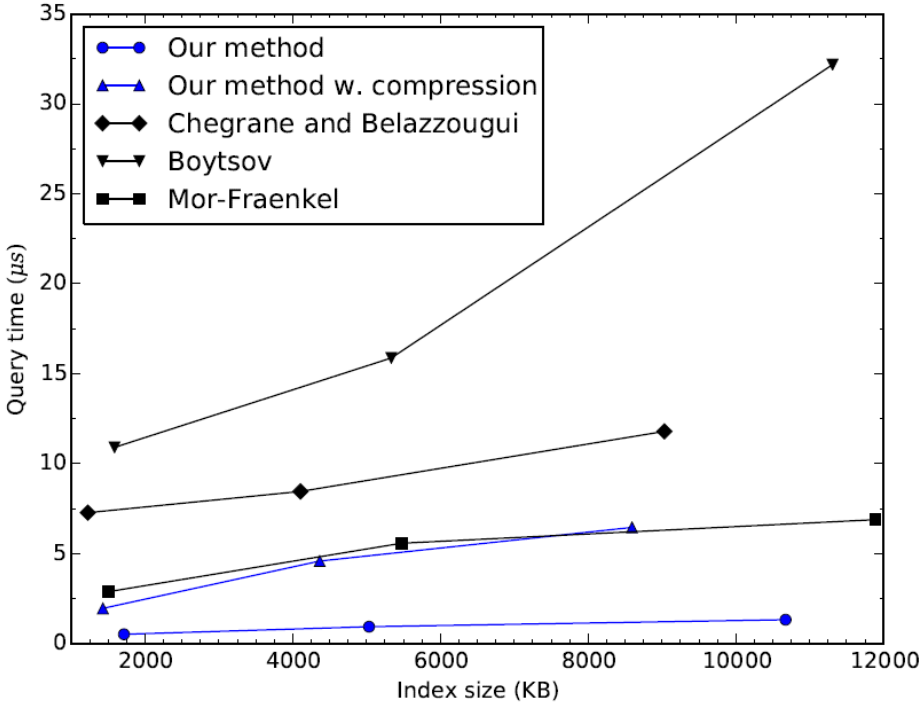

FIGURE 5.4 – Les résultats expérimentaux de [1] sur le dictionnaire anglais pour k=1 avec la distance de Hamming.

une seule erreur (la substitution) seulement, par contre la nôtre traite les trois types d'erreurs en même temps. Cela implique que notre méthode donne de meilleurs résultats par rapport à ceux de [1] (on peut dire qu'elle est 3 fois plus rapide).

4. Le temps de calcul de la méthode sans compression de [1] (qui donne le meilleur résultat) augmente avec le changement de la taille du dictionnaire anglais (la taille du dictionnaire anglais est petite et ne permet pas de donner un bon aperçu), d'après le graphe, on peut déduire que si la taille du dictionnaire augmente, le temps de calcul augmente lui aussi. Quant à notre méthode *TRT_CI*, le temps de calcul est invariant avec l'augmentation de la taille du dictionnaire. Cela signifie que notre méthode est meilleure même avec ce rapport de différence dans le nombre d'erreurs traitées.

5.5.1 L'efficacité de la méthode *TRT_CI*, par rapport à la recherche exacte

La recherche exacte est l'opération la plus basique qui peut être effectuée sur un dictionnaire de mots. Lorsque la recherche est effectuée dans un Trie, la complexité est proportionnelle à la longueur du mot requête. Pour cette raison, on peut la considérer comme une opération de référence qui donne le temps d'exécution le plus optimal par rapport à toutes les autres opérations (la recherche approchée avec $k > 0$).

Puisque le temps d'exécution de notre méthode *TRT_CI* est indépendant de la taille

du dictionnaire D, comme dans la recherche exacte en utilisant un Trie, nous avons fait des expérimentations sur les mêmes ensembles de données (les dictionnaires Anglais et WikiTitle), afin de trouver une relation entre la recherche exacte en utilisant un Trie et notre méthode TRT_CI, voir la figure 5.5.

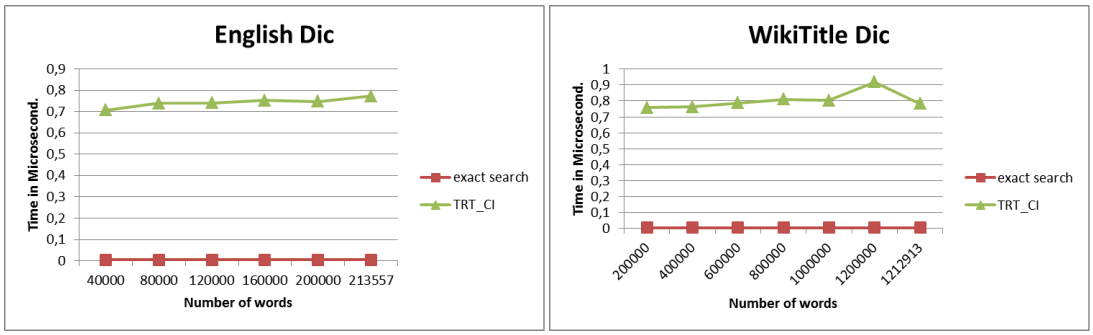

FIGURE 5.5 – La relation entre la méthode TRT_CI et la recherche exacte.

Les résultats expérimentaux de la comparaison entre la méthode TRT_CI et la recherche exacte montrent qu'effectivement, il existe une relation presque constante entre le temps d'exécution de ces deux méthodes, cette relation est décrite comme suit :

Le temps d'exécution de TRT_CI est d'environ 167 fois le temps d'exécution de la recherche exacte. Le facteur 167 s'explique par la différence entre la recherche approchée et la recherche exacte. La recherche exacte (en utilisant un Trie) retourne juste une seule solution, mais la recherche approchée renvoie toutes les solutions valides pour le mot requête, ce qui signifie qu'elle peut impliquer de nombreuses recherches exactes.

Enfin, on remarque que les expérimentations sont réalisées sur des dictionnaires de langues (Anglais, WikiTitle), où les ensembles des mots comme :
$\{P_1a_1P_2, P_1a_2P_2, \ldots, P_1a_{|\Sigma|}P_2,\}$, $(a_i \in \Sigma)$ sont rares. L'étape de vérification est donc réduite à quelques mots seulement.

Dans ce qui suit, nous interprétons seulement les tests et les analyses de nos deux méthodes TRT_WNI et TRT_CWNI afin de souligner l'amélioration pratique apportée à l'algorithme de Amir et al.

Pour notre première méthode TRT_CI, nous allons juste mettre les résultats pour voir la différence avec les autres méthodes. Car cette méthode est beaucoup plus performante que toutes les méthodes testées dans ce chapitre. Entre les quatre méthodes qui se basent sur le Trie et le Trie inversé, le fonctionnement de notre première méthode TRT_CI est lié à la taille de l'alphabet du dictionnaire qui est très petit σ. Contrairement aux autres méthodes (TRT_WNI, TRT_CWNI, et celle de Amir et al.) elles sont tous liées au nombre total de mots du dictionnaire n qui est un grand nombre et cela rend le temps de l'intersection très grand.

Donc on a deux groupes de méthodes, celles qui se basent juste sur la taille du mot et la taille de l'alphabet, le temps de recherche est proportionnel à m et σ qui sont de petits

nombres, et l'autre qui est liée à la taille du mot requête et le nombre total des mots du dictionnaire, donc le temps de recherche est proportionnel à n.

5.5.2 Test des deux méthodes TRT_WNI et TRT_CWNI

Amir et al. ont fait un travail théorique et ils n'ont pas une implémentation de leur méthode, afin de pouvoir comparer nos méthodes avec la leur, nous avons implémenté leur méthode. Pour l'intersection, nous avons utilisé le tableau de bits comme dans nos deux méthodes. Cela rend l'opération de l'intersection la même dans les trois méthodes.

Nous avons fait deux types d'expérimentations, le premier avec les mots exacts où toutes les positions sont considérées comme des positions d'erreur, le deuxième avec des mots qui contiennent exactement une seule erreur. Les résultats sont résumés dans le tableau 5.2.

		Anglais	**WikiTitle**
Mot 1 erreur	Amir et al	297,86	2840,50
	TRT_CI	*0,77*	*0,78*
	TRT_WNI	54,31	306,67
	TRT_CWNI	28,94	231,86
Mot exact	Amir et al	1516,27	8406,68
	TRT_CI	*2,52*	*1,65*
	TRT_WNI	185,23	701,80
	TRT_CWNI	131,51	548,02

TABLE 5.2 – Temps d'exécution en μs pour l'ensemble de données Anglais et WikiTitle.

Dans le tableau 5.2, on peut voir clairement que nos deux approches (TRT_WNI et TRT_CWNI) sont meilleures que la méthode de Amir et al. et cela car elles utilisent des heuristiques pour localiser les positions possibles de l'erreur, et évitent ainsi de tester toutes les positions de façon naïve. De plus, dans nos méthodes on évite de faire l'intersection avec la racine car cela signifie l'intersection avec tout le dictionnaire. Dans les résultats, on voit aussi que notre méthode TRT_CWNI améliore la méthode TRT_WNI, car on diminue le nombre de feuilles considérées dans l'intersection, lorsqu'on choisit les nœuds du premier niveau qui sortent du nœud de l'erreur, et ces nœuds mènent à des solutions possibles.

En utilisant des mots avec juste une seule erreur, notre méthode TRT_WNI dans le dictionnaire Anglais est 5 fois plus rapide que la méthode de Amir et al. et 9 fois plus rapide dans le dictionnaire WikiTitle. Avec les mots exacts où toutes les positions sont considérées comme une position d'erreur, TRT_WNI est 8 fois plus rapide dans le dictionnaire Anglais, et environ 12 fois plus rapide dans le dictionnaire WikiTitle. Il y a une différence dans le temps d'exécution lorsque nous testons avec les mots exacts, et les mots avec une seule erreur, parce que dans le mot exact nous devons vérifier tous les

nœuds dans le chemin du mot requête de la racine à la feuille, et ceci augmente le temps d'exécution.

Nous expérimentons avec des tailles différentes d'un dictionnaire, nous changeons le nombre de mots dans le dictionnaire pour voir l'influence de ce paramètre sur le temps d'exécution. Voir la figure 5.6.

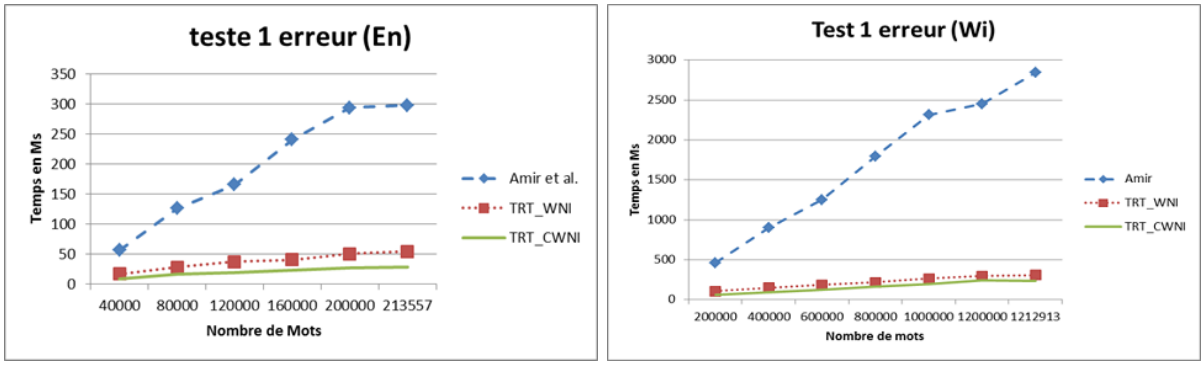

FIGURE 5.6 – Temps d'exécution en fonction du nombre de mots dans les dictionnaires Anglais et WikiTitle.

Dans la figure 5.6, nous présentons l'évaluation des trois méthodes en utilisant des mots avec juste une seule erreur. Il est clair que le temps d'exécution de la méthode de Amir et al. est très élevé comparé à nos méthodes. En plus, le temps d'exécution de leur méthode augmente rapidement lorsque la taille de dictionnaire s'agrandit. Cela pourrait s'expliquer par le fait que la méthode de Amir et al. vérifie toutes les positions dans le mot requête et donc cela revient à vérifier tous les nœuds dans le Trie depuis la racine jusqu'à feuille. De plus, l'intersection utilisée dépend du nombre de mots dans le dictionnaire et cela signifie qu'à chaque fois que l'ensemble de mots devient grand, le temps augmente lui aussi. Si nous avons beaucoup de mots qui ont le même préfixe, et beaucoup de mots qui ont le même suffixe, donc lorsque on vérifie le préfixe dans le Trie on aura un grand ensemble de mots du dictionnaire, et ce sera le même cas pour le suffixe, et donc nous aurons deux grands ensembles à vérifier. Le cardinal de chaque ensemble peut être proportionnel à la taille du dictionnaire.

Le temps d'exécution de notre méthode TRT_WNI augmente lentement. C'est parce qu'elle ne vérifie que les nœuds depuis la racine jusqu'à la position de l'erreur trouvée par la recherche exacte, comme nous l'avons expliqué dans la sous-section 5.2.3. En plus de cela, nous traitons les caractères qui sont à l'extrémité du mot requête directement avec la distance d'édition, et cela afin d'éviter l'intersection avec le nœud racine (voir la sous-section 5.2.5).

Dans ce qui suit nous donnons un tableau récapitulatif de toutes les méthodes testées dans ce travail (voir le tableau 5.3).

Méthode	Anglais	Wikititle
Amir et al	297,86	2840,50
Karch et al	8	34
Chegrane et Belazzougui	4,55	8,28
Méthode 1 : TRT_CI	0,77	0,78
Méthode 2 : TRT_WNI	54,31	306,67
Méthode 3 : TRT_CWNI	28,94	231,86

TABLE 5.3 – Comparaison des méthodes existantes avec nos trois méthodes sur les ensemble de données Anglais et WikiTitle (temps en μs).

5.6 L'application de notre méthode dans l'indexation de texte

On peut facilement adapter notre solution à la recherche des occurrences des mots ou la recherche des facteurs dans un texte.

La recherche des occurrences des mots :
Dans le chapitre précédent on a dit qu'un texte T peut se formuler avec $T' = l_1, l_2, ..., l_d$ où chaque mot peut avoir plusieurs occurrences $l_i = pos_1, pos_2, ..., pos_{nb}$.

Dans la structure de données Trie, dans chaque feuille, on ajoute un champ qui pointe vers une liste qui contient toutes les positions des mots dans le texte.

La recherche des facteurs dans un texte :
Il suffit de remplacer la structure de données Trie par un arbre des suffixes.

Il est clair qu'avec l'arbre des suffixes, on peut faire aussi la recherche de toutes les occurrences des mots, car un mot est aussi un facteur, mais on utilise un Trie avec les positions des mots afin d'optimiser l'espace mémoire, car l'arbre des suffixes stocke tous les suffixes de texte, alors que le Trie stocke les mots une seule fois.

5.7 Conclusion

Dans ce chapitre, nous avons proposé trois méthodes pour résoudre le problème de la recherche approchée dans un dictionnaire avec $k = 1$. Toutes les trois méthodes opèrent sur une structure de données bidirectionnelle (le Trie et le Trie inversé), pour exécuter une recherche exacte sur deux morceaux du motif.

Notre première méthode *TRT_ CI* effectue une intersection des caractères communs de transitions sortantes entre les deux nœuds où l'erreur est rencontrée, dans le Trie (préfixe) et dans le Trie inversé (suffixe) pour déterminer les chemins qui peuvent conduire à des solutions.

Dans la deuxième méthode *TRT_WNI*, nous effectuons une intersection entre deux ensembles de numéros de mots. Cette méthode améliore la méthode de Amir et al [83], grâce à l'utilisation d'heuristiques pour trouver les positions possibles de l'erreur, et pour éviter de faire une intersection avec la racine.

Dans la troisième méthode *TRT_CWNI*, nous combinons les deux méthodes *TRT_CI* et *TRT_WNI*, pour réduire le nombre de feuilles et réduire les ensembles sur lesquels l'intersection de numéros de mots sera effectuée.

Les résultats numériques montrent que notre première méthode *TRT_CI* surpasse toutes les autres implémentations testées jusqu'à ce jour en termes de temps d'exécution. De plus, cette performance est en proportion constante par rapport à la recherche exacte, indépendamment de la taille du dictionnaire.

Nos deux autres méthodes TRT_WNI et TRT_CWNI donnent des résultats meilleurs que la méthode de Amir et al. car elles utilisent des heuristiques pour localiser les positions possibles de l'erreur, et évitent de tester toutes les positions.

Notre méthode TRT_CWNI améliore la méthode TRT_WNI en diminuant le nombre de feuilles qui entrent dans le calcul de l'intersection lorsqu'on choisit les nœuds du premier niveau qui sortent depuis le nœud de l'erreur.

La méthode TRT_CWNI améliore la seconde méthode TRT_WNI, mais la première approche TRT_CI reste meilleure, parce qu'elle ne dépend que de la longueur du mot requête et de la taille de l'alphabet, mais la seconde dépend de tous les mots du dictionnaire.

Nos trois méthodes et la méthode de Amir et al, se basent sur le Trie et le Trie inversé. Notre méthode TRT_CI donne les meilleurs résultats et elle surpasse toutes les autres méthodes pratiques testées. Son fonctionnement dépend de la taille du mot requête et l'alphabet du dictionnaire, alors que les trois autres méthodes dépendent toutes du nombre total n qui est la taille de tout le dictionnaire.

Dans ce chapitre et le chapitre précédent, nous avons décrit deux solutions du problème général de la recherche approchée dans un dictionnaire/texte.

Les performances théoriques de la 1re solution (la recherche approchée avec hachage présentée dans le chapitre précédent) et son application pour un nombre d'erreurs $k \geq 2$ semble suggérer qu'elle est applicable avec d'aussi bonnes performances dans d'autres contextes. Nous avons vu qu'en pratique et pour $k = 1$ erreur la seconde solution (qui utilise le Trie et le Trie inversé proposée dans ce chapitre) est meilleure.

Par ailleurs, dans un problème de recherche approchée particulier, *l'auto-complétion*, où il s'agit de trouver tous les suffixes d'un préfixe contenant des erreurs, nous avons apporté une nouvelle solution meilleure, en pratique, que la solution 1. Cette solution fait l'objet du chapitre qui suit.

Chapitre 6

L'auto-complétion approchée dans une architecture Client-Serveur

6.1 Introduction

Les champs de saisie de texte sont utilisés pour entrer les données dans l'ordinateur, et ils ont été améliorés au cours des années. Aujourd'hui, ils sont équipés avec de nombreuses fonctionnalités afin d'aider l'utilisateur. Parmi l'une des plus intéressantes on trouve l'auto-complétion (on l'appelle aussi l'auto-suggestion).

L'auto-complétion est une technique qui facilite et accélère l'écriture, en proposant une liste de mots ou de phrases (les suggestions) qui complètent les quelques caractères tapés dans le champ de texte, dans un temps très court (généralement, quelques millisecondes). Généralement, la liste de suggestions s'affiche en dessous du champ de saisie.

La manière habituelle (mais pas obligatoire) dont ces mots sont choisis est telle que les quelques caractères tapés sont préfixe des mots[1]. Ensuite, prenant en compte cette liste, l'utilisateur peut choisir l'un des mots ou continuer de taper plus de caractères pour faire plus de filtrage, ce qui fait apparaître de nouvelles listes de mots.

À chaque fois que l'utilisateur tape un autre caractère, le nombre de résultats diminue pour devenir juste quelques mots dans la liste de suggestions. L'utilisateur trouve ce qu'il recherche, ou il continue à taper toute sa requête jusqu'à la fin.

L'auto-complétion est très utile et populaire dans plusieurs domaines et systèmes.

Dans le web, cette fonctionnalité est assurée par les navigateurs pour compléter les URLs. Elle est utilisée dans les pages HTML (côté client), avec des champs de saisie de texte spécifiques équipés par un système d'auto-complétion basé sur l'historique de champ récent [119, 120]. Dans le web côté serveur pour fournir une liste de suggestions et

1. On trouve aussi la complétion où les quelques caractères tapés ne sont pas juste préfixe mais un facteur de la phrase, une sous-chaine apparaissant dans n'importe quelle position.

l'envoyer au client comme par exemple les moteurs de recherche.

Sur les ordinateurs de bureau, cette fonctionnalité est intégrée dans de nombreuses applications, en utilisant par exemple la touche *Tab* dans un interpréteur de ligne de commande (Shell bash sous UNIX). Dans les éditeurs des codes sources pour les programmeurs, cette fonctionnalité est appelée *intellisense* [121].

Dans les appareils mobiles, taper avec précision est une tâche fastidieuse et l'écriture des utilisateurs a tendance à contenir des erreurs typographiques. L'auto-complétion approchée est une fonctionnalité très importante dans ce type d'environnements (les appareils mobiles simples et les Smartphones et les tablettes).

La figure 6.1, représente les différentes catégories où l'auto-complétion peut être utilisée, dans le web, le PC de bureau, le Smartphone.

FIGURE 6.1 – L'auto-complétion dans différentes applications et dispositifs (web, application de PC, Smartphone).

Considérant les bibliothèques de programmation de l'auto-complétion dans le Web, l'une des plus utilisée et qui permet aux programmeurs d'ajouter cette fonctionnalité à leurs champs de saisie est JQuery UI auto-complete (voir http://api.jqueryui.com/autocomplete/ [1]).

Cependant, et cela est la motivation principale de ce travail, la chaîne de caractères d'entrée (les quelques caractères) peut contenir des erreurs. L'erreur peut être soit une erreur de frappe, en particulier lors de la saisie rapide, soit une méconnaissance par l'utilisateur de l'orthographe correcte (nom de personne, nom du produit ... etc.). La bibliothèque JQuery UI auto-complete et les autres bibliothèques de l'auto-complétion ne permettent

1. Visité le : 02-07-2016.

pas de tolérer les erreurs dans la chaine de caractères entrée par l'utilisateur. Dans certains cas ce comportement peut induire l'utilisateur en erreur.

Pour résoudre ce problème, nous devons tolérer un certain nombre d'erreurs dans le préfixe tapé dans les champs de saisie de texte et dans la liste des suggestions afin d'obtenir une liste de complétion exacte et approchée (voire la figure 6.2).

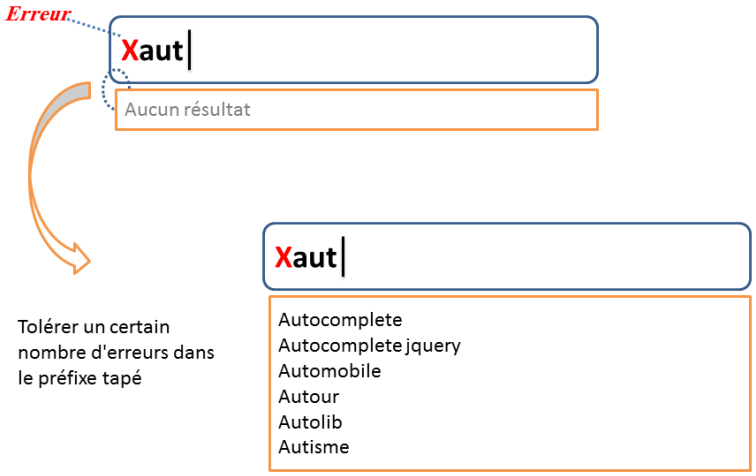

FIGURE 6.2 – L'auto-complétion approchée.

Lorsque l'utilisateur tape les quelque caractères (préfixe) de sa requête, le système lui affiche une liste de suggestions qui contient tous les mots qui sont lexicalement similaires au préfixe tapé. Dans cette liste, les mots qui ont exactement le même préfixe sont proposés en premier (l'auto-complète exacte), ensuite, les mots qui diffèrent d'un certain nombre d'erreurs k (l'auto-complète approchée). La mesure la plus utilisée pour déterminer la différence entre deux chaînes de caractères x et y est la *distance d'édition* [2].

Une conséquence évidente de la tolérance aux erreurs est que la liste de suggestions pourrait être très longue. Pour réduire le nombre de résultats, nous proposons une nouvelle bibliothèque d'auto-complétion nommée `appacolib` qui (a) limite le nombre d'erreurs possibles à au plus à une erreur, et (b) rapporte les k suggestions ayant les scores les plus élevés (dans un ordre décroissant). On appelle cette méthode top-k complétion (top-k suggestion). Le paramètre k est donné par l'utilisateur.

Répondre à des requêtes de préfixe avec erreur sur un dictionnaire est un sujet de recherche étudié depuis un certain temps, mais qui reste néanmoins très actif. Il existe plusieurs solutions algorithmiques qui résolvent ce problème de façon efficace. Cependant, très peu de ces solutions font face au problème dans une architecture client-serveur et proposent une solution pratique utilisable dans les systèmes actuels. Ceci est le but de ce

travail. Nous avons étudié le problème de l'auto-complétion approchée sous la mesure de la distance d'édition, et en particulier la possibilité d'améliorer la qualité des résultats en utilisant un système de classement selon l'importance (score) des mots dans le dictionnaire. Nous avons aussi étudié le problème de la redondance des résultats et nous avons proposé une solution efficace pour les éliminer.

En fait, la bibliothèque `appacolib` est un ensemble de bibliothèques pour différents langages pouvant être utilisés soit sur le serveur (C/C++) ou sur le client (JavaScript) ou sur les deux en même temps afin de répondre rapidement à des requêtes avec erreurs et faire l'auto-complétion en se basant sur un dictionnaire `UTF`-8.

Typiquement, si on a un dictionnaire très volumineux (plus de 3 Mo d'entrées), on peut utiliser la bibliothèque écrite en C/C++ dans le coté serveur avec un module FASTCGI pour fournir une liste de suggestions dans un format JSON, et renvoyer les résultats à travers des appels AJAX depuis notre bibliothèque version client ou n'importe quelle autre interface utilisée dans le coté client. Si le dictionnaire n'est pas très volumineux (par exemple moins de 3 Mo d'entrées), on peut effectuer toutes les opérations au niveau local (sur le navigateur). La bibliothèque version client est utilisée aussi pour envoyer des requêtes d'auto-complétion à des serveurs qui se basent sur une base de données ou un autre système permettant d'offrir une liste de suggestions au format JSON (par exemple un système écrit en PHP).

Lorsque l'on traite des données du monde réel, des problèmes pratiques apparaissent. Généralement, ils ne sont pas pris en compte dans le travail théorique. En particulier, la gestion efficace des fichiers `UTF`-8 en termes de temps et d'espace mémoire qui n'est, techniquement pas triviale.

Notre bibliothèque permet le traitement des dictionnaires `UTF`-8, quel que soit la langue utilisée (Arabe, Chinois, Latin...etc).

Le chapitre est organisé comme suit : Dans la section 6.2, nous détaillons les différentes structures de données utilisées pour faire une auto-complétion approchée efficace. Dans la section 6.3, nous expliquons les méthodes de recherche et comment les listes de suggestions sont générées. Dans la section 6.4 nous proposons différentes méthodes permettant de s'adapter au comportement de l'utilisateur dans le but de réduire les opérations faites dans la recherche, et donc de gagner en temps de calcul. Dans la section 6.5 nous proposons une méthode permettant de réduire le nombre des branches sortantes a tester dans chaque nœud valide, à l'aide d'une table de hachage qui permet de générer des caractères candidats à tester dans chaque branche. Dans la section 6.6 nous expliquons comment réduire le nombre de résultats, et mettre en avant les résultats qui satisfont le plus le besoin de l'utilisateur, en utilisant le système dit *Top-k*.

Dans la section 6.7, nous expliquons comment éliminer les résultats en double. Nous présenterons ensuite la stratégie client/serveur dans la section 6.8. Dans la section 6.9 nous présentons et nous discutons de la vitesse nécessaire pour que l'auto-complétion soit efficace. La section 6.10 est dédiée aux tests et aux expérimentations. Enfin la dernière section 6.11 conclut ce chapitre.

6.2 La structure de données

La plupart des algorithmes efficaces de recherche de préfixe approchée construisent d'abord un index sur le dictionnaire, puis utilisent cet index pour effectuer une recherche efficace pour chaque requête. Dans notre travail, nous construisons un index aussi, semblable à un Trie compact avec un système de classement de tous les mots du dictionnaire.

