

# THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE RENNES

ÉCOLE DOCTORALE N° 601

*Mathématiques, Télécommunications, Informatique,  
Signal, Systèmes, Électronique.*

Spécialité : *Informatique*

Par

**Garance GOURDEL**

## Sketch-Based Approaches to Processing Massive String Data

Thèse présentée et soutenue à Paris, le 26 Octobre 2023

Unité de recherche : Équipe GenScale, Univ Rennes, Inria, CNRS, IRISA

### Rapporteurs avant soutenance :

Thierry LECROQ    Professeur des universités, Université de Rouen Normandie

Simon J. PUGLISI    Professeur, University of Helsinki

### Composition du Jury :

Rapporteurs :    Thierry LECROQ    Professeur, Université de Rouen Normandie

Simon J. PUGLISI    Professeur, University of Helsinki

Examineurs :    Bastien CAZAUX    Maître de conférences, Université de Lille

Élise PRIEUR-GASTON    Maîtresse de conférences, Université de Rouen Normandie

Stéphane VIALETTE    Directeur de recherche CNRS, Université Paris-Est Marne-la-Vallée

Dir. de thèse :    Pierre PETERLONGO    Directeur de recherche Inria, Inria Rennes

Co-dir. de thèse :    Tatiana STARIKOVSKAYA    Maîtresse de conférences, École normale supérieure Paris



# Abstract

---

The simplicity of strings and their impactful usage puts their processing at the heart of many applications, including Bioinformatics, Information Retrieval, and Cybersecurity. Exact pattern matching has been extensively studied [94] as the most natural problem, however, many applications also need more complex queries. Additionally, in all those application fields, the quantity of information to process has been increasing at such a staggering rate [226], that obtaining scalable algorithms is difficult. In this thesis we contribute multiple space- and time-efficient algorithms for various string problems, by relying on sketches: compressions (lossless or lossy) that only keep the essential characteristic of the input needed to answer a given query.

In the first part of this thesis, we study complex queries such as regular expressions search, gapped consecutive matching, and square detection. For regular expression search, we provide a space-efficient algorithm in the streaming model: characters of the text arrive one at a time, and we can only access past characters if we explicitly store them. Next, gapped consecutive matching is a simpler type of query where, given two patterns  $P_1$ ,  $P_2$  and a range  $[a, b]$ , one must report all consecutive occurrences of  $P_1$  followed by  $P_2$  separated by a distance in  $[a, b]$ . We study this problem in two settings: compressed indexing and pattern matching on a compressed text. Motivated by the importance of periodicity detection, next, we investigate square detection for general alphabets (the most abstract setting where squares can be defined). We give an optimal algorithm which answers an open question asked by Main and Lorentz [27] in 1984.

The second part of this thesis proposes several ways to use approximation toward scaling up to large amounts of data in diverse applications including Bioinformatics. We first study approximate matching, where we must report all occurrences at distance at most  $k$  for a given similarity measure. We provide efficient parametrized algorithms for computing the length of the longest common substring with approximately  $k$  mismatches and to compute all positions of a text where a pattern occurs with dynamic time warping distance at most  $k$ . Finally, we propose a compressed index for redundant collections of next-generation sequencing reads, which takes advantage of alignments to an assembled genome to improve the overall compression but can incur false positive occurrences.

# Résumé détaillé en français

---

La simplicité des chaînes de caractères rendent leur traitement crucial pour de nombreuses applications, telles que la bio-informatique, la recherche d'informations et la cybersécurité. Le problème de la recherche exacte d'un motif a naturellement été largement étudié [94], cependant, de nombreuses applications nécessitent également des requêtes plus complexes. De plus, dans ces domaines applicatifs, la quantité de données à traiter augmente à une vitesse stupéfiante [226], et les complexités des requêtes ne permettent pas toujours de passer à l'échelle. Dans cette thèse, nous proposons plusieurs algorithmes efficaces en temps et en espace pour divers problèmes sur les chaînes de caractères, en nous appuyant sur des « sketches » : des compressions (avec ou sans perte) qui ne conservent que les caractéristiques essentielles de l'entrée pour répondre à une requête précise.