Dans notre travail, nous utilisons un Trie compact dans lequel nous optimisons l'espace mémoire ; chaque nœud qui est fils unique de son nœud parent est fusionné avec ce dernier. Un détail important de cette représentation est que chaque transition est en fait marqué par un facteur de l'un des mots dans le dictionnaire (et non pas par un caractère unique comme dans le cas du trie). Ainsi, pour réduire la mémoire nécessaire, en restant toujours efficace, nous codons une transition par son premier caractère, un pointeur dans l'index vers le début d'une occurrence de l'élément correspondant (facteur), et la longueur de la transition qui est donc la longueur du facteur. Cela implique que, dans cette approche, nous gardons à la fois, le dictionnaire des mots et la structure de données (le Trie compact) dans la mémoire centrale.

6.2.1 L'ordre lexicographique de dictionnaire

Dans notre travail, nous ordonnons le dictionnaire selon l'ordre lexicographique dans le but de pouvoir construire le tableau des longueurs des préfixes communs (LCP) entre toutes paires de mots successifs, permettant ainsi d'accélérer la construction du Trie.

L'opération de tri du dictionnaire prend un temps relativement grand, spécialement si la taille de dictionnaire est grande. Comme en généralement, les dictionnaires se présentent sous forme de fichiers (par exemple le dictionnaire Anglais), il est préférable d'ordonner le fichier avant de l'utiliser dans la construction de système d'auto-complétion, pour éviter de l'ordonner à chaque fois qu'on veut l'utiliser.

6.2.2 La construction de Trie à l'aide de tableau LCP

Pour construire notre Trie compact d'une façon efficace, nous utilisons comme indiqué plus haut une autre structure de données appelée *tableau des plus longs préfixes communs*. Le tableau LCP stocke les longueurs des plus longs préfixes communs entre chaque paire de mots consécutifs du dictionnaire, tel que la première case du tableau contient la

longueur du plus long préfixe commun entre le premier et le deuxième mot du dictionnaire.

Exemple : soit le dictionnaire suivant $D = \{AbCdeAa, AbCdeBbbOo, AbCdeBbbSss, AbEfg\}$. On prend les mots deux par deux, et on calcule leurs plus long préfixes communs.

1. $\{AbCdeAa, AbCdeBbbOo\}$, le plus long préfixe commun est $AbCde$, il est de longueur 5.
2. $\{AbCdeBbbOo, AbCdeBbbSss\}$, le plus long préfixe commun est $AbCdeBbb$, il est de longueur 8.
3. $\{AbCdeBbbbSss, AbEfg\}$, le plus long préfixe commun est Ab , il est de longueur 2.

À la fin, notre tableau LCP est comme suit : $\texttt{Tab\_LCP} = [5, 8, 2]$.

La profondeur d'un nœud nd est le nombre de caractères lus sur le chemin depuis la racine jusqu'au ce nœud nd. La profondeur de chaque nœud interne nd est en fait la longueur du plus long préfixe commun entre tous les mots pointé par les feuilles se trouvant sous ce nœud. La profondeur d'un mot pour un nœud feuille est la longueur du mot pointé par le nœud. Dans la construction du Trie, pour chaque mot, on crée un nœud feuille, et pour chaque préfixe commun, on crée un nœud interne.

En supposant que l'on a déjà calculé et construit le tableau LCP (`Tab_LCP`), les étapes de construction du Trie compact à l'aide de tableau LCP sont comme suit :

Algorithme de construction du Trie compact
Entrée : Un fichier trié contenant les mots du dictionnaire. Le tableau LCP.
Sortie : La structure de données Trie compact.

1. Créer le nœud racine.
2. Ajouter le premier mot au Trie, donc créer un nœud feuille pour le premier mot.
3. Pour tous les mots restants $i \in [2..n]$:
   (a) Si le mot x précédent $i-1$ (x_{i-1}) est un préfixe propre dans le mot x actuel i (x_i). Alors on teste si le plus long préfixe commun entre le mot actuel et le mot précédent ($\texttt{Tab\_LCP}[\texttt{i} - 1]$) est supérieur ou égal à la longueur du mot précédent (la profondeur `Prof` du nœud feuille $\texttt{nd}_\texttt{f}$ pointant vers le mot précédent x_{i-1}, $(\texttt{Prof}(\texttt{nd}_{\texttt{f}(\texttt{x}_{\texttt{i}-1})}))$.
   - Si les deux mots ont la même longueur, alors on ne fait rien (le dictionnaire ne contient pas deux copies d'un même mot).
   - Sinon, ajouter un nouveau nœud feuille $\texttt{nd}_{f(x_i)}$ qui sort du nœud père de mot précédent $\texttt{nd}_{p(x_{i-1})}$. Ajouter les caractères restants du mot actuel à la transition entre le nœud père et le nouveau nœud $\texttt{Tr}(\texttt{nd}_{p(x_{i-1})}, \texttt{nd}_{f(x_i)})$. On

ajoute les caractères qui ne sont pas dans le préfixe commun représenté par le nœud père, le nombre de caractères à ajouter est égal à la profondeur du nœud feuille (qui est la taille de mot) moins la profondeur de nœud père ($\texttt{Nb\_char\_add} = \texttt{Prof}(\texttt{nd}_f) - \texttt{Prof}(\texttt{nd}_p)$).

(b) Sinon, le mot précédent (x_{i-1}) n'est pas un préfixe propre du mot actuel (la taille du préfixe commun est différente de la longueur du mot précédent) donc :

- Depuis le nœud feuille pointant vers le mot précédent $\texttt{nd}_{f(x_{i-1})}$, remonter dans ses nœuds ancêtres jusqu'à arriver au nœud $\texttt{nd}_{p(\texttt{LCP}(x_{i-1},x_i))}$ qui représente le préfixe commun entre le mot précédent et le mot actuel. Pour cela, il suffit juste de comparer la taille de préfixe commun ($\texttt{Tab\_LCP}[i-1]$) avec la profondeur des nœuds internes ($\texttt{Prof}(\texttt{nd}_{p_i})$), tant que $\texttt{Tab\_LCP}[i-1] < \texttt{Prof}(\texttt{nd}_p)$, alors remonter au nœud parent et ainsi de suite, jusqu'à arriver au nœud $\texttt{nd}_{p(\texttt{LCP}(x_{i-1},x_i))}$. Donc on a $\texttt{Prof}(\texttt{nd}_{p_1}) < \texttt{Prof}(\texttt{nd}_{p_2}) < ... < \texttt{Prof}(\texttt{nd}_f)$, et $\texttt{nd}_{p_1} = \texttt{nd}_{p(\texttt{LCP}(x_{i-1},x_i))}$. Le nœud $\texttt{nd}_{p_i}$ est le père du nœud $\texttt{nd}_{p_{i+1}}$, le nœud nd_f est le dernier nœud dans le chemin depuis nd_{p_1}.

- Entre le nœud $\texttt{nd}_{p_1}$ et son nœud fils $\texttt{nd}_{p_2}$, créer un nœud intermédiaire $\texttt{nd}_{\texttt{int}}$. L'ordre des nœuds devient alors comme suit : $\texttt{nd}_{p_1}$, $\texttt{nd}_{\texttt{int}}$, $\texttt{nd}_{p_2}$. La chaîne de caractères qui a été sur la transition $\texttt{Tr}(\texttt{nd}_{p_1}, \texttt{nd}_{p_2})$ va être découpée entre les deux nouvelles transitions $\texttt{Tr}(\texttt{nd}_{p_1}, \texttt{nd}_{\texttt{int}})$, $\texttt{Tr}(\texttt{nd}_{\texttt{int}}, \texttt{nd}_{p_2})$, le nombre de caractères à mettre dans la transition $\texttt{Tr}(\texttt{nd}_{p_1}, \texttt{nd}_{\texttt{int}})$ est la profondeur de $\texttt{nd}_{\texttt{int}}$ moins la profondeur de nœud $\texttt{nd}_{p_1}$, ($\texttt{Nb\_char\_add} = \texttt{Prof}(\texttt{nd}_{\texttt{int}}) - \texttt{Prof}(\texttt{nd}_{p_1})$).

- Insérer un nouveau nœud feuille $\texttt{nd}_f$ qui sort de ce nœud intermédiaire $\texttt{nd}_{\texttt{int}}$ et le faire pointer le mot actuel à insérer. Ensuite, ajouter les caractères qui ne sont pas dans le préfixe commun à la nouvelle transition $\texttt{Tr}(\texttt{nd}_{\texttt{int}}, \texttt{nd}_f)$. Le nombre de caractères à ajouter est égal à la profondeur du nœud feuille (qui est la taille du mot) moins la profondeur du nœud père qui est le nœud intermédiaire : $\texttt{Nb\_char\_add} = \texttt{Prof}(\texttt{nd}_f) - \texttt{Prof}(\texttt{nd}_{\texttt{int}})$.

Exemple :
Soit le dictionnaire suivant $D = \{\texttt{AbCdeAa}, \texttt{AbCdeBbbOo}, \texttt{AbCdeBbbSss}, \texttt{AbEfg}\}$. Le tableau LCP est comme suit : $\texttt{Tab\_LCP} = [5, 8, 2]$. On construit notre Trie étape par étape avec les explications :

1. La 1re étape est de créer le nœud racine $\texttt{nd}_0$.

2. Le 1er mot `AbCdeAa` : insérer le 1er nœud feuille $\texttt{nd}_1$ qui est relier avec la transition qui sort de la racine. Et on met le mot `AbCdeAa` sur la transition ($\texttt{Tr}(\texttt{nd}_0, \texttt{nd}_1)$).
3. Le 2e mot `AbCdeBbbOo` : d'après le tableau LCP, il y a un préfixe commun d'une longueur 5 avec le mot précédent, et le mot précédent n'est pas un préfixe propre de mot actuel. La transition précédente $\texttt{Tr}(\texttt{nd}_0, \texttt{nd}_1)$ va être découpée, pour cela, on crée un nœud intermédiaire $\texttt{nd}_2$ entre $\texttt{nd}_0$ et $\texttt{nd}_1$, de telle sorte que la transition $\texttt{Tr}(\texttt{nd}_0, \texttt{nd}_2)$ contienne le préfixe commun `AbCde`, le reste du mot précédent est dans la nouvelle transition $\texttt{Tr}(\texttt{nd}_2, \texttt{nd}_1)$ donc `Aa`. Ensuite, depuis $\texttt{nd}_2$ on ajoute un nœud fils $\texttt{nd}_3$, de telle sorte que la nouvelle transition $\texttt{Tr}(\texttt{nd}_2, \texttt{nd}_3)$ va contenir le reste du mot actuel donc `BbbOo`.
4. Le 3e mot `AbCdeBbbSss` : récupérer la longueur du préfixe commun depuis le tableau LCP, donc 8. Depuis le nœud feuille du dernier mot insérer ($\texttt{nd}_3$), on retrouve le nœud ancêtre le plus profond qui à une profondeur inférieure ou égale à la longueur de préfixe commun 8, on trouve le nœud $\texttt{nd}_2$. On crée alors un nœud intermédiaire $\texttt{nd}_4$ entre la transition $\texttt{Tr}(\texttt{nd}_2, \texttt{nd}_3)$ et on met `Bbb` dans la nouvelle Transition $\texttt{Tr}(\texttt{nd}_2, \texttt{nd}_4)$ car le nœud $\texttt{nd}_2$ a une profondeur 5, et le nouveau nœud a une profondeur 8, donc on ajoute juste 3 caractères à la nouvelle transition. La transition $\texttt{Tr}(\texttt{nd}_4, \texttt{nd}_3)$ va contenir `Oo`. Ensuite, on crée un nouveau nœud feuille $\texttt{nd}_5$ qui sort du $\texttt{nd}_4$, et on met `Sss` sur la nouvelle transition $\texttt{Tr}(\texttt{nd}_4, \texttt{nd}_5)$.
5. Le 4e mot `AbEfg` : d'après le tableau LCP, on a un préfixe commun d'une longueur 2. Alors depuis le nœud feuille de mot précédent $\texttt{nd}_5$, on remonte pour trouver le bon père, on remonte jusqu'au nœud racine $\texttt{nd}_0$. On crée un nouveau nœud intermédiaire $\texttt{nd}_6$ entre la transition $\texttt{Tr}(\texttt{nd}_0, \texttt{nd}_2)$, la nouvelle transition $\texttt{Tr}(\texttt{nd}_0, \texttt{nd}_6)$ va représenter le préfixe commun `aB`. Ensuite, on crée un nouveau nœud feuille $\texttt{nd}_7$ qui sort de $\texttt{nd}_6$, et la transition $\texttt{Tr}(\texttt{nd}_6, \texttt{nd}_7)$ va contenir la chaîne de caractères `Efg`.

Le résultat de la construction de la structure de données est illustré dans la figure 6.3.

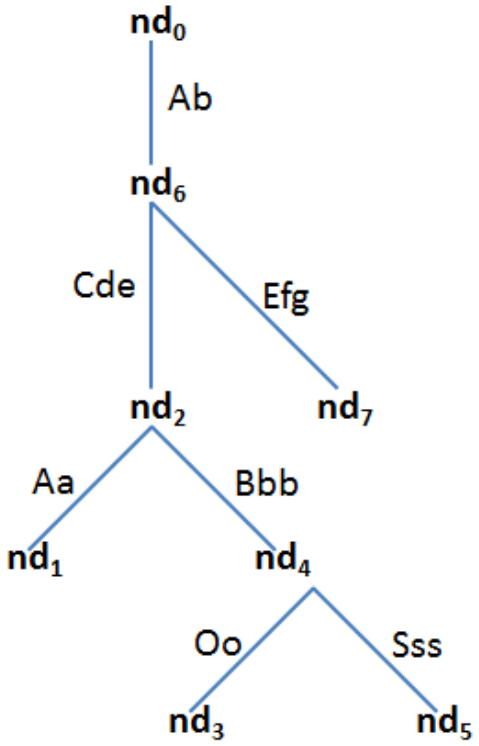

FIGURE 6.3 – Le Trie compact construit avec le tableau LCP.

6.2.3 Réduire la mémoire nécessaire au Trie tout en restant efficace en temps de calcul

Dans cette approche, nous gardons à la fois, le dictionnaire des mots et la structure de données (le Trie compact) dans la mémoire centrale.

Afin de réduire la taille du Trie, toutes les transitions vont être encodées seulement avec leur 1er caractère, et on leur ajoute 1) la longueur de la transition, donc la longueur du facteur, 2) un pointeur vers le début d'une occurrence du facteur dans le dictionnaire des mots. Donc chaque transition, au lieu de contenir un facteur (d'un mot ou plusieurs mots partageant cette transition), elle ne contiendra que 3 éléments, le 1er caractère du facteur, un pointeur vers le dictionnaire et la longueur de la transition.

La réduction de la mémoire du Trie compact avec l'exemple précédent (de la figure 6.3) est illustrée dans la figure 6.4.

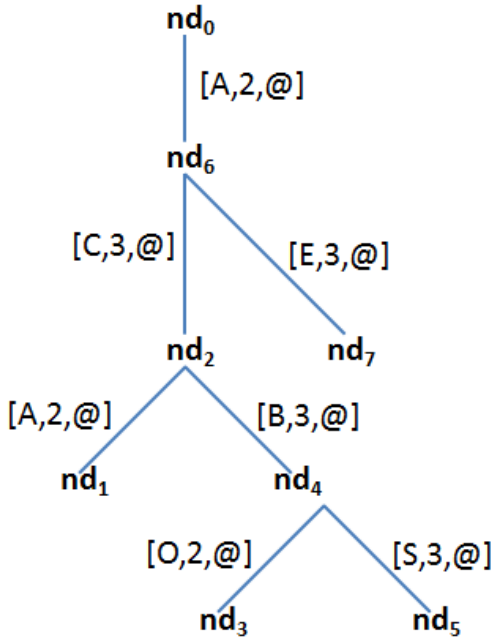

FIGURE 6.4 – Réduire l'espace mémoire occupé par le Trie. Chaque transition est codée par son 1er caractère, la longueur de la transition (le facteur), et un pointeur vers le début d'une occurrence du facteur dans le dictionnaire.

Pour maintenir notre structure de données aussi petite que possible, les caractères UFT-8 et les nombres entiers sont codés par bits [122], les pointeurs sont également codés comme des entiers.

6.2.4 Ajouter les scores des mots au Trie compact

Chaque mot est associé un score statique, dans la construction du Trie compact, lorsqu'on atteint une feuille, on stocke le score du mot associé dans le nœud feuille pointant vers le mot. Après l'insertion de tous les mots dans le Trie, et afin de supporter une complétion Top-k d'une façon efficace et rapide, pour chaque nœud interne, on garde de manière récursive le score maximal parmi ses enfants, jusqu'à ce qu'on arrive à la racine.

Généralement, les scores sont stockés à la fin de chaque mot, donc on doit les extraire pour les insérer dans le Trie.

Exemple : soit le dictionnaire précédent avec des scores
$D = \{\texttt{AbCdeAa}\#55, \texttt{AbCdeBbbOo}\#9, \texttt{AbCdeBbbSss}\#11, \texttt{AbEfg}\#33\}$. Le résultat d'ajouter les scores au Trie est illustré dans la figure 6.5 (dans cette figure, nous avons utilisé la version du Trie compact sans l'étape de réduction de l'espace mémoire) :

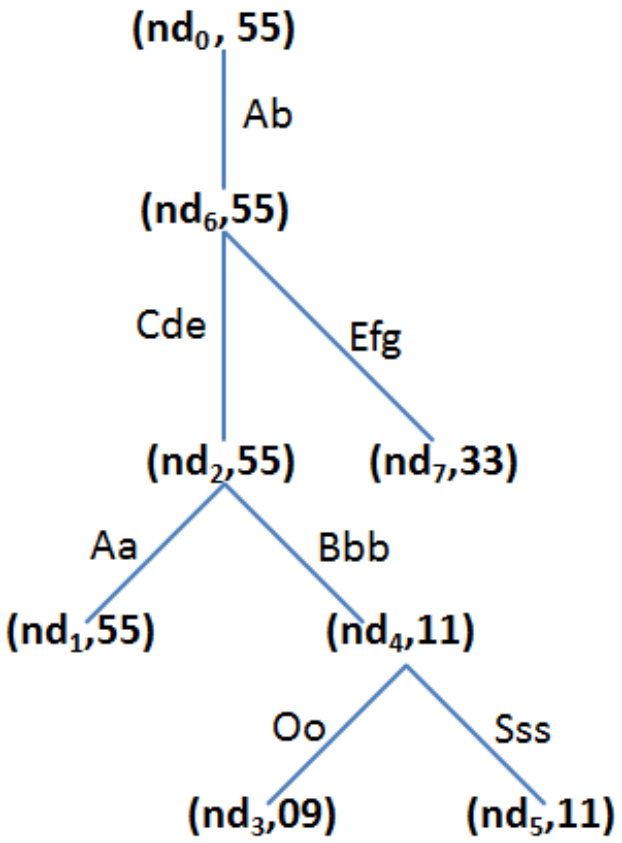

FIGURE 6.5 – Ajouter les scores des mots dans le Trie compact.

6.2.5 Préparer la file de priorité et le tableau de hachage

Nous proposons d'utiliser deux structures de données supplémentaires pour supporter les requêtes Top-k d'une façon efficace et rapide, et aussi pour supprimer les résultats en double de la liste finale.

La file de priorité est une structure de données qui permet de garder une liste d'éléments dans l'ordre, et dont l'élément avec la plus grande priorité est stocké en première position. Cette structure de données est utilisée pour permettre a trouver les résultats Top-k afin de les retourner à l'utilisateur. Pour implémenter notre file de priorité, nous avons utilisé un tas classique implémenté avec tableau.

Nous utilisons un tableau de hachage avec sondage linéaire, afin de vérifier si le résultat existe dans la liste des solutions dans le but d'éliminer les résultats en double.

6.2.6 Résumé de toutes les étapes de construction de la structure de données

Algorithme de construction de la structure de données (les grandes lignes)

Entrée : fichier contenant les mots du dictionnaire.
Sortie : structure de données pour l'auto-complétion.

1. Ordonner le dictionnaire dans l'ordre lexicographique. Pour plus de détails voir la sous-section 6.2.1.
2. Construire le tableau des plus longs préfixes communs (LCP array) de tous les mots du dictionnaire. Pour plus de détails voir la sous-section 6.2.2.
3. Construire le Trie compact en se basant sur le tableau LCP pour accélérer la construction. Pour plus de détails voir la sous-section 6.2.2.
4. Ajouter les scores des mots au Trie compact. Pour plus de détails voir la sous-section 6.2.4.
5. Préparer la file de priorité. Pour plus de détails voir la sous-section 6.2.5.
6. Préparer un tableau de hachage avec sondage linéaire. Pour plus de détails voir la sous-section 6.2.5.

6.3 La méthode de recherche

On commence d'abord par expliquer l'algorithme qui permet de rechercher les résultats et de proposer la liste de suggestions. Dans notre méthode, le Trie est utilisé afin de répondre aux requêtes de manière approchée, en utilisant un classement transversal de certains nœuds du Trie. Soit q le mot requête avec $|q| = m$ et k le nombre des résultats demandés dans la liste de complétion.

6.3.1 Trouver les nœuds valides

Étant donné un mot q, on appelle *locus* la position dans le Trie compact où la recherche exacte de q s'est arrêtée. Cette position peut être sur un nœud, ou au milieu d'une arête (transition). La recherche exacte s'arrête dans la position nommée *locus* car soit on est arrivé à la profondeur $|q|$, donc on a trouvé le mot requête complet (la profondeur $|q|$ est le nombre total des caractères du mot requête q), ou on a obtenu une erreur avant de terminer la recherche et de trouver le mot entier q.

On appelle *nœud locus* le nœud qui représente la position *locus*. Si la position *locus* se termine dans un nœud, ce dernier est alors, le *nœud locus* ; sinon, la position *locus* se termine au milieu d'une arête et donc on considère le nœud à destination de cette arête

comme le *nœud locus*.

On appelle un nœud *nœud valide*, le nœud qui mène à des solutions exactes ou des solutions approchées. Pour les solutions exactes, nous n'avons qu'un seul nœud, par contre pour les solutions approchées, on peut avoir plusieurs nœuds possibles.

Dans ce qui suit, on donne l'algorithme pour trouver tous les nœuds valides. On utilise l'algorithme naïf de la recherche approchée dans un Trie.

Algorithme_tous_les_nœuds_valides
Entrée : la requête (préfixe) q.
Sortie : une liste de nœuds valides.

a. Trouver le *nœud locus* `nd` de q.
b. Si le *nœud locus* `nd` est à une profondeur d'au moins $|q|$, cela signifie que l'on a une solution exacte, alors ajouter ce nœud à la liste des nœuds valides.
c. Pour chaque nœud sur le chemin menant au *nœud locus* trouvé à l'étape a. :
   - Prendre un nœud comme *nœud locus*.
   - Faire une opération d'édition sur toutes les transitons sortantes (sauf pour celle qui est sur le chemin qui mène au *nœud locus* trouvé à l'étape a.).
   - Continuer une recherche exacte pour le suffixe restant de la requête dans le sous-arbre descendant de ce nouveau *nœud locus*.
   - Si une occurrence approchée de q est trouvée, alors ajouter le *nœud locus* de cette solution comme un nœud valide à la liste des solutions.

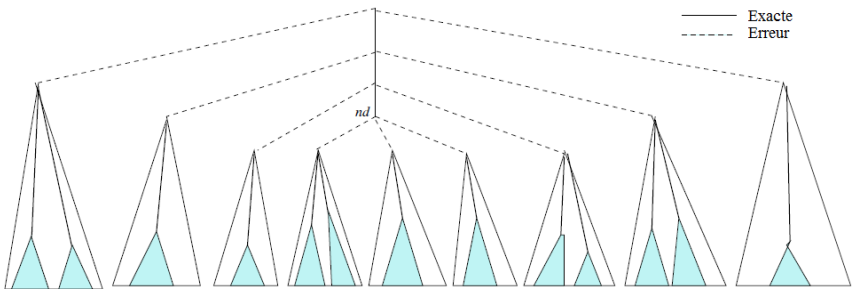

FIGURE 6.6 – La recherche avec une distance d'édition à 1-erreur dans un Trie compact.

La figure 6.6 représente la recherche des nœuds valides qui mènent à des solutions exacte et approchées dans un Trie compact. Le chemin gras représente une correspondance exacte, alors que les chemins avec des points correspondent à celles avec 1-erreur d'édition (insertion, suppression, substitution).

Dans la section 6.5, on détaille une méthode qui permet de faire une recherche approchée d'une manière efficace (sans explorer tout l'espace de Trie).

6.3.2 Obtenir une liste de résultats à suggérer

On se basant sur la liste des nœuds valides trouvés dans l'étape précédente, on calcule l'ensemble des k résultats. Pour cela, on utilise les scores enregistrés dans chaque nœud interne du Trie pour trouver quel chemin on doit prendre afin d'arriver aux résultats ayant les plus hauts scores. On utilise une file de priorité afin de trouver les k meilleurs résultats et non juste le meilleur résultat. L'algorithme permettant de calculer l'ensemble de la liste de suggestions Top-k, une fois que l'ensemble des nœuds valides a été calculé est le suivant :

Algorithme_de_liste_de_suggestions
Entrée : liste des nœuds valides.
Sortie : liste des Top-k complétions.

a. Il y a une solution exacte : on prend tous les nœuds destinations des transitions sortantes depuis le nœud valide représentant une solution exacte, et on les ajoute à la file de priorité.

b. Obtenir le nœud avec le plus grand score (celui en haut de la file de priorité). Si ce nœud représente une feuille, alors on ajoute le mot correspondant à la liste des résultats. Sinon on insère toutes les nœuds destinations de ses transitions (ces fils) dans la file de priorité.

c. Faire le traitement de l'étape b. jusqu'à obtenir k mots dans la liste de suggestions, ou jusqu'à ce que la file de priorité soit vide.

d. Si la file de priorité est vide avant d'avoir k mots dans la liste des suggestions, ou si l'on n'a aucune solution exacte, alors : ajouter tous les nœuds valides restants qui représentent des solutions approchées avec leurs scores dans la file de priorité

e. Faire le traitement de l'étape c.

Dans cet algorithme qui trouve les k résultats de complétions, on commence par le nœud de la solution exacte afin de s'assurer que les premiers résultats dans la liste de suggestions sont des complétions exactes, ensuite, on ajoute les solutions approchées, car on doit d'abord présenter les solutions exactes (si elles existent) et ensuite seulement les solutions approchées.

À cette fin, dans la phase qui trouve les nœuds valides (voir la sous-section 6.3.1), lorsqu'on trouve une solution exacte, on l'ajoute dans la liste des nœuds valides dans le champ réservé à la solution exacte (où on marque juste qu'il s'agit d'une solution exacte) afin de le distinguer des autres nœuds. Ensuite, on ajoute les autres nœuds qui représentent des solutions approchées.

6.4 L'auto-complétion et la saisie de l'utilisateur

Afin de ne pas recalculer tous les *nœuds locus*[1] à partir du début, à chaque fois qu'un nouveau caractère est tapé, on stocke l'ensemble des positions des *nœuds locus* après chaque nouveau caractère, et on continue la recherche dans les sous-arbres dont ils sont racines.

Une analyse des comportements d'un utilisateur en train de taper une requête, permet de les diviser en trois catégories :

1. L'utilisateur tape sa requête pour la première fois, ou alors il tape une requête totalement différente de celle qui la précède.
2. Lorsque l'utilisateur tape sa requête, le système d'auto-complétion en temps réel lui propose des listes de suggestions. Et l'utilisateur continue à taper sa requête donc il ajoute d'autres caractères à la fin si aucune des suggestions ne lui convient.

   Cette deuxième catégorie est la manière habituelle de l'interaction de l'utilisateur avec un système d'auto-complétion.
3. Lorsque l'utilisateur écrit sa requête, ensuite, il fait une modification par exemple une suppression à la fin de la chaîne pour qu'il retape d'autres caractères. En général, la modification peut-être : une suppression ou un ajout à n'importe quelle position dans la requête, à la fin, au milieu ou au début. Généralement, cette troisième catégorie n'est pas la manière habituelle que l'utilisateur utilise avec le système d'auto-complétion.

Dans ce qui suit, nous détaillons chaque catégorie et nous proposons une méthode permettant de s'adapter à l'utilisateur afin d'essayer de réduire les opérations faites dans la recherche et donc de gagner en temps de calcul.

6.4.1 Chercher depuis la racine

La première méthode est la plus simple. C'est la méthode habituellement utilisée pour faire la recherche dans un Trie : elle consiste à faire la recherche à chaque fois en commençant le parcours du Trie depuis la racine. On applique les algorithmes **Tous_nœuds_valides** et **Liste_des_suggestions**. On appelle cette méthode *search_from_root*. Quelle que soit la requête tapée, on refait à chaque fois la recherche depuis la racine.

6.4.2 Ajout à la fin

Le système d'auto-complétion est interactif, lorsqu'on tape une requête dans un champ de texte, les suggestions sortent au fur et à mesure qu'on tape les caractères, et cela dépend de **la durée d'attente entre chaque paire de requêtes consécutives**.

1. Voir la sous-section 6.3.1.

Exemple : on tape AB donc on a une première liste de résultats pour le préfixe AB. Ensuite, on continue à taper et on ajoute CD, le mot requête devient $ABCD$, donc on a une autre liste de suggestion pour le préfixe $ABCD$. De même, à chaque fois qu'on ajoute de nouveaux caractères, on aura une nouvelle liste.

On remarque que chaque requête est un préfixe de la requête suivante. Généralement entre chaque paire de requêtes consécutives traitées par le système, il y a un temps d'attente prédéfini t. Pour ne pas refaire la recherche à chaque fois depuis le début (la racine) on procède comme suit :

Sur les quelques premières frappes de la requête : on applique les algorithmes **Tous nœuds valides** et **Liste des suggestions**.

Quand des caractères sont ajoutés à la fin (en fonction du temps d'attente t, on peut avoir un ou plusieurs caractères), on ne commence pas la recherche à partir de la racine du Trie, mais plutôt, de la position du *nœud locus* dans la dernière recherche exacte (i.e. dans la requête précédente). En d'autres termes, le *nœud locus* va être considéré comme une nouvelle racine et on continuera la recherche à partir de cette position, et on refait exactement les mêmes étapes dans **Tous_nœuds_valides** et **Liste_des_suggestions**.

En général, on considère qu'on a deux mots requêtes différents. On commence la recherche à partir de la position du *nœud locus* du dernier mot requête. Si c'est un préfixe du mot requête courant. Autrement, on commence la recherche à partir de la racine.

Cette manière de faire est la manière habituelle de taper une requête par l'utilisateur (caractère après caractère). Donc cette méthode est adaptée pour le besoin de proposer des listes de complétions d'une façon rapide au fur et à mesure que l'utilisateur tape les caractères de sa requête : à chaque fois qu'il ajoute des caractères, on affiche une liste d'une façon rapide et on continue la recherche depuis le nœud auquel a abouti la recherche précédente.

On nomme cette méthode *search_end_node*.

6.4.3 Une modification dans la requête

En tapant sa requête, l'utilisateur peut de temps en temps faire certaines modifications telles que la suppression, l'insertion au milieu, un ajout à la fin... Etc. On note ici qu'il y a des parties de la requête qui restent inchangées.

Pour ne pas recommencer à chaque fois la recherche depuis la racine, on enregistre tous les nœuds sur le chemin menant de la racine au *nœud locus* dans un tableau T de la 1^{re} requête. La taille du tableau T est la longueur du plus long chemin dans le Trie. On stocke chaque nœud dans la case qui correspond à sa profondeur. Le nœud `nd` est stocké

dans la case numéro i où la profondeur de `nd` est i, cela signifie qu'on à i caractères depuis la racine jusqu'à ce nœud `nd`.

S'il y a un préfixe commun entre la requête modifiée (la 2^{e} requête) et celle qui la précède, alors on trouve le nœud correspondant à ce préfixe commun à partir du tableau précédemment sauvegardé. Pour cela, il suffit juste d'aller directement à la case numéro (profondeur du préfixe commun). Le nœud récupéré est considéré comme étant la racine, ensuite, on continue la recherche en faisant les mêmes étapes des algorithmes **Tous_nœuds_valides** et **Liste_des_suggestions**.

En général, s'il y a un préfixe commun entre deux mots requêtes, on commence la recherche à partir du nœud dont la profondeur est égale à la longueur du plus long préfixe commun.

Cette 2^{e} méthode a une mauvaise complexité temporelle : D'un premier point de vue, on dit que cette méthode permet de gagner un peu de temps de calcul puisqu'on ne recommence pas la recherche depuis le début, il suffit juste de trouver le bon nœud et de le considérer comme une nouvelle racine pour continuer la recherche.

Mais en réalité, pour que cette méthode puisse s'exécuter, on doit faire des modifications ou une ré-initialisation dans le tableau (Tab_node) qui sauvegarde les nœuds sur le chemin de la recherche après chaque nouveau mot requête.

Au début, toutes les cases du tableau sont initialisées à -1. Avec le 1er mot requête, on sauvegarde les nœuds qui correspondent au chemin de la recherche. Dans le 2^{e} mot requête : si on a un préfixe commun, on réinitialise la partie droite avec -1 à partir de la position du préfixe commun. Ensuite, on continue la recherche de suffixe et on sauvegarde les nœuds traversés pour ce dernier. Si on a un mot complètement différent on doit ré-initialiser tout le tableau pour l'utiliser de nouveau avec ce nouveau mot.

Les étapes de ré-initialisation et modification du tableau ajoutent un temps supplémentaire au temps de recherche.

Pour voir si cette méthode est utile ou non, on doit calculer la complexité de ces opérations. Pour le 1er mot requête, on a $2m$. La recherche de mot requête dans le Trie est en $O(m)$, et l'ajout de tous les nœuds au tableau Tab_node est aussi en $O(m)$.

Pour le 2^{e} mot requête, on a $2m$. Calculer le plus grand préfixe commun et ré-initialiser la partie droite de tableau Tab_node donnent $O(m)$. Continuer à chercher le suffixe qui reste et stocker les nœuds trouvés de ce suffixe dans le tableau s'exécutent en $O(m)$.

Cela donne entre le 1er et le 2^{e} mot requête une complexité de $4m$.

Au final, d'après le calcul de la complexité, cette méthode n'est pas efficace, car elle prend beaucoup plus d'opérations que de faire une simple recherche depuis la racine à chaque fois. Faire une simple recherche depuis la racine pour 2 requêtes est en $2m$ par contre cette méthode est en $4m$, donc elle prend un temps double.

On appelle cette méthode *search_ tab_ node*.

6.5 Réduire le nombre des branches sortantes testées dans le Trie

Afin de réduire le nombre des branches sortantes à tester dans chaque nœud valide, nous avons effectué les expérimentations sur la version côté serveur à l'aide d'une table de hachage, qui permet de générer des caractères candidats à tester dans chaque branche.

On stocke un dictionnaire de listes de substitutions tel que défini dans [16, 15], voir la sous-section *dictionnaire des listes de substitutions* 4.2.2 dans le chapitre 4. Ce dictionnaire est construit seulement sur les préfixes ayant une longueur limitée. Pour chaque longueur, on stocke les caractères candidats pour toutes les positions.

Un dictionnaire de listes de substitutions, pour une longueur d stocke une liste de caractères c associée à des motifs de substitutions $p\phi q$, de telle sorte que pcq est une chaîne de caractères de longueur d qui est elle-même le préfixe de certaines chaînes de caractères dans le dictionnaire.

Le dictionnaire de listes de substitutions permet de faire la recherche approchée comme suit : étant donné un mot requête de la forme d'un motif de substitution $p\phi q$, le dictionnaire de listes de substitutions retourne un ensemble des caractères c (une liste) tels que s'ils sont substitués à la place de ϕ, ils génèrent des préfixes pour les chaînes de caractères qui sont dans le dictionnaire.

Dans notre cas, nous avons un paramètre D (par exemple $D = 6$), de telle sorte qu'on stocke des listes de substitutions pour tous les préfixes de la longueur $d \leq D$.

Pour chaque préfixe commun du dictionnaire de longueur D, on prend tous ces préfixes de longueur de 2 (longueur minimum d'un mot) à D, pour générer des listes de substitutions. Ainsi, quelle que soit la longueur du préfixe requête tapé (de 2 à D) on peut appliquer cette méthode de recherche approchée.

Nous avons implémenté trois stratégies. étant donné un mot requête $x[1..m]$ avec $m \leq D$.

6.5.1 La 1re stratégie : construire les mots candidats et les vérifier dans le Trie.

Dans la première stratégie, on applique la méthode expliquée dans le chapitre 4 (la recherche approchée avec hachage), qui trouve les mots candidats qui peuvent être une solution approchée, ensuite, on les vérifie dans le dictionnaire exact.

Pour chaque motif de substitution $p\phi q$, on cherche les caractères candidats depuis la liste de substitutions, et on les remplace à la place de ϕ pour obtenir un mot pcq qui a une très forte chance d'être une solution approchée. Pour vérifier si le mot pcq représente une solution approchée, on le vérifier dans le Trie, en faisant une simple recherche exacte, donc on commence depuis la racine jusqu'à trouver le dernier caractère du motif pcq.

L'algorithme détaillé de cette méthode est comme suit :

Algorithme_tous_les_nœuds_valides_1err_SL
Entrée : le préfixe requête x.
Sortie : une liste de nœuds valides.

a. Trouver le *nœud locus nd* de x.

b. Si le *nœud locus nd* est à une profondeur d'au moins $|x|$, cela signifie qu'on a une solution exacte, donc ajouter ce nœud à la liste des nœuds valides.

c. Pour chaque nœud sur le chemin menant au *nœud locus* trouvé en a.
   - Prendre un nœud comme un *nœud locus*.
   - Placer le caractère spécial ϕ à la position correspondant au nœud choisi, donc obtenir le motif $p\phi q$.
   - Calculer la valeur de hachage de mot $x' = p\phi q$ donc $h(x')$.
   - Interroger le dictionnaire des listes de substitutions (avec $h(x')$) afin d'obtenir une liste de caractères.
   - Remplacer chaque caractère c candidat à la place de ϕ pour obtenir le mot candidat pcq.
   - Faire une recherche exacte dans le Trie pour le mot pcq.
   - Si on le trouve, ce mot pcq est une solution approchée, on ajoute le *nœud locus* de cette solution comme un nœud valide à la liste des solutions.

On appelle cette 1re méthode *1_ err_ SL*.

6.5.2 La 2e stratégie : choisir les chemins candidats avec les caractères de substitutions

Dans la recherche approchée dans un Trie, on doit tester tous les fils d'un nœud pour arriver à la solution approchée. Notre but est de savoir quel est le chemin qui mène à la solution sans tester tous les chemins sortants d'un nœud donné. Pour cela, dans cette deuxième stratégie, dans les nœuds où on doit appliquer l'opération de distance d'édition pour trouver les solutions approchées, on interroge le dictionnaire des listes de substitutions en même temps que l'on traverse le Trie du haut vers le bas pour trouver les caractères qui marquent les transitions à vérifier. Plus précisément, supposons pour un nœud traversé en profondeur d, on interroge le dictionnaire des listes de substitutions pour le motif de substitution $p[1..d-1]\phi[d+1..m]$ afin d'obtenir une liste de caractères. Ensuite, on continue à traverser les transitions marquées par les caractères de la liste obtenue seulement.

Les étapes de cette méthode sont détaillées dans l'algorithme suivant :

Algorithme_tous_les_nœuds_valides_1err_SL_node
Entrée : le préfixe requête x.
Sortie : une liste de nœuds valides.

a. Trouver le *nœud locus nd* de x.

b. Si le *nœud locus nd* est à une profondeur d'au moins $|x|$, cela signifie que l'on a une solution exacte, donc ajouter ce nœud à la liste des nœuds valides.

c. Pour chaque nœud sur le chemin menant au *nœud locus* trouvé en a.
   - Prendre un nœud comme un *nœud locus*.
   - Placer le caractère spécial ϕ à la position correspondant au nœud choisi, donc obtenir le motif $p\phi q$.
   - Interroger le dictionnaire des listes de substitutions afin d'obtenir une liste de caractères.
   - Continuer à traverser les transitions enfants marquées par les caractères dans la liste de substitutions, afin de vérifier le suffixe q qui reste.
   - Si une correspondance approchée de x est trouvée, on ajoute le nœud de cette solution comme un nœud valide à la liste des solutions.

On appelle cette deuxième méthode *1-err_SL_Node*.

Cette 2^{e} méthode est très proche de la première méthode *1-err_SL*. En effet, au lieu de refaire la recherche depuis le début, il suffit juste de choisir le bon chemin dans le Trie avec le caractère récupéré de la liste de substitutions, ensuite, continuer la vérification du suffixe qui reste.

6.5.3 La 3^{e} stratégie : utiliser la liste de substitutions dans seulement les 1ers niveaux du Trie

La recherche approchée dans un arbre (Trie ou autre) peut se diviser en deux (2), les premiers niveaux qui sont proches de la racine, et les niveaux inférieurs qui sont proches des feuilles.

Dans les premiers niveaux, chaque nœud à un grand sous-arbre et un nombre de fils important, et cela influe sur la recherche approchée et la rend lente, car il y a beaucoup de possibilités et de chemins à tester avant de trouver le bon qui représente la solution. Donc la méthode simple de traiter ce problème est inefficace (celle qui applique directement l'opération de distance d'édition dans chaque nœud).

Par contre si on considère que l'on recherche l'erreur seulement dans les derniers niveaux, la méthode simple (naïve) est suffisante et efficace, car chaque nœud à un petit sous-arbre et un nombre de fils très petit, et cela rend la recherche rapide.

En se basant sur cette observation, nous avons proposé une méthode hybride qui utilise les caractères de substitutions juste dans les 3 premiers niveaux, et pour le reste des niveaux, on applique la recherche simple dans le Trie.

À chaque fois qu'on supprime un niveau, un très grand nombre de chemins s'éliminent. Avec seulement trois niveaux, on élimine un nombre très important de possibilités, et on réduit la taille des sous-arbres qui restent. Le but, c'est de minimiser l'espace mémoire utilisé par le dictionnaire des listes de substitutions (pour tous les préfixes de 2 à D, avec $D = 6$, comme nous l'avons expliqué au début de cette section), et en même temps gagner dans le temps d'exécution. Avec trois niveaux, les tests donnent de bons résultats.

Si l'erreur est dans ces 3 premiers niveaux, on applique la recherche avec la méthode qui utilise les caractères de substitutions afin de choisir les bons chemins. Sinon on applique la méthode de recherche naïve dans le Trie.

Dans la partie test (voir la sous-section 6.10.2) la méthode *1err_SL_node* donne de meilleurs résultats par rapport à la première méthode *1err_SL*. Pour cela, dans cette 3e méthode hybride, on utilise la 2e méthode *1err_SL_node* dans les 3 premiers niveaux, et pour le reste, on utilise la méthode classique expliqué dans la sous-section 6.3.1.

Les étapes de cette méthode sont résumées dans l'algorithme suivant :

Algorithme_tous_les_nœuds_valides_1err_SL_3_level
Entrée : le préfixe requête x.
Sortie : une liste de nœuds valides.

a. Trouver le *nœud locus nd* de x.

b. Si le *nœud locus nd* est à une profondeur d'au moins $|x|$, cela signifie qu'on a une solution exacte, donc on ajoute ce nœud à la liste des nœuds valides.

c. Pour chaque nœud sur le chemin menant au *nœud locus* trouvé en (a).
   - Prendre un nœud *nd* comme un *nœud locus*.
   - Si la profondeur de *nœud locus* < 3 ($Prof(nd) < 3$) alors :
     - Placer le caractère spécial ϕ à la position correspondant au nœud choisi, pour obtenir le motif $p\phi q$.
     - Interroger le dictionnaire des listes de substitutions afin d'obtenir une liste de caractères.
     - Continuer à traverser les transitions marquées par les caractères qui sont dans la liste seulement, afin de vérifier le suffixe q qui reste.

- Si une correspondance approchée de x est trouvée, alors ajouter le nœud de cette solution comme un nœud valide à la liste des solutions.

– Sinon, $Prof(nd) \geq 3$ alors :
  - Faire une opération de distance d'édition sur toutes les transitions sortantes (sauf pour celle qui est sur le chemin qui mène au *nœud locus* trouvé en a).
  - Continuer une recherche exacte pour le suffixe restant de la requête dans le sous-arbre descendant de ce nouveau *nœud locus*.
  - Si une correspondance approchée de x est trouvée alors on ajoute le nœud de cette solution comme un nœud valide à la liste des solutions.

On appelle cette méthode : *1-err_SL_3_level*.

6.5.4 Évaluation de la complexité

Nous donnons une évaluation de la complexité moyenne pour les deux méthodes *1_err_SL* et *1-err_SL_Node*.

Théorème 6 *Étant donné un dictionnaire D de d mots, et un mot requête q d'une longueur m. La complexité temporelle moyenne pour une requête préfixe de la recherche approchée pour l'auto-complétion est de $O(m^2)$.*

Preuve 6 *L'algorithme fait une recherche exacte jusqu'à la position de l'erreur. Tous les nœuds depuis la racine à la position de l'erreur vont être vérifiés pour trouver les solutions approchées. Le nombre maximal de positions est égal à m.*

Pour une seule position $i \in [1..m]$, on applique la méthode de listes de substitutions pour obtenir les lettres pouvant mener à une solution. En moyenne on a juste une seule position (voir la preuve dans le chapitre 4 (la recherche avec hachage), dans la preuve 2, et la section expérimentation section 4.6). Donc on aura juste un seul chemin à vérifier, ce qui donne un temps $O(m)$ pour vérifier le préfixe avant la position de l'erreur et le suffixe après la position de l'erreur. Le calcul de la liste de substitutions se fait quant à lui en temps $O(1)$ (voir le détail dans le chapitre sur la recherche avec hachage). Donc le temps total pour une seule position est $O(m)$ et le temps total pour toutes les positions est $O(m \times m) = O(m^2)$.

6.6 Les résultats Top-K

La liste des résultats dans l'auto-complétion pourrait être très grande, surtout dans les deux cas suivants :

1) Lorsque l'utilisateur tape moins de 3 caractères, on aura un très grand sous-arbre pour rendre toutes ses feuilles comme résultats, et donc beaucoup de mots. Par exemple, si

on a qu'un seul caractère dans certains cas, on aura presque la totalité de l'arbre. Beaucoup de bibliothèques (par exemple *JQuery UI Autocomplete*) proposent comme paramètre de réglage un nombre minimum de caractères à taper pour traiter la requête afin d'éviter de rendre une grande partie de dictionnaire comme résultat.

2) Lorsqu'on accepte les erreurs dans la requête, une conséquence évidente est que la liste de suggestions devient très grande aussi.

Afin de réduire le nombre de résultats, et mettre en avant les résultats les plus importants qui satisfont le plus le besoin de l'utilisateur, on utilise le système dit **Top-k** qui rapporte les k suggestions les plus hautement classées dans un ordre décroissant par rapport à leur score.

Le paramètre k est donné par l'utilisateur, le nombre de résultats dépend de la taille de l'interface du dispositif utilisé. Généralement, on utilise une valeur entre 10 et 15. Cette méthode est appelée **Top-k complétion** (Top-k suggestion).

Comme on a expliqué dans la sous-section 6.3.2, on utilise un Trie qui stocke tous les scores des mots dans les nœuds feuilles, et chaque nœud interne garde le score maximal de ces enfants. Le Trie avec les scores et une autre structure de donnée nommée la file de priorité sont combinées dans le but de trouver les k résultats qui ont les scores les plus élevés. L'algorithme permettant de trouver la liste Top-k comme expliqué précédemment est : **Algorithme de liste de suggestions**.

Pour faire la recherche Top-k, les mots du dictionnaire doivent avoir un score qui représente leur importance. Dans le cas où certains mots n'ont pas de score, on leur attribue le score NULL, donc la valeur 0.

Exemple $D = \{"abcd\#10", "abcdefg\#5", "xyz\#02", "ibra", "nasa"\}$.

Dans cet exemple, les mots $\{"abcd", "abcdefg", "xyz"\}$ ont des scores (ils apparaissent à la fin avec un séparateur spécial #). Par contre, les deux mots $\{"ibra", "nasa"\}$ n'ont pas de score, dans ce cas, on leur donne la valeur 0.

Dans une recherche, les résultats ayant un même score sont classés par ordre lexicographique.

Dans les deux sous-sections qui suivent, nous expliquons le concept des résultats Top-k statique et dynamique, et nous donnons une méthode simple qui donne la possibilité à l'utilisateur de visionner tous les résultats possibles d'une requête groupe par groupe sans encombrer l'interface graphique et sans refaire la recherche.

6.6.1 Classement Top-k statique et dynamique.

Les résultats dans un classement statique restent toujours les mêmes dans le temps. Si on tape la même requête, plusieurs fois, l'algorithme retournera comme résultat toujours la même liste, même si par exemple, on choisit toujours le dernier mot de la liste.

Comme exemple des classements statiques : les noms des villes du monde selon leur superficie ou leur nombre d'habitants (le nombre d'habitants change, mais généralement les statistiques sont faites dans des périodes un peu éloignées), l'ordre lexicographique du dictionnaire,... etc.

Dans un classement Top-k dynamique, les résultats dans la liste peuvent changer avec la même requête, car les scores changent. Par exemple si dans une application donnée le score du mot choisi dans la liste augmente, donc à la prochaine requête sa position dans la liste change, alors si l'utilisateur choisit toujours le même mot, ce dernier va apparaître au top de la liste (comme par exemple dans l'IDE de programmation "Visuel Studio").

Le score statique est le nombre donné pour chaque mot, ces scores peuvent être mis à jour. Le score pour un classement dynamique peut être une valeur positive/négative, et le score initial sera changé avec une unité spécifique. Si on a une valeur positive, le score augmente, par exemple : le système de Google +1.

Dans notre travail, les scores sont des nombres entiers donnés pour chaque mot dans le dictionnaire. Lorsque l'utilisateur choisit un mot depuis la liste des résultats Top-k affichée, on récupère le mot choisi par l'utilisateur, et on fait une mise à jour de son score (par exemple avec +1).

Comment faire une mise à jour de score : Pour faire la mise à jour de score du mot choisi, on doit faire la mise à jour à tous les nœuds qui sont sur le chemin dans le Trie depuis la racine jusqu'au nœud feuille. Pour cela, nous avons choisi dans notre travail, de refaire la recherche du mot depuis le début et de sauvegarder dans une liste tous les nœuds trouvés sur le chemin jusqu'au nœud feuille. Lorsqu'on arrive au dernier nœud, on récupère le score et on lui ajoute l'unité de mise à jour (par exemple +1). Ensuite, on prend le nouveau score et on le compare avec les scores des nœuds qui sont sur le chemin de la recherche et qui ont été sauvegardés pour leur appliquer une mise à jour. Donc si le nouveau score est plus grand que les scores de ces nœuds, alors on les modifie. On applique la mise à jour à tous les nœuds qui sont sur le chemin de la recherche pour rééquilibrer le Trie par le nouveau score, puisque chaque nœud interne contient le score le plus élevé de ces fils.

6.6.2 Afficher tous les résultats ordonnés en Top-k groupe par groupe

Dans certains cas, l'utilisateur veut voir une grande partie ou la totalité, des résultats de sa requête. L'affichage restreint aux résultats Top-k ne satisfait pas le besoin de l'utilisateur, comme par exemple une requête de recherche d'un produit donné, un livre ou un film, etc.

Dans ce cas il est plus adéquat de proposer un système qui affiche tous les résultats à l'utilisateur, à sa demande.

Dans notre travail, nous nous inspirons des moteurs de recherche comme Google qui affiche les résultats Top-k et propose des boutons/liens en bas de la page (next) pour afficher les pages suivantes.

En affichant les résultats Top-k, on fournit une option pour obtenir les autres résultats, et les afficher groupe par groupe. Pour une démonstration, voir dans http://5.135.166.57/APPACOLIB/ [1], le bouton `next` , ou la combinaison (CTRL + flèche droite) pour obtenir la même fonction.

Lorsque l'on utilise cette technique, on ne refait pas la recherche du mot requête à chaque fois. Il suffit juste d'aller à l'index (la file de priorité et le Trie) et extraire les résultats par groupe de k éléments.

Lorsque on a une requête, on recherche d'abord les nœuds valides par l'algorithme **Algorithme tous les nœuds valides**, ensuite, on les range dans une file de priorité pour trouver les k résultats par l'algorithme (**Algorithme de liste de suggestions**) expliquer dans la section 6.3. S'il reste des nœuds dans la file de priorité, cela veut dire que nous avons un nombre de résultats plus que k. Ainsi lorsque l'utilisateur veut afficher les k éléments suivants, il suffit juste d'exécuter les mêmes opérations de l'algorithme (**Algorithme de liste de suggestions**) pour extraire les k résultats depuis les nœuds qui restent dans la file de priorité.

Les k résultats ne sont pas pré-calculés pour les afficher par groupe de k éléments, car cela prend un temps considérable pour trouver tous les résultats. Mais à chaque fois que l'utilisateur demande de voir le groupe suivant des résultats, on utilise la file de priorité et on applique les étapes de l'algorithme (**Algorithme de liste de suggestions**). Généralement, l'utilisateur est satisfait juste par le premier groupe des résultats ; dans certains cas, il visualise le premier groupe suivant.

Remarque : Si l'utilisateur ne trouve pas sa requête dans le premier groupe des résultats et ne veut pas consulter les autres groupes, il lui suffit juste de taper quelque caractères supplémentaire de sa requête pour faire plus de filtrage sur les résultats.

1. Visité le : 02-07-2016.

6.7 Éliminer la redondance (les résultats en double)

Dans la recherche approchée il y a toujours des résultats en double. Ce phénomène est encore plus large dans l'auto-complétion approchée. Lorsqu'on applique les trois opérations de distance d'édition pour rechercher des solutions avec une erreur, on peut atteindre les mêmes nœuds valides, et donc tous leurs fils sont des résultats en double.

Il existe deux cas où les nœuds valides causent une redondance des résultats :

1. On trouve le même nœud valide plus qu'une seule fois, par exemple avec l'opération de suppression on trouve nd_1 comme un nœud valide, et avec la substitution on retrouve le même nœud valide nd_1.
2. On trouve nd_1 comme un nœud valide (par exemple avec une opération de suppression), et ensuite, on trouve son fils nd_2 comme un nœud valide (par exemple avec une opération d'insertion), et par conséquent, tous les fils du deuxième nœud sont des résultats en double.

Comment résoudre ce problème des nœuds valides en double ?

Avant d'ajouter les nœuds valides à la liste des solutions, on vérifie d'abord s'ils existent dans cette liste ou non. L'auto complétion ne donne que les premiers éléments Top-k, cela signifie une petite liste d'éléments. Pour cela, on peut utiliser une petite table de hachage. Cette table de hachage est utilisée dans l'algorithme **Algorithme tous les nœuds valides**, avant l'ajout des nœuds valides à la liste des solutions.

Pour la relation entre les fils et les nœuds valides père, on utilise une table de hachage dans l'algorithme **Algorithme de liste de suggestions**, pour vérifier si un mot solution existe dans la liste de suggestions ou non.

6.8 Les stratégies client/serveur de l'auto-complétion

Il y a deux stratégies extrêmes pour l'auto-complétion : soit le programme s'exécute sur le coté serveur, et le client envoie les requêtes et gère les résultats seulement, ou bien le programme s'exécute du coté client (en utilisant JavaScript), et le client fait tous les traitements en local et affiche les choix (résultats) finaux.

Entre ces deux extrêmes, plusieurs scénarios peuvent être considérés. Ils dépendent de plusieurs paramètres, parmi les plus importants :

a) La taille du dictionnaire.

b) La charge du serveur.

c) La vitesse de connexion.

d) Le nombre de connexions courantes au niveau du serveur (nombre de connexions simultanées).

e) La puissance du calcul du coté client (la puissance de calcul du coté serveur est supposée beaucoup plus puissante ($\gg$) que celle du client, mais partagée).

f) Le classement statique ou dynamique.

g) Le délai entre deux requêtes.

Notre bibliothèques est disponible sur https://github.com/AppacoLib/api.appacoLib, elle permet d'exécuter nos algorithmes soit sur le serveur ou du côté client, selon le meilleur scénario recherché. Le cœur de nos bibliothèques est écrit en (C/C++) du côté serveur, et en JavaScript pour le côté client.

Nous expliquons ci-dessous ses deux utilisations principales.

Appacolib du coté serveur, Typiquement si on a un dictionnaire très volumineux (plus de 3 Mo d'entrées), on peut utiliser la bibliothèque écrite en C/C++ avec un module FASTCGI pour fournir une liste de suggestions dans un format JSON et renvoyer les résultats à travers des appels AJAX, puis afficher la liste en utilisant la bibliothèque écrite en JavaSscript (les résultats sont affichés généralement en dessous du champ de texte qui s'auto-complète). Pour une démonstration voir http://5.135.166.57/APPACOLIB/result_from_server_CGI.html.

Appacolib du coté client, Du coté client, on a plusieurs stratégies dépendant du scénario voulu :

1) Toute l'opération de l'auto-complétion approchée peut être effectuée au niveau du navigateur du coté client. Si le dictionnaire n'est pas très volumineux (par exemple moins de 3 Mo d'entrées), on peut effectuer toutes les opérations au niveau local : la construction de l'index, la recherche de requête préfixe, et l'affichage de la liste des suggestions. Dans ce cas, on doit fournir une liste de mots pour construire l'index. La source du dictionnaire peut être locale ou au niveau du serveur, et peut être en différents formats (fichier, une chaîne de caractères avec des séparateurs, un tableau, ...etc.). Pour plus de détails sur le format de source pour construire l'index voir la documentation de *AppacoLib* dans https://github.com/AppacoLib/api.appacoLib/tree/master/doc_appaco_lib.

2) Afficher les résultats au niveau du client uniquement : en utilisant un appel AJAX pour envoyer la requête et recevoir une liste de réponses venues du serveur. Dans ce cas, on peut combiner les deux parties de notre bibliothèque : utiliser la bibliothèque en C/C++ du coté serveur pour gérer la création de l'index et répondre aux requêtes de l'auto-complétion, et utiliser la bibliothèque du coté client pour recevoir les résultats et les afficher.

Lorsqu'on utilise la bibliothèque du coté client pour juste envoyer la requête au serveur, et gérer l'affichage des résultats finaux, le serveur n'est pas forcément notre bibliothèque écrite en C/C++, mais il peut être n'importe quel système qui accepte les requêtes AJAX et qui fournit les résultats en format JSON, comme par exemple un simple système écrit en PHP qui lit les résultats depuis un simple tableau, ou peut être un système plus complexe comme la gestion des complétions depuis les bases de données.

3) Télécharger le Trie construit dans le serveur, et exécuter les requêtes localement.

6.9 À quelle vitesse devrait être l'auto-complétion

Pour être utile, l'auto-complétion doit être sensible, réactive et instantanée. Quand l'auto-complétion est gérée du coté serveur, le calcul du temps de traitement prend en considération le temps pour taper un caractère et le temps du transfert des données entre le client et le serveur, et le temps du traitement du côté client pour gérer l'interface.

Le nombre de frappes par seconde est de $7,5$ [123], cela donne pour un caractère, l'utilisateur met ($133ms$). Miller [124], a montré que pour être instantané pour l'utilisateur, le temps de réponse doit être inférieur à $100ms$ (ms : MilliSeconde). Cela signifie que l'on a seulement environ $100ms$ pour traiter la requête et donner une liste de suggestions qui contient les résultats Top-k. Cette période de $100ms$ inclut le temps de la communication entre le serveur et le client et le temps de gestion de l'interface. Ainsi, le côté serveur doit traiter et renvoyer les résultats très rapidement.

Si on considère seulement le côté client pour faire de l'auto-complétion indépendamment du serveur, on aurait exactement $100ms$ pour effectuer tout le traitement (sans compter le temps de transfert, car tous les calculs sont faits en local).

6.10 Tests et expérimentations

Les tests ont été effectués sur deux fichiers : le dictionnaire Anglais ($213,557$ mots, $2,4$ Mégaoctets) et un extrait des titres des articles de Wikipédia ($1,2$ millions titres, $11,5$ Mégaoctets).

Notre bibliothèque est disponible sur `https://github.com/AppacoLib/api.appacoLib`. Le cœur de nos bibliothèques principales est écrit en (C/C ++) pour le côté serveur, et en JavaScript pour le côté client. L'implémentation de notre bibliothèque est modulaire, elle est divisée en plusieurs fichiers sources indépendants selon les fonctionnalités voulues. **La bibliothèque en C/C++ contient 7 593 lignes de code**, et la bibliothèque écrite en **JavaScript contient 3 884 lignes** de code source. Pour une démonstration de notre bibliothèque voir : `http://5.135.166.57/APPACOLIB/`.

Dans nos tests, nous avons considéré les deux scénarios extrêmes cités avant, l'auto-complétion est faite seulement par le client, et l'auto-complétion faite par le serveur et gérée par le client dans le navigateur.

Nous avons testé les différentes méthodes proposées, la complétion exacte ou approchée, les méthodes pour s'adapter aux modifications de l'utilisateur sur la requête, l'utilisation du hachage pour générer une liste de substitutions pour améliorer la recherche approchée.

Les paramètres pris en considération lors des tests sont :

- Le temps de construction de la structure de données.
- L'espace occupé par l'index.
- Le temps de réponse aux requêtes en fixant le k et on changeant la longueur des préfixes de 2 à 6, pour l'auto-complétion exacte et approchée.
- L'affectation du nombre de résultats k sur le temps du calcul.

Les mots requêtes ont été obtenus en choisissant au hasard parmi les préfixes des mots dans le dictionnaire et introduisant des erreurs aléatoires dans des positions aléatoires. Le temps final a été obtenu en faisant la moyenne de 1000 mots requêtes distinctes.

6.10.1 Le coté serveur, C/C++

Les tests ont été effectués sur un ordinateur Intel core-2 duo e8400, windows 7, 3.0 GHz, 2GB de RAM, avec un compilateur GNU GCC version 4.4.1.

6.10.1.1 Test du temps de réponse pour l'auto-complétion exacte et approchée

Dans ce test, nous avons focalisé sur le temps de réponse à une seule requête. Nous avons fixé k à 10 résultats dans la liste, et nous avons changé les tailles des préfixes de 2 à 6. Les tests sont faits avec la totalité du dictionnaire, l'espace mémoire occupé par l'index est d'environ 5 Mo pour le dictionnaire Anglais (En) et 29 Mo pour le dictionnaire WikiTitle (Wi). Le temps de construction du Trie pour le dictionnaire Anglais (En) est d'environ $177ms$ et $896ms$ pour le dictionnaire WikiTitle (Wi).

Longueur de requête	Exacte (En)	1-erreur (En)	Exacte (Wi)	1-erreur (Wi)
2	0,02	0,11	0,02	0,34
3	0,017	0,16	0,017	0,44
4	0,013	0,19	0,014	0,53
5	0,010	0,21	0,012	0,58
6	0,009	0,25	0,01	0,61

TABLE 6.1 – Temps de requête + Top-k (en millisecondes) pour les dictionnaires Anglais (En) et WikiTitles (Wi).

Dans le tableau 6.1, pour l'auto-complétion exacte, il est intéressant de noter que le temps de réponse pour les requêtes courtes (long = 2) est plus long que pour celui des requêtes longues (long = 6). Cela est dû principalement à l'existence de beaucoup plus de résultats, au delà du seuil fixé, lorsque la requête est courte. En effet, avec une requête courte, la recherche s'arrête dans les premiers niveaux de l'arbre, donc on aura un sous-arbre très grand où on doit trouver les résultats Top-k, ce qui augmente le temps de calcul.

Cependant, pour l'auto-complétion avec erreur, le temps est beaucoup plus élevé comparé à l'auto-complétion exacte. Cela est dû au nombre d'opérations de la complétion exacte et approchée. Dans l'auto-complétion exacte, il suffit juste d'exécuter une seule recherche exacte puis de récupérer les résultats depuis un seul nœud. Par contre, dans l'auto-complétion approchée, on est obligé de faire une recherche exacte plus une opération de distance d'édition sur tous les nœuds sur le chemin depuis la racine jusqu'au dernier nœud où la recherche exacte s'est arrêtée (faire une recherche approchée avec 1 seule erreur), ensuite, trouver les résultats depuis plusieurs nœuds, ceux de la solution exacte (1 seul nœud), et ceux de la solution approchée (plusieurs nœuds).

La différence de temps dans les résultats de recherche exacte et approchée existe, mais elle devient négligeable par rapport à l'ensemble du temps total de réponse qui ne doit pas dépasser les $100ms$.

6.10.1.2 Test avec différentes tailles du dictionnaire

Dans ce test, nous avons fait varier la taille du dictionnaire (le nombre des mots) à chaque fois pour voir l'influence de ce paramètre sur les 3 aspects suivants : le temps de création de l'index, le temps de réponse de requête et l'espace mémoire occupé par l'index.

Le temps de création de l'index : nous commençons par le temps de création de notre index, voir la figure 6.7 et le tableau 6.2, qui illustrent le changement du temps de création des 2 dictionnaires selon leurs tailles.

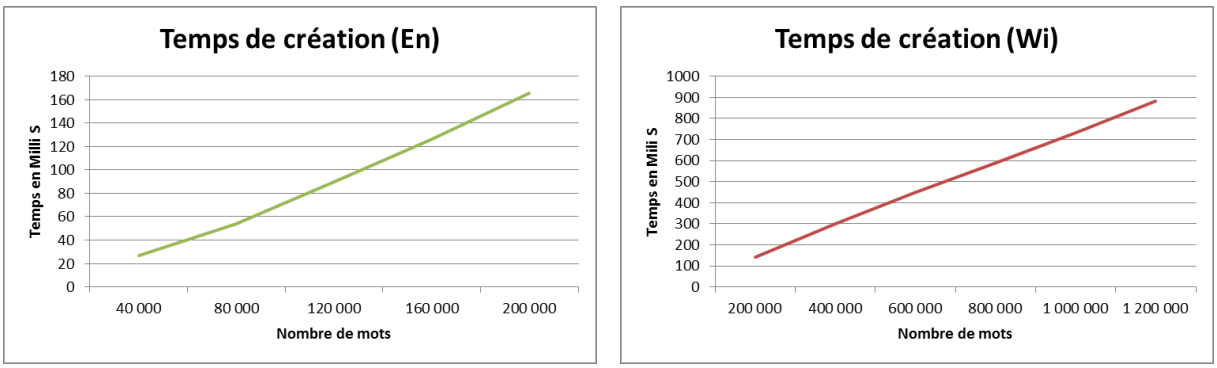

FIGURE 6.7 – Le temps de création pour différentes tailles des dictionnaires Anglais (En) et WikiTitle (Wi).

Dans la figure 6.7, on remarque bien que dans les 2 dictionnaires, le temps de création augmente en corrélation avec la taille du dictionnaire d'une façon linéaire. Pour le dictionnaire Anglais (En), le temps commence de $20ms$ jusqu'à $170ms$, et pour WikiTitle (Wi) le temps est entre $150ms$ et $900ms$.

Nombre Mots	Création (EN)	Création (Wi)
40 000	27	24
80 000	54	48
120 000	90	78
160 000	126	106
200 000	165	141

TABLE 6.2 – Le temps de création en millisecondes pour les deux dictionnaires Anglais (En) et WikiTitle (Wi) avec les mêmes tailles.

Il est intéressant de remarquer dans le tableau 6.2, que les temps de création pour les deux dictionnaires avec la même taille sont proches. La différence qui se trouve entre ces deux dictionnaires est due à la nature de chaque dictionnaire (la distribution des : mots, caractères, préfixes, suffixes, la longueur moyenne des mots, etc.)

Le temps de recherche d'une requête : Le temps de l'auto-complétion des requêtes change en fonction de la taille de l'index et du type de l'index. Nous utilisons un Trie compact où le temps de recherche d'une requête exacte ne dépend pas de la taille de l'index, mais il dépend du nombre de caractères de la requête elle-même. Dans notre cas, nous avons la recherche de Top-k et la recherche approchée, les deux opérations sont influencées par la taille de l'index (le nombre des nœuds dans le Trie, et si la distribution des scores est équilibrée ou non). Les résultats sont illustrés dans les tableaux 6.3 et 6.4.

Nb des Mots	exacte (En)	exacte (Wi)	1-err (En)	1-err (Wi)
40 000	0,0156	0,0179	0,092	0,177
80 000	0,0164	0,0183	0,106	0,224
120 000	0,0159	0,0170	0,132	0,251
160 000	0,0166	0,0177	0,139	0,252
200 000	0,0160	0,0176	0,146	0,254

TABLE 6.3 – Le temps de recherche + Top-k en millisecondes pour différentes tailles des dictionnaires Anglais (En) et WikiTitle (Wi).

Dans les tableaux 6.3 et 6.4, on remarque que dans les deux dictionnaires, le changement de taille n'influe pas sur le temps de l'auto-complétion exacte (recherche+Top-k), le temps est presque stable. Dans le dictionnaire Anglais, le temps est autour de $0,016ms$. Pour le dictionnaire Wikititle, le temps est autour de $0,018ms$. On remarque qu'entre la taille de 40 mille mots jusqu'à la totalité du dictionnaire Wikititle (Wi) (1,2 millions

Nb des Mots	exacte (Wi)	1-err (Wi)
200 000	0,0176	0,254
400 000	0,0180	0,282
600 000	0,0175	0,320
800 000	0,0181	0,350
1 000 000	0,0178	0,373
1 200 000	0,0179	0,485

TABLE 6.4 – Le temps de recherche + Top-k en millisecondes pour WikiTitle (Wi) avec des grandes tailles.

mots), le temps reste presque stable. Entre les deux dictionnaires, même avec la grande différence du nombre de mots 200 mille mots (En), et $1,2$ million pour (Wi), on remarque que le temps ne change pas beaucoup, il y a une différence de $0,02ms$.

Dans ce test, nous avons fait la recherche dans le Trie plus la liste Top-k avec $k = 10$. Le temps de la liste Top-k dépend des scores des mots dans le dictionnaire. Dans notre Trie chaque nœud interne sauvegarde le score maximal de ses fils pour qu'on puisse trouver rapidement le mot qui a le score le plus élevé (cela prend le temps de traverser la hauteur du Trie), et avec l'utilisation de la file de priorité le temps reste stable pour un nombre de résultats k fixe.

Le temps de l'auto-complétion avec erreur, change avec le changement de la taille du dictionnaire. Lorsque la taille du dictionnaire augmente cela implique que la taille de l'index augmente aussi (plus spécialement le Trie). Lorsque le Trie est grand cela signifie que le nombre des chemins à vérifier pour trouver des solutions approchées est grand, et cela influe sur le temps d'exécution final. Pour le Top-k, (l'auto-complétion approchée) on a plusieurs nœuds à traiter, et donc plusieurs sous-arbres, contrairement au cas de l'auto-complétion exacte où on a juste un seul nœud et donc un seul sous-arbre. Lorsque la taille du dictionnaire est grande, cela signifie que les tailles des sous-arbres aussi sont grandes, et cela, influe sur l'augmentation du temps d'exécution de Top-k.

L'espace mémoire : Le changement de la taille du dictionnaire initial influe directement sur la taille de l'index (le Trie). Lorsqu'on augmente le nombre de mots dans le dictionnaire, cela implique aussi une augmentation de la taille de l'index, car des nouveaux mots sont ajoutés au Trie. Voir la figure 6.8.

La figure 6.8, montre l'augmentation de la taille de l'index en corrélation avec l'augmentation du nombre de mots.

L'augmentation de la taille de l'index est linéaire, pour le dictionnaire Anglais (En). Elle commence à 1 Mo pour 40 mille mots jusqu'à 5 Mo pour 200 mille mots. La taille initiale du dictionnaire Anglais est $2,4$ Mo. Et il en est de même pour le dictionnaire WikiTitle (Wi) : elle commence à 5 Mo pour 200 mille mots jusqu'à 29 Mo pour $1,2$ million mots (la taille initiale de dictionnaire WikiTitle est de $11,5$ Mo).

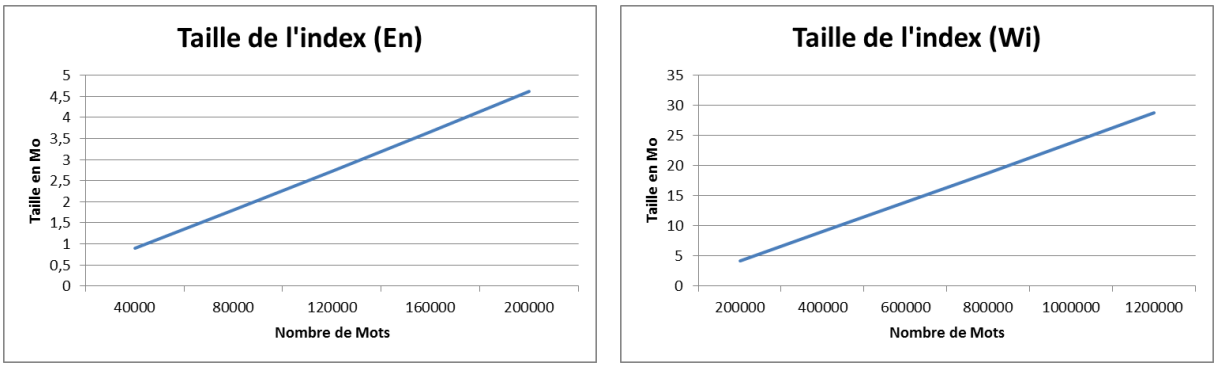

FIGURE 6.8 – Changement de la taille de l'index en fonction de la taille des dictionnaires Anglais (En) et WikiTitle (Wi).

6.10.1.3 Test des trois méthodes qui s'adaptent au comportement de l'utilisateur

Dans la section 6.4, nous avons proposé et expliqué trois méthodes afin de s'adapter à l'utilisateur lorsque il tape sa requête. Les trois méthodes sont :

1. *search_from_root*, la méthode simple, faire la recherche depuis la racine à chaque fois.
2. *search_end_node*, faire la recherche depuis le dernier nœud, si cela est possible.
3. *search_tab_node*, faire la recherche depuis le nœud du plus grand préfixe commun.

Nous n'avons fait que les tests pour les deux premières méthodes. La troisième méthode (*search tab node*), a une complexité élevée par rapport à la recherche simple. Pour plus de détails voir la sous-section 6.4.3. Pour cette troisième méthode une fois que nous avons implémenté et commencé à la tester, les premiers tests n'ont pas donné de bons résultats et la méthode cause beaucoup de Bugs. C'est la raison pour laquelle nous avons arrêté les tests.

Pour réaliser les tests, nous avons pris un tableau qui contient 100 mots, et nous avons généré un autre tableau qui contient les préfixes du premier dans l'ordre, avec une distance entre le mot et son préfixe de *nb_char_def*. Pour comparer les méthodes *search_from_root* et *search_end_node* : nous avons fait le test avec le préfixe ensuite le mot, l'un après l'autre, exemple : *abc* ensuite *abcd*, et on mesure le temps.

Nous avons 2 tableaux (les mots avec leurs préfixes), donc 200 mots requêtes. Nous avons refait le test 10 fois, et à la fin, nous avons pris la moyenne. Les résultats sont illustrés dans la figure 6.9.

Dans la figure 6.9, l'axe des x, représente la longueur de la première requête suivie de la longueur de la seconde requête en dessous, et l'axe des y représente le temps en millisecondes. Dans ces deux graphes, on observe clairement que la méthode *search_end_node*

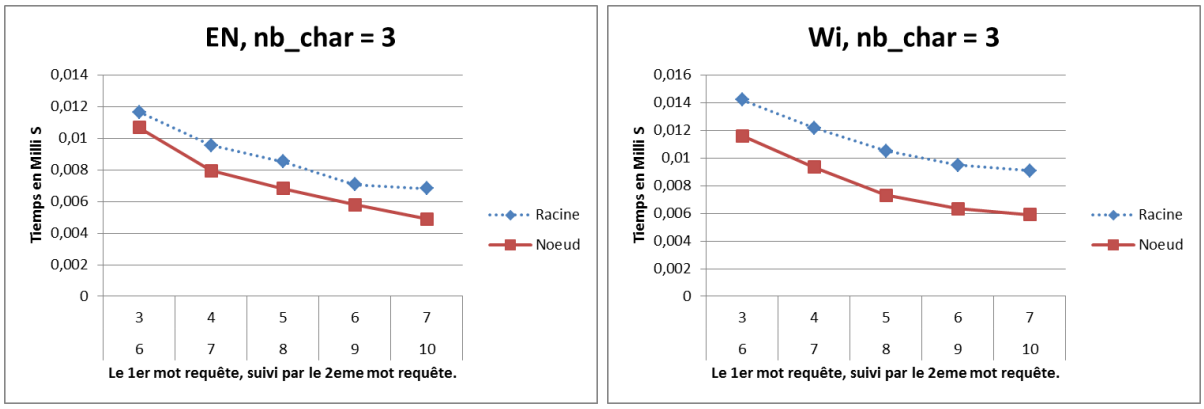

FIGURE 6.9 – Temps de requête en millisecondes pour les 2 méthodes *search_from_root* et *search_end_node*, pour les dictionnaires Anglais (En) et WikiTitle (Wi), avec $nb\_char\_def = 3$.

(continuer depuis le dernier nœud) donne de meilleurs résultats.

Remarque :
Nous observons que la méthode *search_end_node* améliore légèrement les résultats, mais en réalité, cela n'a pas une grande influence sur le comportement du système dans l'interaction avec l'utilisateur si tout le traitement s'effectue du coté client, car les temps des deux méthodes sont très petits comparés au temps nécessaire de l'interaction avec l'utilisateur qui est $100ms$. Mais si le traitement s'effectue du coté serveur, cette amélioration est nécessaire, car le temps de l'interaction avec l'utilisateur est la somme de tous les temps de toutes les étapes : le traitement, le temps du transfert réseau (l'envoi de la requête et la réception des résultats), etc.

6.10.1.4 Influence du paramètre k (Top-k) sur le temps de traitement

Le paramètre k est le nombre de résultats qui doivent être affichés à l'utilisateur selon l'importance des résultats. Dans cette partie, nous changeons à chaque fois ce paramètre k, pour observer son influence sur le temps d'exécution.

Dans ce test, on ne présente que le test de dictionnaire WikiTitle, car ses résultats sont plus clairs (pour le dictionnaire Anglais, on obtient le même résultat pour le changement du temps du calcul en fonction du paramètre k). Nous utilisons des mots requêtes d'une longueur de trois caractères, afin d'avoir un grand sous-arbre et donc beaucoup de résultats, et cela permet de choisir le nombre de résultats à afficher k librement. Car dans le cas contraire, un mot requête long va nous donner juste une petite liste de résultats. Le résultat du test est illustré dans la figure 6.10.

Dans la figure 6.10, on remarque que lorsque k varie entre 10 et 20, le temps ne change pas beaucoup, ensuite il augmente de façon linéaire.

Comme nous l'avons expliqué dans la section 6.3, la méthode de recherche est divisée en deux étapes : la première étape est celle de la recherche dans le Trie pour trouver les

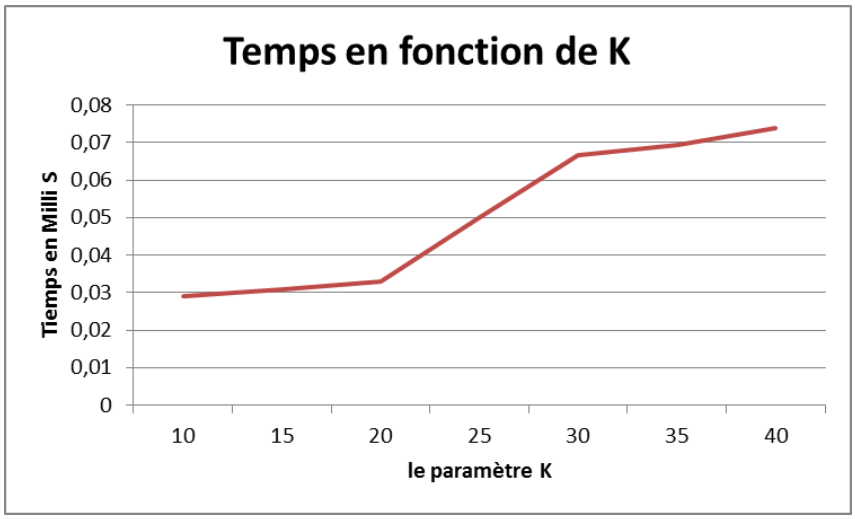

FIGURE 6.10 – Temps de requête en millisecondes en fonction du paramètre k, pour le dictionnaire Wikititle (Wi).

nœuds valides, la seconde étape insère ces derniers dans une file de priorité pour trouver la liste Top-k. Dans notre test, la première étape n'influe pas sur le temps d'exécution, car quelque soit k, la méthode de recherche restera la même. La seconde étape consiste à insérer les nœuds dans la file de priorité, puis itérativement, vérifier si le nœud en tête de cette file représente une feuille et d'ajouter le mot correspondant, à la liste des résultats, sinon ajouter ces fils à la file de priorité. On exécute cette opération jusqu'à l'obtention de k éléments dans la liste des résultats. Cela implique qu'à chaque fois que k augmente le nombre d'opérations augmente aussi, et cela implique un temps de calcul plus élevé.

6.10.2 La liste de substitutions, coté serveur

Nous avons expérimenté avec l'utilisation du dictionnaire des listes de substitutions afin d'accélérer le temps de la recherche approchée.

Le dictionnaire des listes de substitutions augmente la taille de notre index avec 0,7 Mégaoctet pour le dictionnaire Anglais et 5 Mo pour WikiTitle.

Pour comparer l'efficacité du dictionnaire des listes de substitutions, nous l'avons comparé avec la recherche approchée simple dans un Trie.

Le temps des requêtes par les trois méthodes, la recherche simple dans un Trie (*1-err*) et les 2 méthodes avec l'utilisation du dictionnaire des listes de substitutions (*1-err_SL* et *1-err_SL_Node*) pour des préfixes d'une longueur 2 jusqu'à 6 sont illustrés dans le tableau 6.5 et la figure 6.11.

Dans le tableau 6.5, on remarque que la méthode *1-err_SL_Node* améliore les résultats, elle est plus rapide par rapport à la méthode de recherche simple dans le Trie (*1-err*). Par contre, la méthode *1-err_SL* n'est pas aussi bonne que la deuxième méthode. D'après le tableau, elle donne un temps plus long pour les préfixes de petite longueur, et de bons temps avec les préfixes de longueur 5 et 6.

Longueur préfixe	1-err (En)	1-err_SL (En)	1-err_SL _Node (En)	1-err (Wi)	1-err_SL (Wi)	1-err_SL _Node (Wi)
2	0.11	0,18	0,11	0,35	0,50	0,37
3	0.14	0,17	0,13	0,45	0,56	0,40
4	0.16	0,15	0,11	0,51	0,50	0,32
5	0.17	0,13	0,09	0,53	0,36	0,20
6	0.17	0,10	0,06	0,55	0,25	0,13

TABLE 6.5 – Le temps des requêtes avec les 3 méthodes de la recherche approchée.

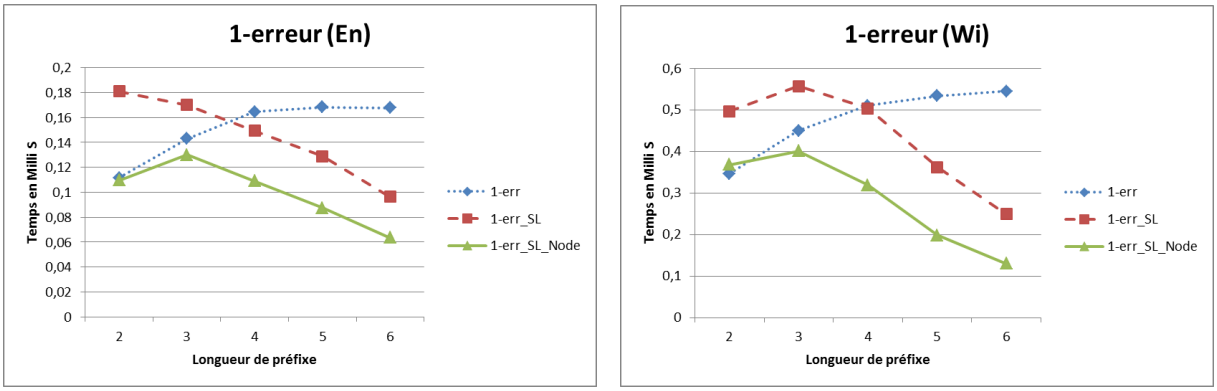

FIGURE 6.11 – Temps de l'auto-complétion approchée avec les 3 méthodes 1-err (naïve), 1-err_SL et 1-err_SL_Node, pour les dictionnaires Anglais (En) et WikiTitle (Wi).

La figure 6.11, montre que la méthode *1-err_SL_Node* donne de meilleurs résultats. La première méthode *1-err_SL* donne des résultats meilleurs que la méthode naïve *1-err* (la recherche simple dans le Trie) avec des préfixes de longueur > 5, mais cela n'est pas intéressant dans l'auto-complétion, car ce qui nous intéresse, ce sont les premiers caractères tapés par l'utilisateur (généralement 2, 3 ou 4).

Un fait intéressant : on peut observer que, lorsqu'on utilise le dictionnaire de substitutions, le temps de recherche diminue quand on augmente la longueur de la requête. Cela est dû au fait que la taille des listes de substitutions est plus courte pour les préfixes longs que les préfixes courts. Intuitivement, un motif de substitution plus long $p\phi q$, va correspondre à un nombre de préfixe moindre, par rapport au motif de substitution court.

6.10.2.1 Utiliser les caractères de substitutions juste dans les premiers niveaux du Trie

Dans cette partie, nous testons la méthode qui divise le Trie en deux parties, dans les premiers niveaux du Trie, nous utilisons le dictionnaire de substitutions, et dans le reste de Trie, nous utilisons la méthode de recherche naïve. Pour plus de détails, voir la sous-section 6.5.3.

Pour les tests, nous procédons de la même manière : l'erreur est aléatoire dans le préfixe, elle peut être dans tous les niveaux.

Nous faisons les tests pour des préfixes d'une longueur 4, 5 et 6 seulement, car avec

les préfixes de longueur 2 et 3, nous n'utilisons que la méthode du dictionnaire de substitutions, qui donne les mêmes résultats que la méthode *1-err_SL_node*.

Nous avons vu dans les tests précédents que la méthode *1-err_SL_node* a donné les meilleurs résultats comparée à la méthode naïve, et à la méthode *1-err_SL*. Pour cela, dans ce test, nous faisons la comparaison juste avec la méthode *1-err_SL_node*. Les résultats sont illustrés dans le tableau 6.6.

Longueur préfixe	1-err_SL_Node En	1-err_SL_3L En	1-err_SL_Node Wi	1-err_SL_3L Wi
4	0,11	0,10	0,32	0,26
5	0,09	0,07	0,20	0,17
6	0,06	0,06	0,13	0,10

TABLE 6.6 – Temps de l'auto-complétion en utilisant les 2 méthodes, *1-err_SL_node* et *1-err_SL_3_level* pour les 2 dictionnaires Anglais (En) et WikiTitle (Wi).

Dans le tableau 6.6, on observe que la méthode hybride *1-err_SL_3_level*, améliore les résultats (les résultats avec le dictionnaire WikiTitle le montrent plus).

L'utilisation de cette méthode permet de combiner les caractéristiques de la méthode simple qui a un temps de construction rapide et un espace mémoire petit par rapport à *1-err_SL_Node* avec la rapidité de cette dernière du temps de recherche.

La construction du dictionnaire de substitutions juste pour les trois premiers caractères (préfixe jusqu'à 3) au lieu de la totalité de la longueur du mot (ou comme nous avons fait dans les tests jusqu'à la taille 6) divise le temps de construction et l'espace du dictionnaire de substitutions par deux.

6.10.3 Le coté Client, JavaScript

Les tests ont été effectués sur un ordinateur Netbook qui a les caractéristiques suivantes : un processeur Intel Atom CPU N455 1,67 Ghz, 2Go RAM et window 7 starter 32 bits, (Aujourd'hui, cet ordinateur Netbook est moins puissant qu'un Smartphone).

Les différents tests pour l'auto-complétion exacte et approchée sur les dictionnaires Anglais et WikiTitle sont illustrés dans les tableaux (6.8, 6.7, 6.10, 6.9).

Longueur requête	Exacte (En)	1-erreur (En)	Exacte (Wi)	1-erreur (Wi)
2	0,006	0,47	0,0123	1,88
3	0,009	0,59	0,0242	2,41
4	0,0114	0,65	0,0186	2,42
5	0,0125	0,62	0,0206	2,28
6	0,0143	0,58	0,0233	2,08

TABLE 6.7 – Le navigateur Google Chrome 39. Le temps des requêtes + Top-k (en millisecondes) sur les dictionnaires Anglais (En) et WikiTitle (Wi). La construction du Trie pour le dictionnaire Anglais est $2,7$ secondes et 13 secondes pour le dictionnaire Wikititle.

Longueur requête	Exacte (En)	1-erreur (En)	Exacte (Wi)	1-erreur (Wi)
2	0,0021	0,4541	0,0026	1,1547
3	0,0035	0,5117	0,0059	1,453
4	0,0042	0,4474	0,0061	1,2086
5	0,0061	0,3554	0,0076	0,9461
6	0,005	0,2808	0,0081	0,7408

TABLE 6.8 – Le navigateur Firefox 29. Le temps des requêtes + Top-k (en millisecondes) sur les dictionnaires Anglais (En) et les Titres de Wikipédia (Wi). La construction du Trie pour le dictionnaire Anglais est $7,4$ secondes et 47 secondes pour le dictionnaire Wikititle.

Longueur requête	Exacte (En)	1-erreur (En)	Exacte (Wi)	1-erreur (Wi)
2	0,003	0,5975	0,005	0,9819
3	0,0054	0,3052	0,0087	0,9652
4	0,005	0,355	0,0072	1,0334
5	0,0064	0,2933	0,007	0,6881
6	0,0059	0,2412	0,0078	0,6455

TABLE 6.9 – Le navigateur Opera 26. Le temps des requêtes + Top-k (en millisecondes) sur les dictionnaires Anglais (En) et les Titres de Wikipédia (Wi). La construction du Trie pour le dictionnaire Anglais est $1,5$ secondes et $6,2$ secondes pour le dictionnaire Wikititle.

Longueur requête	Exacte (En)	1-erreur (En)	Exacte (Wi)	1-erreur (Wi)
2	0,0471	1,64	0,0868	7,07
3	0,0492	1,93	0,0883	8,72
4	0,0608	2,37	0,1063	10,56
5	0,0896	3,29	0,1172	11,07
6	0,1013	3,59	0,1308	10,59

TABLE 6.10 – Le navigateur Internet Explorer 11. Le temps des requêtes + Top-k (en millisecondes) sur les dictionnaires Anglais (En) et les Titres de Wikipédia (Wi). La construction du Trie pour le dictionnaire Anglais est 30 secondes et 150 secondes pour le dictionnaire Wikititle.

Les résultats des quatre navigateurs sont beaucoup plus petits que $100ms$, cela veut dire que nos méthodes d'auto-complétion fonctionnent très bien dans tous les navigateurs.

Afin de mieux visualiser les résultats et afin d'avoir une idée de la performance de nos méthodes de l'auto-complétion exacte et approchée sur les différents navigateurs, nous avons fait une comparaison entre eux illustrée par les deux figures 6.12, 6.13 suivantes :

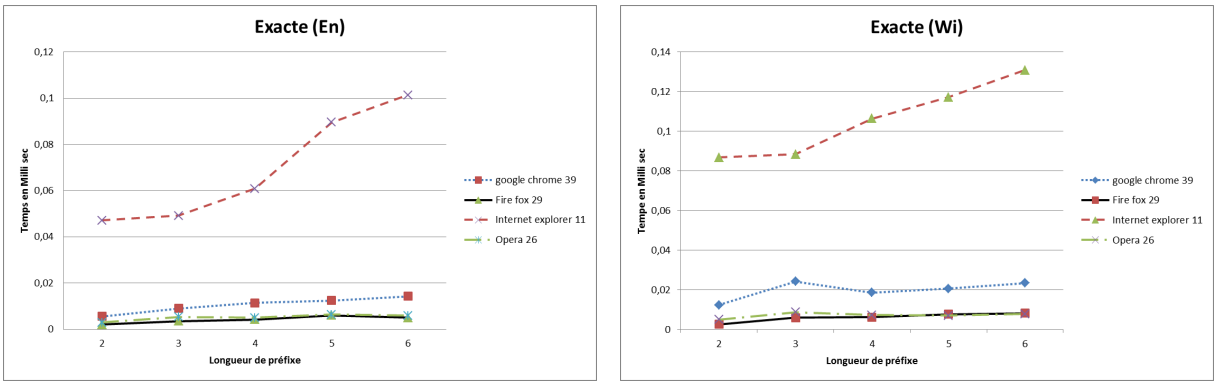

FIGURE 6.12 – Temps des 4 navigateurs pour une auto-complétion exacte avec les dictionnaires Anglais (En) et WikiTitle (Wi).

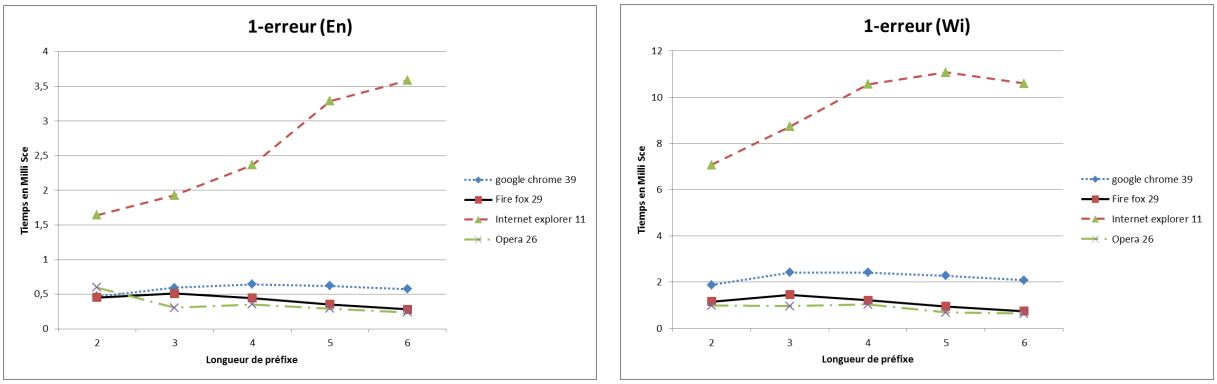

FIGURE 6.13 – Temps des 4 navigateurs pour une auto-complétion approchée (1-erreur) pour les dictionnaires Anglais (En) et WikiTitle (Wi).

Les quatre graphes présentés dans les deux figures (6.12, 6.13) illustrent bien la différence entre les quatre navigateurs et montrent que la meilleure performance a été obtenue avec *Opera 26* et *Firefox 29* qui sont presque au même niveau dans l'auto-complétion exacte, *Firefox* est légèrement plus lent dans l'auto-complétion approchée. Ces deux navigateurs sont suivis par *Google Chrome 39* (légèrement derrière), et enfin *Microsoft Internet Explorer 11*.

On remarque qu'avec les trois navigateurs (Chrome, Firefox et Opera) le temps ne change pas beaucoup avec le changement de la taille des requêtes. Par contre avec *Internet explorer 11* le temps augmente considérablement. Chrome, qui est un navigateur populaire, est seulement légèrement plus lent que les 2 plus rapides. Internet Explorer est le plus lent.

6.11 Conclusion

Dans ce chapitre, nous avons présenté une approche efficace pour résoudre le problème de l'auto-complétion approchée sous la contrainte de la distance d'édition dans une architecture client-serveur, et nous avons présenté la méthode Top-k pour améliorer la qualité des résultats en utilisant un système de classement dynamique et statique, selon l'importance (score) des mots.

Nous avons également présenté une méthode efficace d'élimination des résultats en double, ce qui est un réel problème dans les résultats de l'auto-complétion approchée.

Nous avons présenté une bibliothèque (nommée Appaco_lib), pour être utilisée soit du coté serveur ou du coté client ou les deux en même temps pour répondre rapidement à des requêtes d'auto-complétion approchée pour des dictionnaires `UTF-8` (donc toutes les langues).

Pour les détails de l'utilisation de notre bibliothèque voir le document qui explique toutes les fonctionnalités et les options proposées dans : `https://github.com/AppacoLib/api.appacoLib/tree/master/doc_appaco_lib`.

L'intérêt de cette étude, c'est d'avoir démontré que ce problème, bien qu'il soit un type particulier de la recherche approchée, ne pouvait pas être résolu de manière optimale par les solutions présentées dans les deux chapitres précédents. Nous avons ainsi étendu l'étude de la recherche approchée à un contexte différent qui nécessite une solution différente. Sachant que $|prefixe(m)| \leq |m|$, et que l'auto-complétion nécessite une deuxième étape pour trouver tous les suffixes, les solutions présentées dans les chapitres précédents ne pouvaient pas rester optimales.

Remarquons enfin que la recherche approchée pour laquelle nous avons présenté plusieurs solutions, se base sur une comparaison entre mots. Il s'agit fondamentalement, quelle

que soit la solution, de comparer la proximité entre mots. Il existe plusieurs fonctions mathématiques qui mesurent cette proximité, elles sont appelées fonction de distance.

Une fonction de distance $dist(x, y)$ détermine l'écart entre les mots x et y, c'est-à-dire le nombre d'opérations [1] minimum nécessaires pour transformer le mot x en y.

Si x et y sont des séquences biologiques (ex : ADN, ARN), le problème de l'alignement revient à la recherche de cette proximité entre x et y, et c'est donc un problème de recherche approchée dans lequel le nombre d'erreurs k est inconnu.

Trouver le meilleur alignement entre x et y revient à trouver le plus petit k, autrement dit, la meilleure façon de mettre x face à y avec un nombre minimum de différences.

Nous nous intéressons à ce nouveau problème de recherche approchée pour les séquences génomiques (des longues chaînes de texte) *l'alignement*, dans le chapitre qui suit.

1. Chaque fonction à ses propres opérations. Voir la définition dans le chapitre 2.

Chapitre 7

Alignement multiple de séquences d'ADN avec un nouvel algorithme *DiaWay*

7.1 Introduction

L'alignement des séquences biologiques est parmi les domaines d'application les plus importants de la recherche approchée de motifs. Les séquences biologiques (ADN, ARN, protéines) sont considérées comme de longues chaînes de caractères sur un alphabet spécifique, ($\Sigma_{ADN} = \{A, C, T, G\}$ pour l'ADN, $\Sigma_{ARN} = \{A, C, U, G\}$ pour l'ARN, et $\Sigma_{protéine} = \{20\ lettres\ d'acides\ aminés\}$ pour les protéines). L'alignement est un problème appartenant au domaine de la bio-informatique permettant de visualiser les ressemblances entre deux séquences ou plus et de déterminer leurs éventuelles homologies. Pour plus de détails sur les méthodes et les techniques d'alignement des séquences biologiques, référer à [10, 11, 125, 12].

L'alignement des séquences biologiques est la représentation de deux ou plusieurs séquences les unes sous les autres, afin de faire ressortir les régions similaires, plus précisément déterminer les différences et les similitudes.

Généralement pour mesurer le taux de similarité entre les séquences on utilise la distance d'édition où on a les trois opérations suivantes : la substitution (le A est substitué à T et le G est substitué à C), l'insertion et la suppression qui nécessitent en général l'introduction de *trous* (appelés *gaps*) à certaines positions dans les séquences, de manière à aligner les caractères communs sur des colonnes successives, voir la figure 7.1.

Le défi important pour la bio-informatique est de concevoir des algorithmes capables de trouver automatiquement des alignements biologiquement corrects.

Le problème d'alignement par paire a été largement considéré comme étant résolu [67, 68]. La plupart des efforts sont concentrés sur l'amélioration des algorithmes d'aligne-

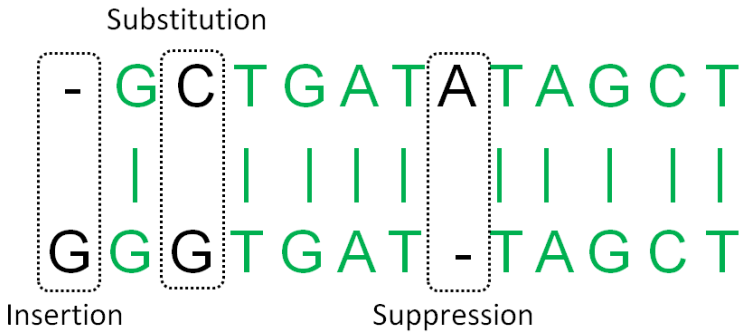

FIGURE 7.1 – Alignement de deux séquences d'ADN.

ments multiple, pour trouver des solutions optimales ou raisonnablement bonnes. L'alignement multiple correspond à une généralisation de l'alignement par paire, la recherche de similitude ne se fait pas qu'au niveau de deux séquences, mais au niveau de toutes les séquences. Il existe de nombreuses méthodes qui fournissent des solutions à ce problème [126, 127, 128, 129, 130, 131, 132].

Dans ce chapitre, nous apportons une solution au problème de l'alignement multiple. Notre solution est basée sur une méthode d'alignement connue *DIALIGN* [127, 133], qui est un algorithme conçu pour l'alignement par paire, ainsi que l'alignement multiple des séquences d'acides nucléiques et de protéines.

Notre nouvelle approche (appelée *DiaWay*) s'exécute selon les étapes suivantes : après extraction des diagonales de la matrice dot plot, au lieu de supprimer automatiquement les diagonales inconsistantes, on essaye d'aligner les sous-diagonales (fragments) consistantes de celles-ci afin d'améliorer la qualité de l'alignement. L'étape d'indexation des diagonales, où nous créons une structure d'index pour accélérer l'accès aux différents fragments des diagonales sera suivie par la vérification de l'inconsistance. L'étape de tri des diagonales consiste à trier toutes les diagonales consistantes pour l'alignement final, en fonction de leurs positions dans les séquences biologiques. La dernière étape consiste simplement à rassembler les fragments des diagonales et ensuite insérer des gaps (le symbole −) dans les vides.

Les expérimentations, comparées à *DIALIGN*, montrent que notre approche *DiaWay* donne de meilleurs résultats dans les deux cas d'alignement d'ADN par paire et multiple.

Le travail présenté dans ce chapitre a été publié en 2015 dans deux articles [18, 19], où nous avons présenté deux implémentations de notre approche *DiaWay 1.0*, *DiaWay 2.0*.

Ce travail de doctorat est l'aboutissement d'un premier travail réalisé en master [134], qui a été poursuivi pendant la durée du doctorat en cherchant à améliorer notre approche et faire les différents tests nécessaires afin d'arriver à une version performante qui fait un

alignement efficace et rapide. Ce travail a abouti à la publication de deux articles scientifiques, ainsi qu'à l'implémentation de deux versions logiciels.

Le chapitre est organisé comme suit : La section 7.2 couvre le concept de l'approche originale de *DIALIGN*, et une brève explication de ses étapes. Les étapes de notre nouvelle approche *DiaWay* de l'extraction des diagonales au processus d'alignement final sont présentées dans les sections 7.3, 7.4, 7.5, 7.6 et 7.7. La section 7.8 portes sur les tests, les comparaisons, et les analyses. Et enfin, une conclusion est donnée dans la section 7.9.

7.2 DIALIGN

DIALIGN [127, 133], est un algorithme connu qui fait un alignement par paire et multiple des séquences biologiques. Il combine les caractéristiques d'alignement locales et globales. La méthode consiste à aligner les similitudes de paires locales. L'alignement final est composé d'une collection de diagonales qui répondent à un certain critère de consistance puis sélectionne un ensemble consistant de diagonales avec une somme maximale de poids.

Les grandes lignes de l'algorithme d'alignement multiple *DIALIGN* sont :

1. Construire une matrice de similarité pour chaque paire des séquences afin de récupérer toutes les diagonales possibles. Une diagonale (également appelé fragment ou bloc) est définie par des régions similaires appartenant aux deux séquences biologiques.
2. Donner pour chaque diagonale un score puis les trier selon leurs scores et selon leurs degré de chevauchement avec les autres diagonales. La liste de diagonales est ensuite utilisée pour assembler un alignement multiple d'une manière gloutonne (gourmande).
3. Les diagonales sont intégrées une par une dans l'alignement multiple en commençant par la diagonale de poids (score) maximum. Avant d'ajouter une diagonale à l'alignement, on doit vérifier si elle est consistante avec les diagonales déjà intégrées.
4. Dans une étape finale, le programme introduit des gaps (-) dans les séquences jusqu'à ce que tous les résidus liés par les diagonales sélectionnées soient correctement disposées.

Le poids d'une diagonale, et le score d'alignement : *DIALIGN* essaye de sélectionner un ensemble consistant de diagonales avec une somme maximale de poids. Les diagonales d'un poids élevé sont plus susceptibles d'être biologiquement pertinentes, et, par conséquent, sont ajoutées d'abord à l'alignement. La qualité des alignements produits par la méthode *DIALIGN* dépend de la manière dont le score ou le poids des diagonales

est définis. *DIALIGN* a défini un nouveau système de score qui est différent des autres approches traditionnelles.

Le premier système de score est défini dans [127, 135], dans cette version *DIALIGN 1*, la dépendance sur le seuil T (T représente une valeur de seuil positive ou nulle qui rentre dans le processus du calcul des poids des diagonales), était un sérieux inconvénient, car il n'y avait pas de règle générale pour choisir la valeur T qui permet de produire des alignements de qualité. En plus, la longueur minimum requise d'une diagonale était arbitraire. Dans la 2e version de Dialign (*DIALIGN 2*) Morgenstern et al. [133] introduisent une nouvelle fonction de poids pour les diagonales afin de remédier au problème de seuil T et la taille minimale de diagonale.

La consistance des diagonales avec la méthode DIALIGN : Un concept crucial et délicat décrit dans *DIALIGN*. Comment décider si une diagonale est consistante avec les diagonales déjà intégrées dans l'alignement ou non. Une collection de diagonales est dite consistante s'il n'y a pas un double conflit ou croisement des résidus des fragments des diagonales [127]. De même, pour les diagonales impliquant la même paire de séquences ne sont pas autorisées à avoir un chevauchement. Voir la figure 7.2.

FIGURE 7.2 – L'inconstance des diagonales, d_1 se croise/chevauche avec d_0

Une diagonale qui traverse (se croise) ou se chevauche avec une autre diagonale qui a un poids plus élevé, est une diagonale inconsistante, parce qu'elle ne permet pas un alignement correct avec cette diagonale qui a un poids plus élevé. À chaque fois qu'on ajoute une diagonale à l'alignement, on vérifie si elle est consistante avec les diagonales déjà intégrées. Les diagonales qui ne sont pas consistantes avec l'ensemble croissant des diagonales consistantes seront rejetées et donc supprimées. Trouver un ensemble de diagonales consistantes est un problème NP-complet [136].

7.3 L'extraction des diagonales

Soit S l'ensemble de toutes les séquences biologiques, et $S = \{s_1, s_2, ..., s_l\}$ où l est le nombre total de toutes les séquences, $s_k \in S$ où $k \in [1..l]$.

Soit D l'ensemble de toutes les diagonales, $D = \{d_1, d_2, ..., d_n\}$ où n est le nombre total de toutes les diagonales initiales. Soit D' l'ensemble des diagonales consistantes seulement, $d_j \in D'$ où $j = [1..m]$ et m est le nombre total de toutes les diagonales consistantes seulement. Chaque diagonale comporte une paire de fragment (sous-séquence), le

premier segment est désigné par x, et l'autre par y. Pour désigner une diagonale et ses deux fragments, on écrit $d_i = (x_i, y_i)$ où $i \in [1..n]$.

Les diagonales sont extraites à partir d'une matrice dot plot construite pour chaque paire de séquences, tels que les éléments $[i, j]$ de cette matrice sont remplis par 1 dans le cas de correspondance entre les caractères de la position i et j dans les deux séquences respectivement, et 0 dans le cas contraire.

Une diagonale est définie par des régions similaires appartenant à deux séquences. Elle est représentée par une paire de fragments (sous-séquences) d'une taille égale et qui contiennent des similitudes tout en acceptant la mutation c'est-à-dire : concordance entre A et T et entre C et G.

Soient S_1 et S_2 deux séquences d'ADN, et soit M leur matrice dot plot. La diagonale d est donc comme suit $d = \{M[i_1][j_1], M[i_2][j_2], \ldots, M[i_m][j_m]\}$, où m est la longueur de cette diagonale, et $i \in [1..|S_1|]$ et $j \in [1..|S_2|]$. Comme nous l'avons dit précédemment, une diagonale est désignée par ses deux fragments et on écrit $d_i = (x_i, y_i)$ où $i \in [1..n]$.

Exemple : Soient deux séquences d'ADN (de petites longueurs) $S = \{s_1, s_2\}$, avec $s_1 = ATTCCGACT$ et $s_2 = AATTCGCGT$. Le résultat de la construction de la matrice dot plot est illustré dans la figure 7.3.

	A	T	T	C	C	G	A	C	T
A	1	**0**	0	0	0	0	1	0	0
A	**1**	0	**0**	0	0	0	1	0	0
T	0	**1**	1	0	0	0	0	0	1
T	0	1	**1**	0	0	0	0	0	1
C	0	0	0	**1**	1	0	0	1	0
G	0	0	0	**0**	**0**	1	0	0	0
C	0	0	0	1	**1**	**0**	0	1	0
G	0	0	0	0	0	**1**	0	0	0
T	0	1	1	0	0	0	**0**	0	1

FIGURE 7.3 – La matrice de similarité dot plot de deux séquences d'ADN.

Dans la figure 7.3, nous donnons comme exemple les trois diagonales qui sont entourées par des ellipses :

$d_1 =$ TT
AA

$d_2 =$ ATTCCG
ATTCGC

$$d_3 = \begin{matrix} \text{CCGA} \\ \text{GCGT} \end{matrix}$$

Nous avons remarqué que lorsqu'on supprime les diagonales inconsistantes, même si c'est seulement une partie de leurs résidus qui cause le problème de l'inconsistance, on perd dans la qualité de l'alignement. Et nous avons remarqué qu'on peut aligner les parties des diagonales (les sous-diagonales) qui ne posent pas de problème d'inconsistance pour améliorer le résultat de l'alignement final. Pour cela, nous procédons comme suit :

7.3.1 Méthode 1

Lorsqu'on extrait les diagonales, on extrait aussi leurs sous-diagonales, qui sont elles-mêmes considérées comme des diagonales. Car une diagonale peut être inconsistante, tandis que l'une de ses sous-diagonales est consistante, et donc ses sous-diagonales vont avoir une chance d'être alignées.

Pour extraire les sous-diagonales, à chaque fois, on enlève les premières paires de caractères des deux fragments qui constituent la diagonale, donc on enlève le premier caractère gauche du premier fragment et on fait la même chose pour le deuxième fragment, et dans les deux fragments, il reste leurs suffixes. Les deux nouveaux sous-fragments vont constituer une nouvelle sous-diagonale. Tous les suffixes d'une diagonale d_i sont considérés comme des sous-diagonales. On prend seulement les suffixes qui ont une longueur supérieure ou égale à la longueur minimale de la diagonale permise.

Exemple : Soit la longueur minimale des diagonales égales à 2, et soit la diagonale d_2 (de l'exemple précédent) suivante :

$$d_2 = \begin{matrix} \text{ATTCCG} \\ \text{ATTCGC} \end{matrix}$$

Donc ses sous-diagonales sont extraites comme suit :

$$\{d_{2.1}, d_{2.2}, d_{2.3}, d_{2.4}\} = \begin{matrix} \text{TTCCG} , & \text{TCCG} , & \text{CCG} , & \text{CG} \\ \text{TTCGC} , & \text{TCGC} , & \text{CGC} , & \text{GC} \end{matrix}$$

Dans le processus de vérification de l'inconsistance d'une diagonale d_i, si on trouve qu'elle est consistante, on supprime directement toutes ses sous-diagonales de l'ensemble D. Car la diagonale principale est consistante, donc on n'a pas besoin de ses sous-diagonales. En plus, toutes ses diagonales sont consistantes à leur tour, et leur fragments sont inclus dans la diagonale principale.

Les diagonales sont indexées en fonction de leurs numéros de séquences afin de permettre un accès rapide pour les trouver. Voir la section 7.4. Cette méthode est utilisée dans notre première implémentation appelée *DiaWay* 1.0.

7.3.2 Méthode 2

Il s'est avéré qu'avec la méthode précédente, le nombre de diagonales devient très importants, et cela influx sur le temps de calcul, en particulier pendant le processus de vérification de l'inconsistance des diagonales.

Pour cette raison, dans la deuxième version de notre approche, nous utilisons la méthode expliquée dans [137], au lieu d'extraire toutes les sous-diagonales, on supprime seulement les parties (les résidus) inconsistantes à partir des deux fragments de la diagonale principale, et on donne aux sous-fragments restants une autre chance dans le processus de sélection des diagonales consistantes. On calcule le poids de la partie restante (la sous-diagonale restante), et on la place dans la liste des diagonales qui restent à vérifier.

Cette seconde méthode est utilisée dans la version appelée *DiaWay* 2.0.

7.4 L'indexation des diagonales

Certaines étapes de notre approche nécessitent la recherche des diagonales et donc un accès rapide à ces derniers. Pour remédier à ce problème, on construit une structure d'index (une matrice) qui contient les numéros des diagonales des fragments en fonction de leurs appartenances aux séquences biologiques.

On commence par la construction de la structure de données, on utilise un tableau de pointeur qui pointe vers des tableaux avec des tailles différentes. La taille de chaque tableau est le nombre maximal des fragments des diagonales possibles dans la séquence qui porte le numéro de ce tableau. On peut aussi utiliser une simple matrice, si le nombre de fragments dans chaque ligne est approximativement le même.

La deuxième étape consiste à remplir notre index. Au moment de l'extraction des diagonales, pour chaque paire de séquences (s_i, s_j) avec $i \neq j$, lorsqu'on extrait une diagonale d_k, on ajoute ses deux fragments à l'index, le premier fragment de la séquence s_i dans le tableau(ou la ligne numéro) i, et le deuxième fragment dans le tableau (ou la ligne numéro) j.

Lorsqu'on extrait les diagonales, si la méthode d'extraction ne permet pas d'obtenir les fragments dans l'ordre d'appartenance à la séquence (par exemple l'extraction des diagonales de la partie supérieure de la diagonale principale de la matrice (M[0,0] M[|t|][|t|] avec $t = min(|s_2|, |s_1|)$), ensuite, la partie inférieure). Dans ce cas, on doit trier les fragments qui sont dans la même ligne (tableau) de notre index, selon l'indice de leur début dans la séquence biologique.

Exemple :
Soit les diagonales suivantes $D = \{d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7\}$, ordonnées selon leur poids. Les fragments des diagonales sont répartis sur les quatre séquences biologiques comme il

est montré dans la figure 7.4.

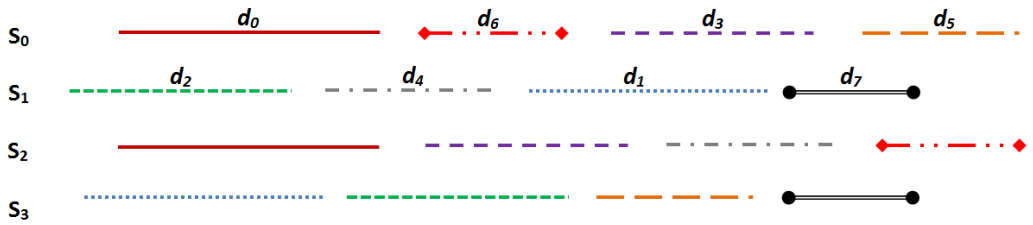

FIGURE 7.4 – Les fragments des 7 diagonales répartis sur les 4 séquences biologiques.

Le résultat de la construction de l'index est illustré dans le tableau 7.1 suivant (dans ce simple exemple, on utilise une matrice).

S_0	$d_{0.1}$	$d_{6.1}$	$d_{3.1}$	$d_{5.1}$
S_1	$d_{2.1}$	$d_{4.1}$	$d_{1.1}$	$d_{7.1}$
S_2	$d_{0.2}$	$d_{3.2}$	$d_{4.2}$	$d_{6.2}$
S_3	$d_{1.2}$	$d_{2.2}$	$d_{5.2}$	$d_{7.2}$

TABLE 7.1 – Les diagonales $d_0, d_1, d_2, d_3, d_4, d_5, d_6, d_7$ indexées selon les positions de leurs fragments dans les séquences.

Si on veut obtenir les diagonales situées sur les mêmes séquences que d_0, il suffit d'accéder directement à l'index de $d_0(S_0, S_2)$ qui correspond aux séquences 0 et 2, et dans ce cas on trouve les diagonales d_3 et d_6.

La méthode de recherche consiste à aller à la première ligne portant le premier fragment et enregistrer les numéros de toutes les diagonales trouvées dans cette ligne. Ensuite, aller à la deuxième ligne qui contient le deuxième fragment de la diagonale, et on cherche s'il y a des numéros de diagonales déjà enregistrées dans la première ligne.

Pour une raison d'optimisation du processus précédent, on utilise un simple tableau de bits, pour vérifier l'intersection entre les numéros des diagonales qui sont dans les deux lignes de la matrice comme suit : On utilise un tableau de bits initialisé à 0, la longueur de ce tableau de bits L est le nombre total des diagonales. On prend la première ligne de la matrice et on marque toutes les cases, correspondantes aux numéros des diagonales, dans le tableau de bits par un 1. On prend la deuxième ligne de la matrice, et pour chaque numéro de diagonale, dans la position de tableau de bits représenté par ce numéro, on vérifie simplement si on a 1. Si oui, alors la diagonale correspondante est prise comme une solution.

L'étape de l'indexation des diagonales est utilisée :

– Dans le processus de l'inconsistance simple. Voir la sous-section 7.5.3.
– Pour trier les diagonales pendant le processus d'alignement. Voir la section 7.6.

7.5 Une nouvelle méthode pour la consistance des diagonales

La consistance des diagonales est un problème difficile, elle est un concept crucial dans l'algorithme *DIALIGN*. La diagonale qui se croise ou chevauche avec une diagonale consistante qui a un poids plus élevé est considérée comme une diagonale inconsistante [127], voir la figure 7.2, et cela, parce qu'elle ne permet pas aux diagonales avec un poids plus élevé d'être alignées correctement.

L'approche originale [127] vérifie la consistance d'une diagonale avec les diagonales déjà incorporées afin de l'ajouter à l'alignement. Nous avons proposé un concept totalement différent du concept original. Contrairement à l'algorithme *DIALIGN*, nous essayons de prouver l'inconsistance des diagonales en utilisant une modélisation basée sur les graphes. Notre approche prouve qu'une diagonale est inconsistante avec les diagonales déjà testées et acceptées comme consistantes.

Dans notre travail, on utilise un graphe orienté, on considère les débuts des fragments des séquences x_i ou y_i comme des nœuds (sommets) du graphe, et les arcs correspondent à la relation entre le début d'un fragment d'une diagonale $d_i = (x_i, y_i)$ avec le début d'un autre fragment (x_j *ou* y_j) d'une diagonale d_j située sur la même séquence, où la position de début du fragment de d_i est juste avant celui de d_j. Ainsi que la relation de début du fragment x_i de la diagonale d_i vers l'autre fragment y_i de la même diagonale (ou vice versa).

Le graphe orienté $G = (V, E)$ est défini comme suit :

$V = \{x_1, x_2, ..., x_n\} \cup \{y_1, y_2, ..., y_n\}$ où x_i et y_i : le début du 1er et 2^{e} fragment de la diagonale d_i , et $i \in [0..n]$

$E = E_1 \cup E_2 \cup E_3 \cup E_4 \cup E_5$
$E_1 = \{(x_i, y_i), (y_i, x_i) \mid i \in [0..n]\}$
$E_2 = \{(x_i, x_j) \mid i, j \in [0..n]$ et $\exists k$ où $x_i, x_j \in s_k$ et x_j le premier fragment qui apparait après x_i, et s_k le numéro de la séquence biologique k $\}$
$E_3 = \{(x_i, y_j) \mid i, j \in [0..n]$ et $\exists k \mid x_i, y_j \in s_k$ et y_j le premier fragment qui apparait après x_i $\}$
$E_4 = \{(y_i, x_j) \mid i, j \in [0..n]$ et $\exists k \mid y_i, x_j \in s_k$ et x_j le premier fragment qui apparait après y_i $\}$
$E_5 = \{(y_i, y_j) \mid i, j \in [0..n]$ et $\exists k \mid y_i, y_j \in s_k$ et y_j le premier fragment qui apparait après y_i $\}$

Exemple : on a 4 diagonales (8 fragments), situés sur 3 séquences biologiques différentes (S_1, S_2 et S_3).

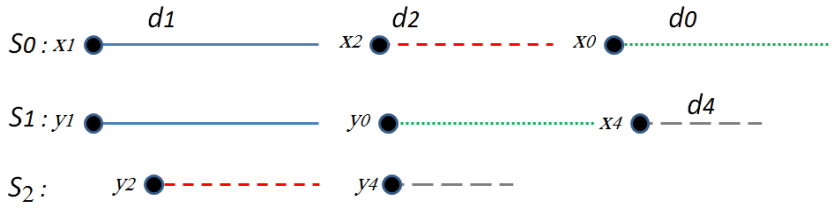

FIGURE 7.5 – Quatre diagonales et leurs fragments.

Donc le graphe sera $G(8, 13)$ avec 8 sommets et 13 arcs, voir la figure 7.6 :

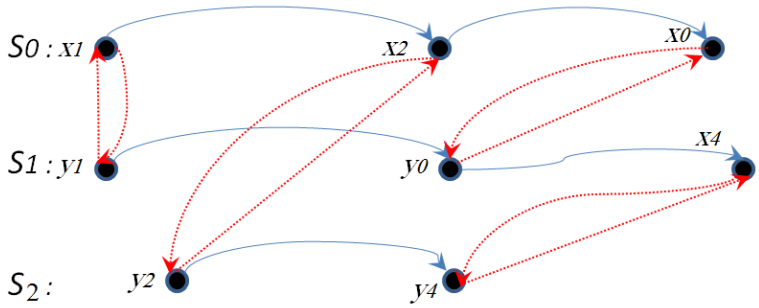

FIGURE 7.6 – Graphe de consistance $G(8, 13)$. Les arcs en pointillés (les arcs verticaux avec la couleur rouge) représentent l'ensemble E_1, les autres arcs horizontaux représentent les ensembles E_2, E_3, E_4 et E_5.

7.5.1 L'inconsistance d'une diagonale avec un chemin simple

La diagonale d_i est appelée inconsistante avec toutes les diagonales déjà acceptées dans D', si et seulement s'il y a un chemin simple [1] qui connecte x_i et y_i passant à travers les fragments des diagonales consistantes dans D'. ($x_i \longrightarrow y_i$ ou $x_i \longleftarrow y_i$). En d'autres termes, pour prouver que d_i est une diagonale inconsistante avec D' on doit trouver un chemin simple λ entre x_i et y_i avec $\lambda = x_0u_1x_1...x_{k-1}u_kx_k$ où ($x_0 = x_i$ et $x_k = y_i$) ou ($x_0 = y_i$ et $x_k = x_i$).

L'exemple suivant illustre ce concept. Soit la diagonale $d_5(x_5, y_5)$ dans la figure 7.7, pour vérifier l'inconsistance de cette diagonale, on essaye de trouver un chemin simple qui passe à travers les fragments des diagonales consistantes dans D' ($D' = \{d_0, d_1, d_2, d_3, d_4\}$).

1. Le chemin est un graphe dirigé $G = (V, E)$. C'est une séquence alternative de sommets et d'arcs : $\lambda = x_0u_1x_1...x_{k-1}u_kx_k$ où $i \in [1..k]$, et le sommet x_i est le bout initial de l'arc u_i et le sommet x_{i+1} est son bout final. λ est un chemin de x_0 à x_k de longueur k. Dans un chemin simple tous ses arcs sont distincts l'un de l'autre [138].

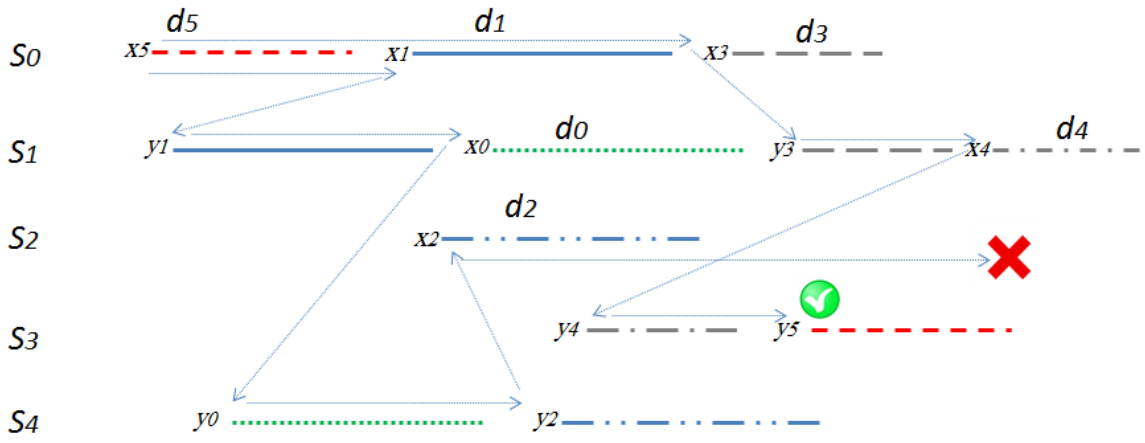

FIGURE 7.7 – Un chemin simple entre les deux fragments de la diagonale d_5 qui passe à travers les fragments des diagonales consistantes ($\lambda = x_5\, x_1\, x_3\, y_3\, x_4\, y_4\, y_5$).

Dans la figure 7.7, on trouve un chemin simple qui passe à travers les fragments des diagonales consistantes ($\lambda = x_5\, x_1\, x_3\, y_3\, x_4\, y_4\, y_5$), donc la diagonale d_5 est considérée comme inconsistante, et par conséquent elle est supprimée de la liste des diagonales D. Lorsqu'on essaye de trouver un chemin simple, on peut avoir plusieurs possibilités. Dans cette figure (figure 7.7), on peut voir qu'on a deux chemins, le 1er mène à la solution et l'autre pas. On appel cette méthode *inconsistance de chemin*.

Pour faire la recherche de chemin simple dans notre graphe orienté, on considère que la racine est le sommet représenté par le premier fragment de la diagonale d_i, ça peut être soit x_i ou y_i. Le but est de trouver l'autre sommet dans le graphe, c'est-à-dire si la racine est x_i, on essaye de trouver y_i et vice versa. La recherche de chemin simple utilise l'algorithme de *recherche en profondeur d'abord*, pour plus de détails sur cet algorithme voir [139].

Dans notre graphe orienté, deux branches différentes peuvent converger vers un même sommet, voir la figure 7.8. Cela signifie qu'un sous-graphe peut être vérifié plus d'une seule fois inutilement, pour cela on doit s'assurer que dans la phase de recherche, le chemin passe qu'une seule fois par un sommet donné. Donc si dans la recherche on arrive à un sommet qui a été déjà vérifié, on arrête, pour ne pas vérifier le sous-graphe plusieurs fois.

7.5.2 L'inconsistance avec le chemin simple trouvé

Lors de l'étape de recherche de chemin simple, pour vérifier l'inconsistance d'une diagonale d_i, si ce chemin existe, toutes les diagonales dans D qui appartiennent à ce chemin dans la direction spécifiée sont supprimées. voir la figure 7.9.

Le chemin simple trouvé, pour prouver l'inconsistance de la diagonale d_3, est utilisé pour éliminer les diagonales d_4 et d_5 qui appartiennent à ce chemin et suivent le bon sens.

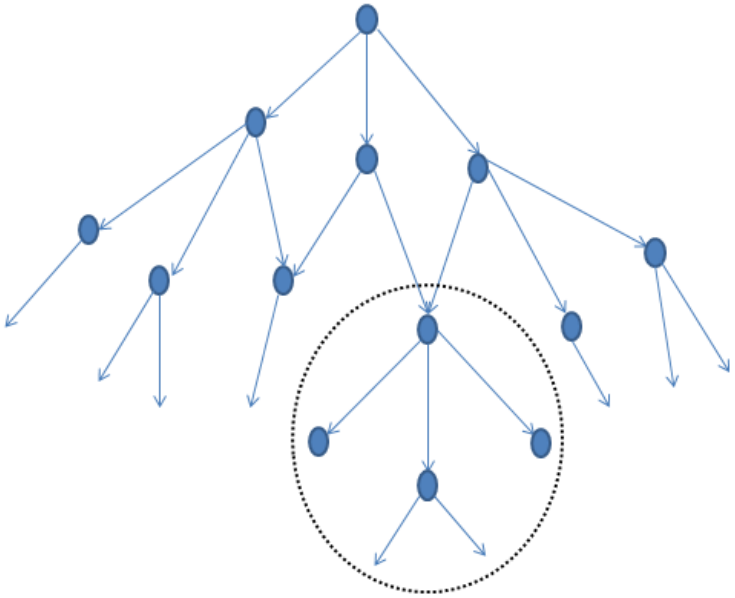

FIGURE 7.8 – Le graphe de recherche d'un chemin simple. Le sous-graphe entouré, n'est vérifié qu'une seule fois.

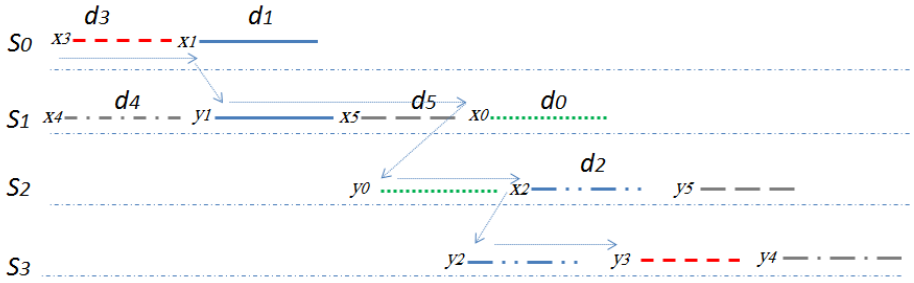

FIGURE 7.9 – Les diagonales d_4 et d_5 appartiennent au chemin simple.

Avec cette technique on évite de vérifier ces diagonales avec la recherche de chemin simple qui peut prendre beaucoup de temps.

7.5.3 L'inconsistance simple

La première méthode *inconsistance de chemin* (voir la sous-section 7.5.1) suffit à elle seule pour éliminer toutes les diagonales inconsistantes, mais avec cette approche la recherche peut emprunter plusieurs chemins avant de trouver le bon, et cela implique un temps d'exécution très élevé. Pour cela, on a choisi d'employer une seconde méthode appelée *inconsistance simple* qui permet d'éliminer les inconsistances entre deux diagonales situées dans les mêmes séquences, et cela permet de réduire considérablement le temps d'exécution.

L'algorithme prend une diagonale consistante d_i ($d_i \in D^{'}$) et la compare avec la dia-

gonale d_j où $d_j \in D$), si on trouve que leurs sous-séquences se trouvent dans les mêmes séquences et d_j se croise ou se chevauche avec la diagonale d_i alors d_j est inconsistante, donc on la supprime directement de D. On vérifie toutes les diagonales qui se situent sur les mêmes séquences de d_i.

Exemple :
On considère deux séquences biologiques $S = \{s_0, s_1\}$, et soit $d_0 = (x_0, y_0)$ une diagonale consistante, et soit les 3 diagonales $D = \{d_1(x_1, y_1), d_2(x_2, y_2), d_3(x_3, y_3)\}$ qui ne sont pas encore vérifiées. Tous leurs fragments sont sur les mêmes séquences. voir la figure 7.10.

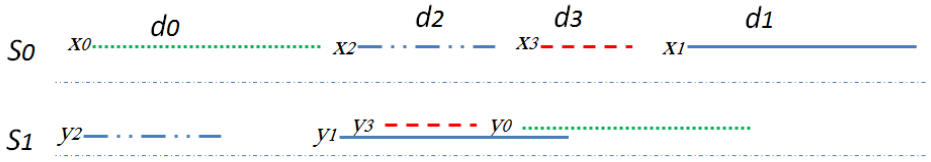

FIGURE 7.10 – Inconsistance simple : les diagonales d_1, d_2 se croisent avec d_0 et d_3 se chevauche avec d_0.

Dans la figure 7.10, on voit clairement que les diagonales d_1, d_2 se croisent avec d_0 et d_3 se chevauche avec d_0, donc toutes ces diagonales d_1, d_2, d_3 sont inconsistantes et seront donc supprimées.

7.5.4 Chemin vertical

Un chemin est dit vertical si et seulement s'il ne passe que par les sommets (les fragments) d'une même diagonale d_i, et si on a plus d'une diagonale, dans ce cas leurs fragments doivent partager le même sommet (c'est-à-dire leurs fragments sont dans les mêmes séquences et ont le même début).

Les diagonales qui sont reliées par un chemin simple vertical ne sont pas considérées comme des diagonales inconsistantes, car elles ne causent pas de problème pour l'alignement. Pour cela, on ne les supprime pas, on les garde pour les utiliser dans la suppression des autres diagonales avec la méthode *inconsistance simple*.

Soit l'exemple dans la figure 7.11 suivante.

Selon la méthode *inconsistance de chemin* (voir la sous-section 7.5.1), la diagonale d_3 est une diagonale inconsistante car il y a un chemin simple entre ses deux fragments, donc on la supprime, même si elle ne cause pas de problème pour l'alignement, d_3 sera alignée par transitivité lorsque les diagonales d_0, d_1 et d_2 seront alignées.

Mais, afin de supprimer les diagonales inconsistantes avec d_3 en utilisant la méthode *inconsistance simple* (voir la sous-section 7.5.3), on doit la garder, et l'ajouter à l'ensemble des diagonales consistantes D'.

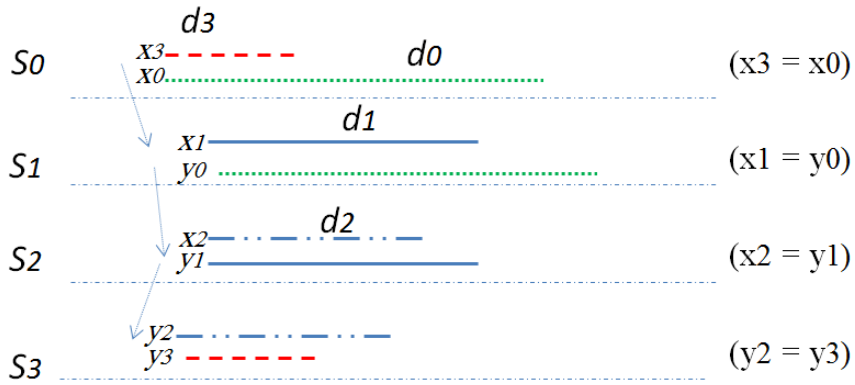

FIGURE 7.11 – Un chemin simple vertical qui relie les deux fragments (x_3, y_3) de la diagonale d_3.

7.5.5 Le processus du traitement de l'approche de l'inconsistance

Étape I : initialisation de l'ensemble des diagonales consistantes $D^{'}$:

- On ajoute la première diagonale d_0 à $D^{'}$. La première diagonale d_0 a le plus grand poids dans l'ensemble D, donc elle est automatiquement consistante.
- On utilise la méthode *inconsistance simple* pour supprimer les diagonales de D qui sont inconsistantes simple avec d_0.
- On ajoute la première diagonale dans D (appelée d_1) à l'ensemble consistant $D^{'}$. On a prouvé que la diagonale d_1 est consistante à l'aide de la méthode *inconsistance simple*, par conséquent d_1 est consistante avec d_0, maintenant $D^{'} = \{d_0, d_1\}$.
- On fait appel à la méthode *inconsistance simple* pour supprimer toutes les diagonales de D qui sont inconsistantes simple avec d_1.

Étape II : Traiter toutes les diagonales inconsistantes restantes dans D :

- Trouver une diagonale consistante :

  Appliquer la méthode *inconsistance de chemin* (voir la sous section 7.5.1), pour éliminer toutes les diagonales inconsistantes dans D avec toutes les diagonales de $D^{'}$, une par une, jusqu'à ce qu'on trouve une diagonale consistante avec les diagonales déjà acceptées (qui sont dans $D^{'}$), ou on arrive à la fin de l'ensemble D, et cela veut dire qu'on a supprimé toutes les diagonales de D.

  Dans cette méthode, lorsqu'on trouve un chemin simple entre les deux fragments d'une diagonale d_i on la supprime. Toutes les diagonales qui sont dans ce chemin simple sont supprimées en utilisant la méthode *inconsistance avec chemin trouvé*, voir la sous section 7.5.2.

  Lorsqu'on fait la vérification d'une diagonale d_i avec la méthode *inconsistance de chemin* et on ne trouve pas de chemin simple entre ces deux fragments (x_i, y_i), cela signifie que cette diagonale est consistante, donc on l'ajoute à l'ensemble $D^{'}$.

– Appliquer la méthode *inconsistance simple* :
On applique la méthode *inconsistance simple* pour supprimer toutes les diagonales de D qui sont inconsistantes simple avec d_i trouvé dans l'étape précédente.

– Répéter ces deux étapes précédentes, jusqu'à traiter toutes les diagonales qui sont dans D. À la fin, D' ne contiendra que des diagonales consistantes, et l'ensemble D sera vide.

7.6 Le tri des diagonales

Après la phase de la vérification de la consistance des diagonales, voir la section 7.5, où on supprime toutes les diagonales inconsistantes qui posent un problème pour l'alignement, on obtient un ensemble contenant uniquement des diagonales consistantes. L'alignement est effectué sur les diagonales restantes, de façon à ce que chaque fragment est placé en face de l'autre fragment de la même diagonale.

La méthode triviale pour aligner les diagonales restantes (placer les 2 fragments de chaque diagonales l'un en face de l'autre) est la suivante :

7.6.1 Méthode 1

Les diagonales restantes sont triées en fonction de leur poids dans l'ensemble final. Pour les aligner, on commence par la première diagonale dans la liste (celle qui a le plus grand poids), et on met ses deux fragments ($d_i = (x_i, y_i)$), l'un devant l'autre. Si le début du fragment x_i est, avant le début du fragment y_i, alors x_i est déplacé vers y_i, et vice versa. Pour déplacer le fragment (x_i/y_i), toute la sous-séquence qui commence à partir de ce fragment doit être déplacée.

Lorsqu'on passe à la prochaine diagonale (d_j avec $j > i$) pour rassembler ses fragments, le déplacement des fragments de cette diagonale d_j peut détériorer l'alignement des diagonales déjà alignées. Pour cela, on commence d'abord par vérifier si les fragments des diagonales déjà alignées sont déplacées (leurs positions ont été changées et leur alignement est détérioré) afin de ré-effectuer à nouveau leur alignement. Ce processus est répété pour toutes les diagonales déjà alignées une par une. Par ailleurs, pour l'ensemble de toutes les diagonales, chaque fois qu'une nouvelle diagonale d_j est alignée, on doit vérifier de manière récurrente pour chaque diagonale d_k ($k \in [1..j]$), toutes les diagonales précédentes dans l'alignement ($[1..k]$), jusqu'à arriver à la première diagonale d_1.

Cette méthode est très couteuse en matière de temps d'exécution, car pour chaque diagonale alignée, on doit vérifier toutes les diagonales précédentes ajoutées dans le processus d'alignement (l'étape finale qui met ensemble (regroupe) les fragments des diagonales).

Exemple : Si on considère qu'on est dans la 5e diagonale, la diagonale alignée d_j ($j = 5$), donc le nombre de vérifications est comme suit :

$5+4+3+2+1$
$+4+3+2+1$
$+3+2+1$
$+2+1$
$+1$

Le nombre de vérifications à faire pour une seule position est donné par la formule suivante : $\sum_{k=1}^{k=j} \frac{k(k+1)}{2}$ [1].

Le nombre total d'opérations pour vérifier n diagonales est : $\sum_{j=1}^{j=n} \sum_{k=1}^{k=j} \frac{k(k+1)}{2}$ dont la complexité temporelle est de l'ordre $\Omega(n^4)$.

7.6.2 Méthode 2

La méthode précédente ayant une complexité temporelle élevée, nous avons proposé une nouvelle méthode qui est plus efficace et qui peut assurer l'alignement final seulement en $O(k \times n)$, où n est le nombre de diagonales consistantes restantes, et k est le nombre de séquences biologiques à aligner.

Cette méthode consiste à trier toutes les diagonales consistantes pour l'alignement final, en fonction de leurs positions dans les séquences biologiques. La condition de vérification pour le tri n'est pas triviale, car les fragments des diagonales ne sont pas situés dans les mêmes séquences biologiques, pour cette raison, on applique la méthode suivante pour les trier :

1. L'indexation des fragments des diagonales en fonction des numéros des séquences de leurs fragments, comme nous l'avons déjà expliqué, dans la section 7.4.

2. Construire progressivement l'ensemble final des diagonales pour l'alignement de telle sorte que : la diagonale du premier tour est celle dont ses deux fragments n'ont pas de fragments qui les précèdent dans la même séquence. C'est-à-dire les fragments qui sont situés les premiers dans les séquences biologiques. Après cela, on supprime les fragments de la structure d'index, et on ajoute leurs diagonales à la liste triée. Ce processus est répété jusqu'à ce que tous les fragments soient supprimés et leurs diagonales sont ajoutées à la liste triée.

1. Il y a une petite erreur dans notre article [19]. Nous avons utilisé la formule $\sum_{k=1}^{j-1} j + (j-k)^{k+1}$ au lieu de la formule $j + \sum_{k=1}^{j-1}(j-k)(k+1)$, et bien sur cette erreur a fait que la complexité temporelle obtenue qui est de l'ordre de $O(n^n)$ est fausse. Dans cette thèse, nous obtenons donc la formule $\sum_{k=1}^{k=j} \frac{k(k+1)}{2}$ qui est égale à $j + \sum_{k=1}^{j-1}(j-k)(k+1)$.

Pour obtenir les fragments satisfaisant ces conditions on procède comme suit : on parcourt la première colonne de la matrice ($i \in [1..k]$), k est le nombre des lignes de la matrice (en d'autre terme, on parcourt la première case des lignes de la matrice ou la structure d'index ligne après ligne). On marque le premier fragment d'une diagonale à la ligne i. Si le second fragment situé à la ligne j (où $j > i$) est situé en premier sur sa ligne, la condition est satisfaite. Dans le cas contraire, si la condition n'est pas satisfaite, on saute à la ligne suivante $i+1$. Ce processus est répété jusqu'à ce qu'on arrive à la dernière ligne de la matrice.

Un fragment est considéré le premier dans sa ligne, s'il n'y a aucun fragment avant lui. Dans le cas où on trouve une diagonale qui satisfait la condition, on recommence la recherche depuis la 1re ligne. Si on arrive à la dernière ligne sans qu'on trouve une nouvelle diagonale qui satisfait la condition de tri, on passe à la colonne suivante dans la matrice.

Le calcul de la complexité : Pour trouver la bonne diagonale qui satisfait la condition (d'être la première), il se peut qu'on doive parcourir toute la colonne (tous les niveaux). On est sure que dans chaque étape (le parcourir d'une seule colonne, autrement un seul niveau), on trouve au moins une seule diagonale. Dans le pire cas, pour une seule diagonale, il faut parcourir toute la colonne donc les (k) lignes de la matrice (le nombre des séquences biologiques). Au total, on a n diagonales, donc pour traiter toutes les diagonales, il nous faut $O(k \times n)$.

Ensuite, dans la dernière étape 7.7, on regroupe les fragments des diagonales avec un temps de l'ordre de $O(n)$. La complexité globale de ces deux étapes devient $O(k \times n)$.

L'étape finale (regrouper les fragments des diagonales) peut être faite directement en même temps que l'opération de tri. À chaque fois qu'on trouve la diagonale qui satisfait la condition de tri, on la supprime de l'index et on regroupe ses fragments.

Exemple :
On considère l'ensemble suivant après la phase de suppression des diagonales inconsistantes : $D' = \{d_0, d_1, d_2, d_3, d_4, d_5, d_6\}$ (voir la figure 7.12).

On constate dans la figure 7.12 que la diagonale d_0 satisfait la condition expliquée précédemment, donc on l'ajoute dans l'ensemble trié des diagonales D'', et on supprime ses deux fragments.

Dans la figure 7.13a, la diagonale d_1, est la diagonale qui satisfait la condition expliquée précédemment.

Dans la figure 7.13b, on a deux diagonales d_2, et d_4 qui satisfont la condition expliquée précédemment en même temps. Donc on suit l'ordre d'apparition de leurs fragments dans la matrice, on commence par la diagonale d_2, on l'ajoute à D'' et on supprime ses fragments de la matrice. Ensuite on fait la même chose pour la diagonale d_4. L'étape montrée dans

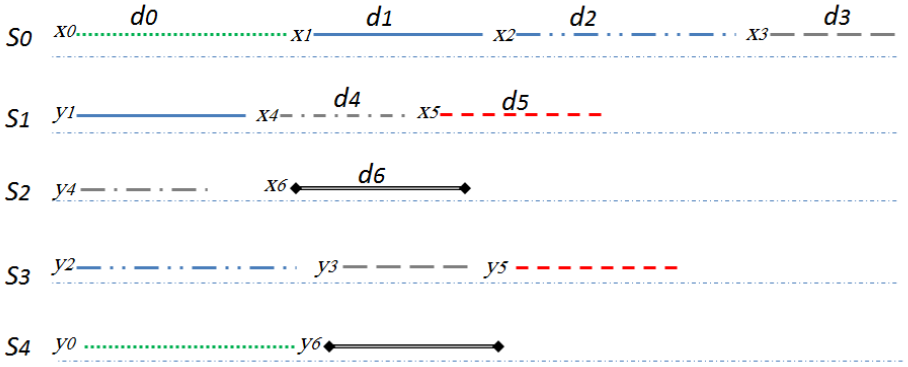

FIGURE 7.12 – L'ensemble des diagonales indexées dans une matrice.

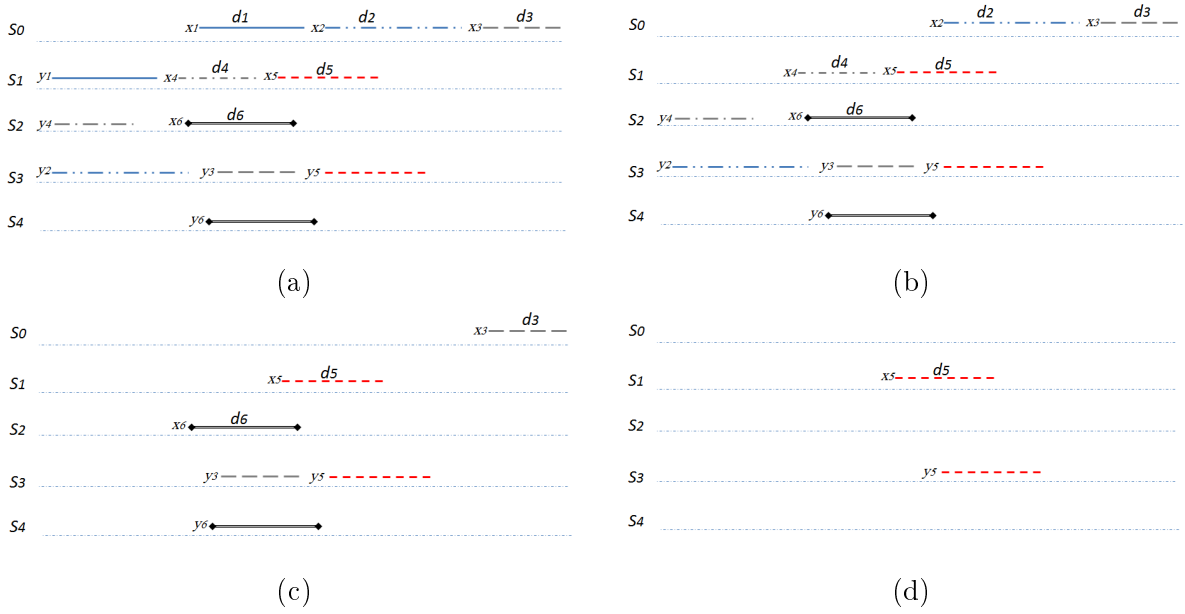

FIGURE 7.13 – Les itérations du tri des diagonales.

cette figure se fait en deux itérations, celle de d_2, ensuite celle de d_4.

Dans la figure 7.13c, on a aussi deux d_3, et d_6 qui satisfont la condition en même temps.

Dans la figure 7.13d, la diagonale d_5, est la diagonale qui satisfait la condition expliquée précédemment.

Pour notre exemple, l'ensemble des diagonales initiales qui est trié selon leurs poids était comme suit : $D' = \{d_0, d_1, d_2, d_3, d_4, d_5, d_6\}$.

Après l'opération de tri selon les positions des fragments des diagonales dans les séquences biologiques, l'ensemble devient : $D'' = \{d_0, d_1, d_2, d_4, d_3, d_6, d_5\}$.

7.7 Mettre les fragments des diagonales ensemble, et insérer les gaps (-)

Dans cette dernière étape, on prend l'ensemble des diagonales triées avec la deuxième méthode expliquée dans la section précédente, et pour chaque diagonale ($d_i = (x_i, y_i)$), on met ses deux fragments l'un en face de l'autre (l'un devant l'autre). Si le début du fragment x_i est, avant le début du fragment y_i, alors x_i est déplacé vers y_i, et vice versa. Pour déplacer le fragment, toute la sous-séquence qui commence à partir du fragment (x_i/y_i) doit être déplacée.

Après cette étape, on doit insérer les gaps ($-$) dans les espaces vides (qui sont le résultat du déplacement des fragments).

7.8 L'analyse et la comparaison

Nous avons réalisé deux implémentations de notre approche. La première version de notre approche est appelée *DiaWay 1.0* (Diagonal Way). Dans cette implémentation, pour chaque diagonale, on extrait toutes ses sous-diagonales comme nous l'avons expliqué dans la section 7.3, méthode 1. Notre deuxième implémentation est appelée *DiaWay 2.0*. Dans cette version, nous n'extrayons pas les sous-diagonales, on utilise une autre méthode pour améliorer la qualité d'alignement, voir la section 7.3 méthode 2 (dans la version actuelle de *DiaWay 2.0*, la méthode 2 expliquée dans la section 7.3 n'est encore implémentée, on utilise simplement les diagonales seules), et aussi, nous trions les diagonales avant leurs alignement final. La deuxième version de notre approche est une amélioration de la première version. Comme nous l'avons expliqué, nous avons changé les deux méthodes (l'extraction des diagonales, et leurs tri avant la phase finale) afin de rendre le programme beaucoup plus rapide.

Notre implémentation est modulaire, elle a été programmée en langage C++ avec l'utilisation du Framework Qt. Les tests ont été effectués sur Windows 7, 32 bits, avec un processeur Intel Core 2 duo E8400 3,00 GHz et 2 Gigabyte de RAM. Nous avons utilisé un seul cœur. Le code source et les deux versions de notre approche sont disponibles sur : https://github.com/chegrane/DiaWay_1.0 et https://github.com/chegrane/DiaWay_2.0. Notre projet est distribué sous la licence publique générale limitée GNU (GNU LGPL) (Anglais : GNU Lesser General Public License).

Nous avons fait une analyse sur les résultats des tests, et une comparaison avec l'algorithme original *DIALIGN*. Les ensembles de données (benchmarks) des séquences d'ADN sont récupérés depuis les bases de données **BAliBASE** et **SMART** [140].

Dans notre travail, nous utilisons la même formule de score définie dans [127, 135, 133], pour cela, l'analyse est basée sur le temps de calcul seulement.

Le temps d'exécution varie en fonction du nombre de séquences à aligner, de leurs longueurs et la longueur minimale de la diagonale autorisée. Soit un ensemble de séquences biologiques S avec une longueur $|S|$, la longueur des séquences moyennes est notée par $\bar{S}$. $|D|$ représente la longueur de l'ensemble D de toutes les diagonales (donc le nombre total des diagonales initiales). Et $|D^{'}|$ la taille de l'ensemble $D^{'}$ qui ne contient que les diagonales consistantes.

Le tableau 7.2 suivant, illustre la relation entre le nombre de diagonales consistantes et le nombre total de diagonales initiales.

$\|S\|$	$\bar{S}$	$\|D\|$	$\|D^{'}\|$
3	139	43757	156
5	202	101261	499
8	170	243845	661
10	101	131515	1474

TABLE 7.2 – Le nombre total de diagonales initiales et les diagonales consistantes restantes.

Dans le tableau 7.2, les résultats montrent qu'avant d'arriver à l'ensemble des diagonales consistantes, nous sommes obligés de supprimer presque toutes les diagonales initiales. Le nombre des diagonales consistantes restantes est inférieur à 1% de toutes les diagonales initiales.

Le temps d'exécution varie selon le nombre de toutes les diagonales initiales. Dans la figure 7.14, nous donnons le temps d'exécution en millisecondes pour une longueur minimale autorisée de diagonale = 7.

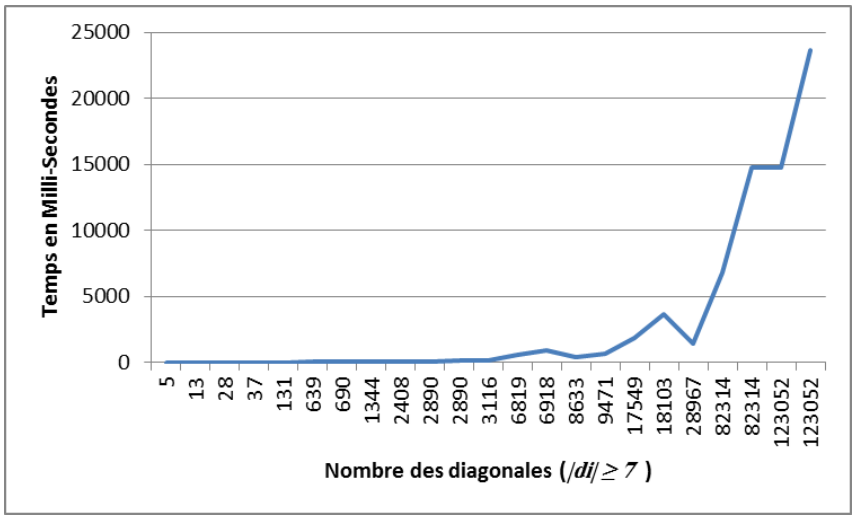

FIGURE 7.14 – Le temps d'exécution en millisecondes pour une longueur minimale de diagonale = 7.

Le graphe de la figure 7.14, montre que le temps d'exécution augmente en corrélation avec le nombre de diagonales. Cela signifie que si le nombre initial des diagonales est petit

	$\|S\|$	S	$\|D\|$	T en min
Haut similarité	9	40	15318	0,59
	9	76	57983	3,61
	10	73	63137	3,44
	10	75	67058	7,42
Similarité régulière	8	180	226217	0,64
	8	295	583156	1,36
	10	166	303685	1,52
	11	190	551049	2,27

TABLE 7.3 – Le temps (en minute) d'exécution en fonction du degré de similitude des séquences.

(juste après la première étape qui permet leur extraction voir 7.3), cela implique un temps d'exécution petit. Et dans le cas contraire, où on a un très grand nombre de diagonales, le temps de calcul est très élevé.

Si nous avons des séquences biologiques avec un degré élevé de similitude, le nombre de diagonales consistantes sera énorme. Cela veut dire que le graphe de vérification de l'inconsistance des diagonales (voir la section 7.5.1) devient aussi très grand. Cela rend le traitement de la méthode inconsistance de chemin plus lent, vu le nombre de chemins qui doivent être testés avant de trouver le bon, et qui implique un temps d'exécution plus élevé. Voir le tableau 7.3.

Dans le tableau 7.3, on a deux groupes de séquences. Le premier contient les séquences qui ont un degré élevé de similarité, et le second celles qui ont un degré de similarité régulier. On voit que la taille des séquences du deuxième ensemble (avec des similarités ordinaires) est plus grande que le premier ensemble avec une similarité élevée, et également, le nombre de diagonales du deuxième ensemble est excessivement plus grand que celui du premier ensemble. Malgré cela, le temps d'exécution de l'ensemble de similitude régulier est plus petit que l'ensemble de similarité élevée. Car, comme nous l'avons expliqué ci-dessus, dans le cas du degré de similitude élevé, le nombre de diagonales consistantes est très grand, et cela rend le traitement d'inconsistance plus lent (spécialement, la méthode de l'inconsistance de chemin, voir la section 7.5.1). Et cela implique un temps d'exécution plus élevé.

Nous avons comparé nos deux implémentations avec *DIALIGN 2.2* [141], son code source est disponible sur `http://bibiserv2.cebitec.uni-bielefeld.de/dialign/` [1]. Le temps d'exécution dépend du nombre de séquences biologiques, et leur longueur moyenne, voir le tableau 7.4.

Le symbole étoile (*) dans le tableau 7.4, signifie que l'application plante et ne fonc-

1. Visité le : 02-07-2016.

$\|S\|$	$\bar{S}$	DW.2.0	DW.1.0	DIALIGN 2.2
3	530	63	421	1312
4	250	31	125	766
5	200	47	125	*
6	119	32	1203	1750
7	195	125	422	1640
8	294	437	1453	4062
9	325	1047	6828	6297
10	476	3204	14782	16094
11	190	875	3656	3281

TABLE 7.4 – Le temps d'exécution de DIALIGN 2.2, DiaWay 1.0 (DW.1.0), et DiaWay 2.0 (DW.2.0) en millisecondes selon le nombre des séquences et leurs longueurs.

tionne pas pour cet exemple. D'après le tableau 7.4, on peut constater les points suivants :

- Le temps pour aligner 4 ou 5 séquences est petit par rapport au temps pour aligner 3 séquences, parce que leurs longueurs sont plus petites que celles des 3 séquences.
- La méthode *DiaWay 1.0* (DW.1.0) a un meilleur temps d'exécution par rapport à *DIALIGN 2.2* pour un petit ensemble de séquences avec une longueur moyenne relativement petite. Mais pour un grand nombre de séquences, son temps d'exécution est très mauvais parce que : 1) *DiaWay 1.0* (DW.1.0) extrait pour chaque diagonale toutes ses sous-diagonales et cela rend l'ensemble des diagonales énorme, 2) dans l'étape finale de l'alignement (mettre les fragments ensembles), on ne fait pas le tri des diagonales. Ces deux raisons rendent le temps de traitement très élevé.
- Le temps d'exécution pour notre version *DiaWay 2,0* (DW.2.0) est beaucoup plus petit que les deux autres méthodes, *DIALIGN* et *DiaWay 1.0* (DW.1.0). Voir la figure 7.15.

Dans le prochain test, nous allons prendre 10 séquences biologiques, et nous changeons à chaque fois la longueur des séquences afin de voir l'influence de ce paramètre sur le temps d'exécution, voir la figure 7.15.

Le temps d'exécution de notre version *DiaWay 2.0* (DW.2.0) est meilleur que celui des deux autres méthodes. Notre version *DiaWay 1.0* (DW.1.0) donne un temps d'exécution très mauvais pour une grande taille de séquences. Parce que, comme nous l'avons expliqué avant, elle extrait pour chaque diagonale toutes ses sous-diagonales, et aussi on ne fait pas le tri des diagonales dans l'étape finale de l'alignement, ce qui implique un temps d'exécution très important.

Pour montrer l'efficacité de notre méthode *DiaWay 2.0* (DW.2.0) par rapport à celle de *DIALIGN 2.2*, nous avons fait un test pour l'alignement par paire, nous prenons seulement deux séquences biologiques avec de très grande tailles. De plus, nous changeons à chaque fois la longueur. Voir le tableau 7.5, et la figure 7.16.

Dans le tableau 7.5, ont peut voir que le temps d'exécution de notre méthode DW.2.0

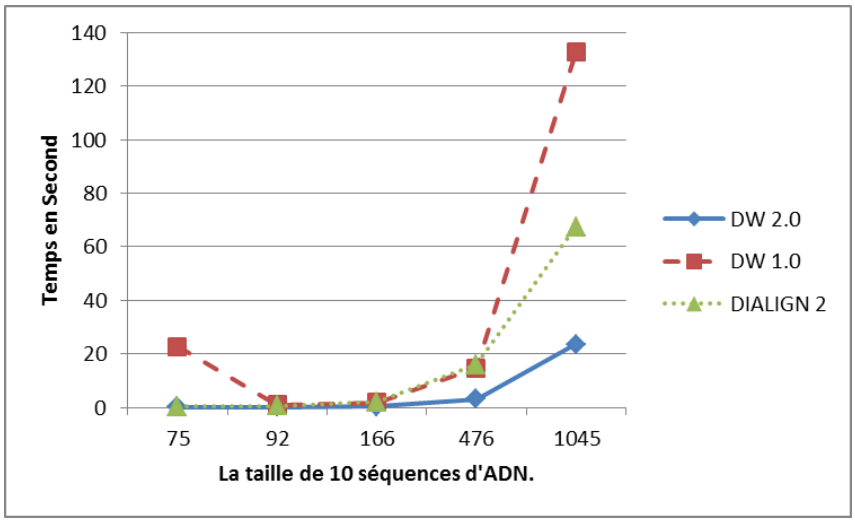

FIGURE 7.15 – Le temps d'exécution pour aligner 10 séquences de longueurs différentes.

$\|S\|$	$\tilde{S}$	DW.2.0	DW.1.0	DIALIGN 2.2
2	2000	203	57640	6796
2	3000	437	172359	14843
2	4032	812	392703	25718
2	5000	1235	710984	38828
2	7499	2828	2223297	81281

TABLE 7.5 – Le temps d'exécution en millisecondes selon la longueur de 2 séquences seulement.

est beaucoup plus petit que *DIALIGN 2.2*.

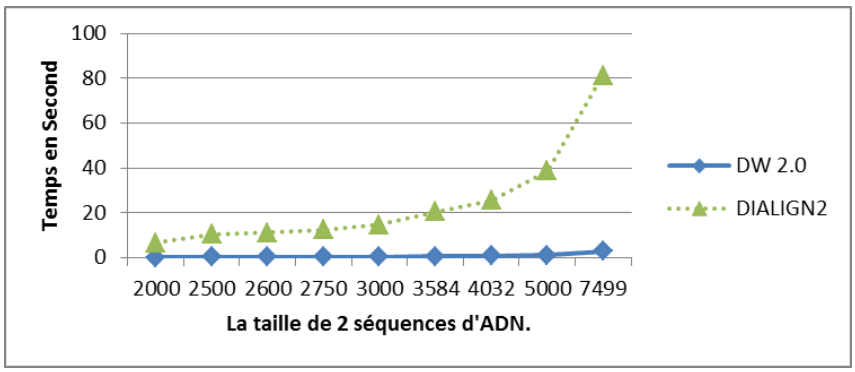

FIGURE 7.16 – Comparaison de l'alignement par paire, avec des séquences de très grandes tailles.

Dans ce graphe (la figure 7.16) nous avons omis la méthode *DiaWay 1.0* (DW.1.0) car elle donne un temps d'exécution très mauvais. Les résultats montrent que *DiaWay 2.0* (DW 2.0) est très efficace en comparant avec l'algorithme *DIALIGN 2.2*. Le temps d'exécution de notre méthode est presque stable.

7.9 Conclusion

Dans ce chapitre, nous avons présenté un nouvel algorithme d'alignement des séquences biologiques par paire et multiple. Un algorithme robuste et efficace en terme de temps d'exécution. Notre algorithme est basé sur l'algorithme *DIALIGN* qui est très connue pour l'alignement des séquences d'ADN et de protéine.

Nous avons présenté les différentes étapes de notre approche, de l'étape d'extraction des diagonales à l'étape finale qui consiste à regrouper les fragments des diagonales et l'insertion des gaps, en passant par l'étape de l'indexation des diagonales et l'étape de la vérification de l'inconsistance des diagonales, où nous avons proposé une nouvelle méthode qui utilise les graphes afin de résoudre ce problème de l'inconsistance.

Dans l'étape de tri des diagonales pour l'alignement final, nous avons proposé une nouvelle technique de tri, car le besoin est différent et on ne peut pas appliquer les algorithmes de tri classique.

Nous avons réalisé deux implémentations de notre approche, *DiaWay 1.0* et *DiaWay 2.0*. Nous avons fait différents tests sur des séquences par paires et multiple. Les résultats montrent que notre approche *DiaWay 2.0* est très efficace dans les deux cas d'alignement d'ADN par paire et multiple, et donne un très bon temps d'exécution comparé à *DIALIGN 2.2*.

Chapitre 8

Conclusion générale

L'objet de cette thèse porte sur l'étude du problème de la recherche approchée en général, et son étude dans différents contextes importants de l'informatique.

a La recherche approchée dans un dictionnaire ou un texte.

b L'auto-complétion approchée.

c L'alignement de séquences biologiques.

Chaque contexte à de nombreuses applications. Nous avons apporté des solutions qui donnent aussi bien en théorie qu'en pratique des résultats satisfaisants en complexité.

La recherche approchée dans un dictionnaire/texte. Nous avons apporté deux solutions.

La première solution utilise les tables de hachage, nous avons présenté une solution pour résoudre le problème de la recherche approchée pour $k \geq 2$. L'idée de notre algorithme est basée sur l'utilisation des tableaux de hachage avec sondage linéaire et des signatures de hachage afin d'accélérer le temps de recherche. Nous avons proposé deux variantes de notre solution, une version non-compacte qui est incrémentale (possibilité d'insertion des nouveaux éléments), et une version compacte qui optimise l'espace mémoire.

La deuxième solution résout le problème de la recherche approchée dans un dictionnaire/texte pour $k = 1$ erreur. La solution utilise le Trie et le Trie inversé afin d'obtenir les caractères possibles pour une position d'erreur donnée et de vérifier les parties exactes du mot requête rapidement. Nous avons présenté trois variantes de notre solution, la première TRT_CI effectue une intersection sur des caractères communs des transitions sortantes entre les deux nœuds d'erreur du Trie et le Trie inversé pour déterminer les chemins qui peuvent conduire à des solutions. Dans la seconde méthode TRT_WNI, nous faisons une intersection entre deux ensembles de numéros de mots pour déterminer les solutions. Dans la troisième variante TRT_CWNI, nous combinons les deux méthodes TRT_CI et

TRT_WNI, pour réduire le nombre de feuilles avant de faire une intersection entre leurs numéros.

Dans la première partie, dans les deux chapitres 4 et 5 où on a proposé deux solutions pour le problème de la recherche approchée (une solution qui se base sur le hachage et l'autre qui se base sur le Trie et Trie inversé), les mots du dictionnaire/texte recherchés sont de longueur moyenne, en particulier lorsque en considère les langues naturelles, et le nombre d'erreurs k est donnée d'hypothèse. La complexité dépend de la taille des mots, la taille de dictionnaire, la taille de l'alphabet et k.

L'auto-complétion approchée Dans ce travail, nous avons présenté une solution pratique pour résoudre le problème de l'auto-complétion approchée dans une architecture client-serveur, et nous avons présenté la méthode Top-k pour améliorer la qualité des résultats en utilisant un système de classement dynamique et statique.

Dans le problème de l'auto-complétion, la requête recherchée par un algorithme de recherche approchée est un préfixe d'un mot, donc sa taille est relativement plus petite que celle des mots du cas précédent, ou de même ordre de grandeur. Après l'étape de la recherche approchée de préfixe taper, on a besoin d'une 2^{e} étape pour obtenir tous les suffixes de ce préfixe. Pour cela, on est obligé de proposer des solutions spécifiques à ce problème (l'auto-complétion approchée) indépendamment de problème général de la recherche approchée. C'est la raison pour laquelle nous avons étudié ce problème et proposé quatre variantes pour la recherche approchée du préfixe. La première recherche dans un Trie le préfixe avec retour arrière (la méthode naïve). La seconde recherche le préfixe par l'algorithme 1 (avec hachage) du problème général. Et la troisième solution combine une structure de Trie avec hachage pour utiliser une liste de substitutions qui permet de choisir les chemins des solutions. La quatrième variante, prend en compte la profondeur du Trie dans lequel le préfixe est recherché. Dans les niveaux proches de la racine, la recherche se fait par la variante 3, et dans les niveaux proches des feuilles, elle se fait par la variante 1. Les expérimentations ont montré que cette dernière variante donne les meilleures performances de recherche.

L'auto-complétion nécessite une étape qui trouve tous les suffixes d'un préfixe. Pour cela nous avons proposé trois variantes d'algorithmes pour s'adapter au comportement de l'utilisateur. 1) Commencer la recherche à partir de la racine. 2) Commencer la recherche depuis le dernier nœud de la requête précédente lorsque le mot requête précédent est un préfixe complet du mot actuel. 3) Commencer la recherche depuis le plus grand préfixe commun.

Pour réduire le nombre des résultats obtenus, nous appliquons une fonction qui retourne les résultats les plus pertinents (Top-k) vis-vis d'un score (statique ou dynamique)

associé.

Nous avons réalisé une bibliothèque (nommée Appaco_lib), pour être utilisée soit du côté serveur ou du côté client ou les deux en même temps pour répondre rapidement à des requêtes d'auto-complétion approchée pour des dictionnaires UTF-8 (donc toutes les langues).

Alignement multiple des séquences d'ADN : Nous avons aussi examiné la recherche approchée dans le contexte de la bio-informatique.

Aligner deux séquences x et y est de mettre les régions similaires de x en face de y. Trouver le bon alignement revient à minimiser le nombre d'opérations qui permet de transformer la séquence x en la séquence y, ou soit à maximiser la similarité entre x et y.

Ce problème ainsi exprimé, est donc un problème de recherche approchée où les mots (les séquences biologiques) sont de taille relativement grande, le seuil d'erreur k est inconnu, et la taille de l'alphabet est réduite.

Dans ce travail, nous avons présenté un nouvel algorithme d'alignement des séquences biologiques par paire et multiple robuste et efficace en terme de temps d'exécution, notre algorithme est basé sur l'algorithme *DIALIGN* qui est très connu pour l'alignement des séquences d'ADN et de protéine.

Nous avons réalisé deux implémentations de notre approche, *DiaWay 1.0* et *DiaWay 2.0*. Nous avons fait différents tests sur des séquences multiples et par paires. Les résultats montrent que notre approche (*DiaWay 2.0*) est très efficace dans les deux cas, l'alignement d'ADN multiple et par paire, et donne un très bon temps d'exécution comparé à *DIALIGN 2.2*.

Toutes nos implémentations sont distribuées sous la licence publique générale limitée GNU (GNU LGPL) (Anglais : GNU Lesser General Public License). Les résultats de notre travail sont disponibles sous forme de bibliothèques sur :

- La recherche approchée avec les tableaux de hachage : https://code.google.com/p/compact-approximate-string-dictionary/
- La recherche approchée avec le Trie et le Trie inversé : https://github.com/chegrane/TrieRTrie
- L'auto-complétion approchée avec top-k : https://github.com/AppacoLib/api.appacoLib
- L'alignement multiple des séquences biologiques : https://github.com/chegrane/diaway pour la première version, et sur https://github.com/chegrane/DiaWay_2.0 pour la deuxième version.

Au terme de ce travail et avant d'envisager quelques perspectives rappelons quelques problèmes qui se sont posés dans toute cette problématique de la recherche approchée

dans un mot autrement dit la présence de(s) erreur(s) dans les mots. Nous en mentionnons trois :

1. La position de l'erreur dans le mot requête : tout simplement lorsqu'on a une requête q d'une longueur m, la difficulté provient du fait que l'on ne sait pas à quelle position l'erreur se trouve, elle peut être à n'importe quelle position de l'intervalle $[1..m]$.

   Lorsque le nombre d'erreurs est ≥ 2, le problème devient plus complexe étant donné la combinaison des positions des erreurs que l'on doit considérer : une combinaison de $\binom{m}{k}$ positions.

   Inversement, si on connait par avance la position de l'erreur (ou les positions possibles), la solution devient facile et rapide à obtenir. Dans le cas où $k \geq 2$, si seulement on connaît juste une partie des positions de l'erreur par exemple celle d'une partie de gauche ou de droite du mot requête, la complexité des combinaisons des positions peut être réduite.

   On peut alors envisager quelques questions à explorer qui pourraient améliorer ce travail, par exemple :

2. Dans une position donnée, quels sont les caractères qui représentent une solution où au moins qui mènent à des solutions possibles. Ce problème peut aussi dépendre de la taille de l'alphabet, petite ou grande.

3. Quels types d'erreurs considérer ? (en fonction des différentes mesures de distance entre deux mots). Dans le cas de $k \geq 2$ erreurs, on a une combinaison à faire entre les types d'erreurs et les positions où elles pourraient se trouver. Quel est l'impact du choix de la fonction de distance dans ce problème ?

   Si on sait d'avance quels sont les types d'erreurs concernées dans le travail (par exemple dans une recherche approchée où les erreurs sont celles d'un système d'OCR, il est possible de caractériser, expérimentalement, les erreurs possible d'un OCR), il serait alors possible de diminuer le nombre de combinaisons et donc un temps de calcul plus réduit. Cela permet aussi de proposer des algorithmes moins complexes adaptés au besoin.

Il y a d'autres paramètres que l'on pourrait aussi prendre en considération afin de résoudre ce problème comme l'espace mémoire occupé par la structure de donnée et le temps du pré-traitement. Une réflexion dans cette direction peut apporter de nouvelles solutions meilleures.

Références

[1] Cisłak, A., Grabowski, S. : A practical index for approximate dictionary matching with few mismatches. arXiv preprint arXiv :1501.04948 (2015) xi, 79, 80, 81, 82

[2] Levenshtein, V.I. : Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. Soviet Physics Doklady **10** (1966) 707–710 2, 9, 90

[3] Navarro, G. : A guided tour to approximate string matching. ACM Comput. Surv. **33**(1) (2001) 31–88 2, 3, 17, 20, 21, 68

[4] Boytsov, L. : Indexing methods for approximate dictionary searching : Comparative analysis. ACM Journal of Experimental Algorithmics **16**(1) (2011) 2, 17

[5] Cucerzan, S., Brill, E. : Spelling correction as an iterative process that exploits the collective knowledge of web users. In : EMNLP. Volume 4. (2004) 293–300 3

[6] Pollock, J.J., Zamora, A. : Automatic spelling correction in scientific and scholarly text. Communications of the ACM **27**(4) (1984) 358–368 3

[7] Reynaert, M. : Non-interactive OCR post-correction for giga-scale digitization projects. In : Computational Linguistics and Intelligent Text Processing. Springer (2008) 617–630 3

[8] Manber, U., Wu, S. : An algorithm for approximate membership checking with application to password security. Information Processing Letters **50**(4) (1994) 191–197 3

[9] Chaudhuri, S., Ganjam, K., Ganti, V., Motwani, R. : Robust and efficient fuzzy match for online data cleaning. In : Proceedings of the 2003 ACM SIGMOD international conference on Management of data, ACM (2003) 313–324 3

[10] Waterman, M.S. : Introduction to computational biology: maps, sequences and genomes. CRC Press (1995) 3, 129

[11] Gusfield, D.: Algorithms on strings, trees, and sequences: computer science and computational biology. Volume 24. Cambridge University Press (1997) 3, 8, 10, 11, 17, 20, 129

[12] Lesk, A.: Introduction to bioinformatics. Oxford University Press (2013) 3, 129

[13] Navarro, G., Raffinot, M.: Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences. Cambridge University Press (2002) 3, 17

[14] Vardi, M.Y.: Boolean satisfiability: theory and engineering. Commun. ACM **57**(3) (2014) 5 4

[15] Belazzougui, D.: Faster and Space-Optimal Edit Distance "1" Dictionary. In: Combinatorial Pattern Matching, 20th Annual Symposium, CPM 2009, Lille, France, June 22-24, 2009, Proceedings. (2009) 154–167 5, 22, 23, 25, 35, 40, 48, 105

[16] Chegrane, I., Belazzougui, D.: Simple, compact and robust approximate string dictionary. Journal of Discrete Algorithms **28** (2014) 49 – 60 StringMasters 2012 & 2013 Special Issue (Volume 1). 5, 42, 43, 79, 80, 105

[17] Chegrane, I., Belazzougui, D., Raffinot, M.: JQuery UI like approximate autocomplete. In: International Symposium on Web Algorithms. (2015) 6

[18] Chegrane, I., Seghier, A., Ighilaza, C., Boutorh, A.: Diagonal consistency problem resolution in dialign algorithm. In: the International Conference on Bioinformatics Models, Methods and Algorithms. (2015) 225–231 7, 130

[19] Seghier, A., Chegrane, I., Ighilaza, C.: DNA multiple alignment problem with the new DiaWay algorithm. In: 12th International Symposium on Programming and Systems (ISPS) 2015, IEEE (2015) 1–7 7, 130, 144

[20] Carton, O.: Langages formels, calculabilité et complexité. Volume 101. Vuibert (2008) 8

[21] Cormen, T.H.: Introduction to algorithms. MIT press (2009) 8, 10, 12, 13

[22] Knuth, D.E.: The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1998) 8, 13

[23] Hamming, R.W.: Error detecting and error correcting codes. Bell System technical journal **29**(2) (1950) 147–160 10

[24] Damerau, F.J.: A technique for computer detection and correction of spelling errors. Communications of the ACM **7**(3) (1964) 171–176 10

[25] Crochemore, M., Hancart, C., Lecroq, T.: Algorithmique du texte. Volume 347. Vuibert Paris (2001) 10, 17

[26] Crochemore, M., Rytter, W.: Jewels of stringology: text algorithms. World Scientific (2002) 10, 17

[27] Crochemore, M., Lecroq, T.: Suffix tree. In: Encyclopedia of Database Systems. Springer (2009) 2876–2880 10

[28] Weiner, P.: Linear pattern matching algorithms. In: Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on. (1973) 1–11 11, 20

[29] McCreight, E.M.: A Space-Economical Suffix Tree Construction Algorithm (1976) 11, 20

[30] Ukkonen, E.: On-line construction of suffix trees (1995) 11, 20

[31] Crochemore, M., Lecroq, T.: Trie. In: Encyclopedia of Database Systems. Springer (2009) 3179–3182 11

[32] De La Briandais, R.: File searching using variable length keys. In: Papers presented at the the March 3-5, 1959, western joint computer conference, ACM (1959) 295–298 11

[33] Fredkin, E.: Trie memory. Communications of the ACM **3**(9) (1960) 490–499 11

[34] Manber, U., Myers, G.: Suffix Arrays: A New Method for On-Line String Searches. SIAM Journal on Computing **22**(5) (October 1993) 935–948 11, 20

[35] Kasai, T., Lee, G., Arimura, H., Arikawa, S., Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching. CPM '01, Springer-Verlag (2001) 181–192 11

[36] Williams, J.W.J.: Algorithm-232-heapsort (1964) 12

[37] Emde Boas, P., Kaas, R., Zijlstra, E.: Design and implementation of an efficient priority queue. Mathematical Systems Theory **10**(1) (December 1976) 99–127 12

[38] Jones, D.W.: An empirical comparison of priority-queue and event-set implementations. Communications of the ACM **29**(4) (March 1986) 300–311 12

[39] Elias, P.: Efficient storage and retrieval by content and address of static files. Journal of the ACM (JACM) **21**(2) (1974) 246–260 12

[40] Fano, R.M.: On the number of bits required to implement an associative memory. Massachusetts Institute of Technology, Project MAC (1971) 12

[41] Fenwick, P.M.: A new data structure for cumulative frequency tables. Software: Practice and Experience **24**(3) (1994) 327–336 12

[42] Jacobson, G.: Space-efficient static trees and graphs. In: Foundations of Computer Science, 1989., 30th Annual Symposium on, IEEE (1989) 549–554 13, 50

[43] Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In: Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '02, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2002) 233–242 13

[44] Munro, J.I.: Tables. In: Foundations of Software Technology and Theoretical Computer Science, Springer (1996) 37–42 13, 50

[45] Cichelli, R.J.: Minimal perfect hash functions made simple. Communications of the ACM **23**(1) (1980) 17–19 14

[46] Jaeschke, G.: Reciprocal hashing: A method for generating minimal perfect hashing functions. Communications of the ACM **24**(12) (1981) 829–833 14

[47] Czech, Z.J., Havas, G., Majewski, B.S.: An optimal algorithm for generating minimal perfect hash functions. Information Processing Letters **43**(5) (1992) 257–264 14

[48] Knuth, D.E.: Notes on "Open" addressing (1963) Unpublished Memorandum. 14, 41, 58

[49] Knuth, D.: The Art of Computer Programming: Fundamental algorithms. Number vol. 1 in Addison-Wesley series in computer science and information processing. Addison-Wesley (1997) 14

[50] Knuth, D.E.: Big omicron and big omega and big theta. ACM Sigact News **8**(2) (1976) 18–24 15

[51] Aggarwal, A., Vitter, J., et al.: The input/output complexity of sorting and related problems. Communications of the ACM **31**(9) (1988) 1116–1127 15

[52] Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Foundations of Computer Science, 1999. 40th Annual Symposium on, IEEE (1999) 285–297 15

[53] Karp, R.M.: A survey of parallel algorithms for shared-memory machines. (1988) 15, 23

[54] Turing, A.M., Copeland, B.J.: The essential turing: seminal writings in computing, logic, philosophy, artificial intelligence, and artificial life plus the secrets of enigma. (2004) 15

[55] Luginbuhl, D.R.: Computational complexity of random access models. Technical report, DTIC Document (1990) 16

[56] Cook, S.A., Reckhow, R.A.: Time-bounded random access machines. In: Proceedings of the fourth annual ACM symposium on Theory of computing, ACM (1972) 73–80 16

[57] Levin, L.A.: Average case complete problems. SIAM Journal on Computing **15**(1) (1986) 285–286 16

[58] Bogdanov, A., Trevisan, L.: Average-case complexity. arXiv preprint cs/0606037 (2006) 16

[59] Navarro, G., Baeza-Yates, R.A., Sutinen, E., Tarhio, J.: Indexing methods for approximate string matching. IEEE Data Eng. Bull. **24**(4) (2001) 19–27 17, 21, 68

[60] Charras, C., Lecroq, T.: Handbook of exact string matching algorithms. King's College Publications London, UK: (2004) 17

[61] Bellman, R.: On the theory of dynamic programming. Proceedings of the National Academy of Sciences of the United States of America **38**(8) (1952) 716 19

[62] Bellman, R.: Dynamic programming and Lagrange multipliers. Proceedings of the National Academy of Sciences of the United States of America **42**(10) (1956) 767 19

[63] Sellers, P.H.: The theory and computation of evolutionary distances: pattern recognition. Journal of algorithms **1**(4) (1980) 359–373 20

[64] Masek, W.J., Paterson, M.S.: A faster algorithm computing string edit distances. Journal of Computer and System sciences **20**(1) (1980) 18–31 20

[65] Ukkonen, E.: Approximate string-matching over suffix trees. Combinatorial Pattern Matching (1993) 228–242 20, 21, 23

[66] Sutinen, E., Tarhio, J.: On using q-gram locations in approximate string matching. In: Algorithms–ESA'95. Springer (1995) 327–340 20, 23

[67] Needleman, S.B., Wunsch, C.D.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. Journal of Molecular Biology **48**(3) (1970) 443 – 453 20, 129

[68] Smith, T., Waterman, M.: Identification of common molecular subsequences. Journal of Molecular Biology **147**(1) (1981) 195 – 197 20, 129

[69] Morrison, D.R.: PATRICIA—Practical Algorithm To Retrieve Information Coded in Alphanumeric. Journal of the ACM **15**(4) (October 1968) 514–534 20

[70] Grossi, R., Vitter, J.S.: Compressed suffix arrays and suffix trees with applications to text indexing and string matching. SIAM Journal on Computing **35**(2) (2005) 378–407 20

[71] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnell, R.M.: Linear size finite automata for the set of all subwords of a word - an outline of results. Bulletin of the EATCS **21** (1983) 12–20 20

[72] Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M.T., Seiferas, J.: The smallest automation recognizing the subwords of a text. Theoretical Computer Science **40** (1985) 31–55 20

[73] Blumer, A., Blumer, J., Ehrenfeucht, A., Haussler, D., McConnell, R.: Building a complete inverted file for a set of text files in linear time. In: Proceedings of the sixteenth annual ACM symposium on Theory of computing, ACM (1984) 349–358 20

[74] Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on, IEEE (2000) 390–398 20

[75] Ferragina, P., Manzini, G.: Indexing compressed text. Journal of the ACM (JACM) **52**(4) (2005) 552–581 20

[76] Shang, H., Merrettal, T.: Tries for approximate string matching. Knowledge and Data Engineering, IEEE Transactions on **8**(4) (1996) 540–547 20

[77] Russo, L., Navarro, G., Oliveira, A.L., Morales, P.: Approximate string matching with compressed indexes. Algorithms **2**(3) (2009) 1105–1136 20

[78] Bieganski, P., Riedl, J., Retzel, E.F., et al.: Generalized suffix trees for biological sequence data: Applications and implementation. In: System Sciences, 1994. Proceedings of the Twenty-Seventh Hawaii International Conference on. Volume 5., IEEE (1994) 35–44 20

[79] Zobel, J., Dart, P.: Finding approximate matches in large lexicons. Softw., Pract. Exper. **25**(3) (1995) 331–345 20

[80] Rivest, R.L.: Partial-match retrieval algorithms. SIAM Journal on Computing **5**(1) (1976) 19–50 21

[81] Wu, S., Manber, U.: Fast text searching: allowing errors. Communications of the ACM **35**(10) (1992) 83–91 21, 23

[82] Brodal, G.S., Gasieniec, L.: Approximate dictionary queries. In: Combinatorial Pattern Matching, Springer (1996) 65–74 21, 23

[83] Amir, A., Keselman, D., Landau, G.M., Lewenstein, M., Lewenstein, N., Rodeh, M.: Text indexing and dictionary matching with one error. Journal of Algorithms **37**(2) (November 2000) 309–325 21, 25, 38, 68, 75, 76, 87

[84] Mor, M., Fraenkel, A.S.: A hash code method for detecting and correcting spelling errors. Communications of the ACM **25**(12) (1982) 935–938 22

[85] Russo, L.M., Oliveira, A.L.: An efficient algorithm for generating super condensed neighborhoods. In: Combinatorial Pattern Matching, Springer (2005) 104–115 22

[86] Myers, E.W.: A sublinear algorithm for approximate keyword searching. Algorithmica **12**(4-5) (1994) 345–374 22

[87] Bocek, T., Hunt, E., Hausheer, D., Stiller, B.: Fast similarity search in peer-to-peer networks. NOMS 2008 - 2008 IEEE Network Operations and Management Symposium (2008) 22, 23

[88] Karch, D., Luxen, D., Sanders, P.: Improved fast similarity search in dictionaries. In: String Processing and Information Retrieval, Springer (2010) 173–178 22, 23, 62, 79, 80

[89] Boytsov, L.: Super-linear indices for approximate dictionary searching. In: SISAP. (2012) 162–176 22, 51, 62, 63

[90] Karp, R.M., Rabin, M.O.: Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development **31**(2) (1987) 249–260 22, 41

[91] Ukkonen, E.: Approximate string-matching with q-grams and maximal matches. Theoretical computer science **92**(1) (1992) 191–211 22

[92] Wu, S., Manber, U.: Agrep–a fast approximate pattern-matching tool. Usenix Winter 1992 (1992) 153–162 22

[93] Baeza-Yates, R., Gonnet, G.H.: A new approach to text searching. Communications of the ACM **35**(10) (1992) 74–82 23

[94] Wu, S., Manber, U., Myers, G.: A subquadratic algorithm for approximate limited expression matching. Algorithmica **15**(1) (1996) 50–67 23

[95] Myers, G.: A fast bit-vector algorithm for approximate string matching based on dynamic programming. Journal of the ACM (JACM) **46**(3) (1999) 395–415 23

[96] Navarro, G., Raffinot, M.: Fast and flexible string matching by combining bit-parallelism and suffix automata. Journal of Experimental Algorithmics (JEA) **5** (2000) 4 23

[97] Navarro, G., Raffinot, M.: A bit-parallel approach to suffix automata: Fast extended string matching. In: Combinatorial Pattern Matching, Springer (1998) 14–33 23

[98] Hyyrö, H.: Explaining and extending the bit-parallel approximate string matching algorithm of myers. Technical report, Citeseer (2001) 23

[99] Hyyrö, H., Navarro, G.: Faster bit-parallel approximate string matching. In: Combinatorial Pattern Matching, Springer (2002) 203–224 23

[100] Hyyrö, H.: Practical Methods for Approximate String Matching. Tampereen yliopisto (2003) 23

[101] Hyyrö, H., Fredriksson, K., Navarro, G.: Increased bit-parallelism for approximate and multiple string matching. Journal of Experimental Algorithmics (JEA) **10** (2005) 2–6 23

[102] Cole, R., Gottlieb, L.A., Lewenstein, M.: Dictionary matching and indexing with errors and don't cares. In: STOC. (2004) 91–100 23, 37, 38

[103] Cole, R., Hariharan, R.: Approximate string matching: A simpler faster algorithm. SIAM Journal on Computing **31**(6) (2002) 1761–1782 23

[104] Pacheco, P.: An introduction to parallel programming. Elsevier (2011) 23

[105] Grama, A.: Introduction to parallel computing. Pearson Education (2003) 23

[106] Landau, G.M., Vishkin, U.: Fast parallel and serial approximate string matching. Journal of algorithms **10**(2) (1989) 157–169 23

[107] Landau, G.M., Vishkin, U.: Introducing efficient parallelism into approximate string matching and a new serial algorithm. In: Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing. STOC '86, New York, NY, USA, ACM (1986) 220–230 23

[108] Kouzinopoulos, C.S., Margaritis, K.G.: String matching on a multicore GPU using CUDA. In: Informatics, 2009. PCI'09. 13th Panhellenic Conference on, IEEE (2009) 14–18 23

[109] Lin, C.H., Tsai, S.Y., Liu, C.H., Chang, S.C., Shyu, J.M.: Accelerating string matching using multi-threaded algorithm on GPU. In: Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE, IEEE (2010) 1–5 23

[110] Zha, X., Sahni, S.: Multipattern string matching on a GPU. In: 2011 IEEE Symposium on Computers and Communications (ISCC), IEEE (2011) 277–282 23

[111] Buchsbaum, A.L., Goodrich, M.T., Westbrook, J.R.: Range searching over tree cross products. In: Algorithms-ESA 2000. Springer (2000) 120–131 25

[112] Agarwal, P.K., Erickson, J., et al.: Geometric range searching and its relatives. Contemporary Mathematics **223** (1999) 1–56 25

[113] Berge, C., Minieka, E.: Graphs and hypergraphs. Volume 7. North-Holland publishing company Amsterdam (1973) 25

[114] Benoit, D., Demaine, E.D., Munro, J.I., Raman, R., Raman, V., Rao, S.S.: Representing trees of higher degree. Algorithmica **43**(4) (2005) 275–292 27

[115] Overmars, M.H.: Efficient data structures for range searching on a grid. Journal of Algorithms **9**(2) (1988) 254–275 39

[116] Heileman, G.L., Luo, W.: How caching affects hashing. In: ALENEX/ANALCO. (2005) 141–154 41

[117] Anderson, S.E.: Bit twiddling hacks. https://graphics.stanford.edu/~seander/bithacks.html Visité le: 02-07-2016. 50

[118] Mitzenmacher, M., Vadhan, S.: Why Simple Hash Functions Work: Exploiting the Entropy in a Data Stream. In: Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms. SODA '08, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics (2008) 746–755 59

[119] World Wide Web Consortium (W3C): HTML5: Edition for Web Authors. http://www.w3.org/TR/2013/NOTE-html5-author-20130528/the-input-element.html (2013) Visité le: 02-07-2016. 88

[120] World Wide Web Consortium (W3C): HTML5: A vocabulary and associated APIs for HTML and XHTML. http://www.w3.org/TR/2012/WD-html5-20121025/common-input-element-attributes.html (2012) Visité le: 02-07-2016. 88

[121] Wikipedia: Intelligent code completion. http://en.wikipedia.org/wiki/Intelligent_code_completion (2014) Visité le: 02-07-2016. 89

[122] Elias, P.: Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory **21**(2) (March 1975) 194–203 97

[123] Starner, T.: Human-powered wearable computing. IBM Systems Journal **35**(3.4) (1996) 618–629 115

[124] Miller, R.B.: Response time in man-computer conversational transactions. In: Proceedings of the December 9-11, 1968, fall joint computer conference, part I on -

AFIPS '68 (Fall, part I), New York, New York, USA, ACM Press (December 1968) 267 115

[125] Mount, D.W.: Bioinformatics: sequence and genome analysis. Volume 2. Cold spring harbor laboratory press New York: (2001) 129

[126] Thompson, J.D., Higgins, D.G., Gibson, T.J.: CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. Nucleic Acids Research **22**(22) (1994) 4673–4680 130

[127] Morgenstern, B., Dress, A., Werner, T.: Multiple DNA and protein sequence alignment based on segment-to-segment comparison. Proceedings of the National Academy of Sciences **93**(October) (1996) 12098–12103 130, 131, 132, 137, 147

[128] Notredame, C., Higgins, D.G., Heringa, J.: T-coffee: a novel method for fast and accurate multiple sequence alignment. Journal of Molecular Biology **302**(1) (2000) 205 – 217 130

[129] Edgar, R.C.: MUSCLE: multiple sequence alignment with high accuracy and high throughput. Nucleic Acids Research **32**(5) (2004) 1792–1797 130

[130] Derrien, V., Richer, J.M., Hao, J.K.: Plasma, un nouvel algorithme progressif pour l'alignement multiple de séquences. In Solnon, C., ed.: Premières Journées Francophones de Programmation par Contraintes, Lens, CRIL - CNRS FRE 2499, Université d'Artois (June 2005) 39–48 http://www710.univ-lyon1.fr/čsolnon. 130

[131] Pearson, W.R., Lipman, D.J.: Improved tools for biological sequence comparison. Proceedings of the National Academy of Sciences **85**(8) (1988) 2444–2448 130

[132] Altschul, S.F., Gish, W., Miller, W., Myers, E.W., Lipman, D.J.: Basic local alignment search tool. Journal of molecular biology **215**(3) (1990) 403–410 130

[133] Morgenstern, B., Atchley, W., Hahn, K., Dress, A.: Segment-based scores for pairwise and multiple sequence alignments. Ismb (1998) 130, 131, 132, 147

[134] Seghier, A., Chegrane, I.: Amélioration de l'algorithme d'alignement multiple des séquences d'ADN DiAlign et mise en œuvre d'une solution repartie. Master's thesis, USTHB, Bab Ezzouar, Alger (6 2010) 17/20. 130

[135] Morgenstern, B., Frech, K., Dress, A., Werner, T.: Dialign: finding local similarities by multiple sequence alignment. Bioinformatics **14**(3) (1998) 290–294 132, 147

[136] Subramanian, A.R., Kaufmann, M., Morgenstern, B.: DIALIGN-TX: greedy and progressive approaches for segment-based multiple sequence alignment. Algorithms for molecular biology : AMB **3** (January 2008) 6 132

[137] Subramanian, A.R., Weyer-Menkhoff, J., Kaufmann, M., Morgenstern, B.: DIALIGN-T: an improved algorithm for segment-based multiple sequence alignment. BMC bioinformatics **6** (January 2005) 66 135

[138] Bondy, J., Murty, U.: Graph Theory with Applications. Macmillan (1976) 138

[139] Cormen, T., Leiserson, C., Rivest, R., Stein, C.: Depth-first search. In: Introduction To Algorithms. MIT Press (2001) 540–549 139

[140] Carroll, H., Beckstead, W., O'Connor, T., Ebbert, M., Clement, M., Snell, Q., McClellan, D.: DNA reference alignment benchmarks based on tertiary structure of encoded proteins. Bioinformatics **23**(19) (2007) 2648–2649 147

[141] Morgenstern, B.: DIALIGN 2: improvement of the segment-to-segment approach to multiple sequence alignment. Bioinformatics **15**(3) (1999) 211–218 149