Dans la première partie de cette thèse, nous étudions des requêtes complexes telles que la recherche par expressions régulières, la recherche de motifs consécutifs avec espacement et la détection de carrés. Pour la recherche d'expressions régulières, nous présentons un algorithme utilisant peu d'espace dans le modèle de flot de données (« streaming ») : les caractères du texte arrivent un par un, et nous ne pouvons accéder aux anciens que si nous les avons stockés explicitement. Ensuite, nous étudions la recherche de motifs consécutifs avec espacement, un type de requête plus simple, où étant donnés deux motifs  $P_1$ ,  $P_2$  et un intervalle  $[a, b]$ , il faut renvoyer toutes les occurrences consécutives (sans autres occurrences des motifs entre les deux) de  $P_1$  suivies de  $P_2$  espacées d'une distance comprise entre  $a$  et  $b$ . Nous étudions ce problème sous plusieurs angles : l'indexation compressée et la recherche de motifs dans un texte compressé. Motivés par l'importance de la périodicité, nous étudions ensuite la détection de carrés pour alphabets sans ordres (le cadre le plus abstrait dans lequel les carrés peuvent être définis). Nous fournissons un algorithme optimal et répondons à une question ouverte posée par Main et Lorentz [27] en 1984.

La seconde partie de cette thèse propose quelques utilisations d'approximations pour aider à passer à l'échelle sur des grandes quantités de données, en particulier avec application à la bio-informatique. Nous étudions tout d'abord la recherche approximative de motifs, où nous devons rapporter toutes les occurrences à une distance au plus égale à  $k$  pour une mesure de similarité donnée. Nous fournissons des algorithmes paramétrés efficaces pour calculer la longueur de la plus longue sous-chaîne commune avec environ  $k$  différences, puis pour permettre la recherche de motifs apparaissant avec une distance de « dynamic time warping » au plus  $k$ . Enfin, nous proposons un index compressé pour des collections de lectures de séquençage. Cet index tire parti d'alignements sur un génome assemblé pour améliorer la compression, mais l'index est approximatif car il peut renvoyer des faux positifs lors de ses requêtes.

---

## Contexte de la thèse

Lorsque je réponds à la question classique « Quel est le sujet de ta thèse ? » posée par ma famille et mes amis, je commence toujours par la fonction « Ctrl + F » dans leur éditeur de texte ou leur navigateur web préféré. Cela permet de mettre rapidement en évidence l'une des applications de la recherche exacte des motifs. Mais même mes grands-parents savent immédiatement qu'une recherche efficace dans un texte est possible depuis des décennies et qu'il ne peut s'agir de mon véritable sujet de recherche.

En effet, le problème de la recherche exacte de motif dans un texte a été largement étudié, avec en particulier, le célèbre algorithme de Knuth–Morris–Pratt qui fait partie des algorithmes classiques. Charras et Lecroq ont publié un manuel détaillé [94] sur les différentes solutions pour la recherche exacte de motif dans un texte et ces solutions ont également été comparées dans le détail en pratique [143, 170]. En général, cependant, le besoin de traitement de texte va bien au-delà de la recherche exacte de motifs. Dans cette thèse, nous regroupons les problèmes de traitement de chaînes de caractères étudiés en trois grandes catégories : la recherche de motifs complexes, le calcul de distance et mesure de similarité, et enfin la détection de répétitions. Nous détaillerons ensuite la nécessité d'avoir des algorithmes ultra-efficaces pour essayer de passer à l'échelle sur les grandes quantités de données des domaines applicatifs utilisant le traitement de chaînes de caractères.

## Traitement des chaînes de caractères

**Recherche de Motifs Complexes.** L'un des modèles le plus utilisé et classique pour les requêtes complexes est la recherche par expressions régulières, introduite par Kleene en 1951 [3]. Le formalisme des expressions régulières permet une description concise d'ensembles de chaînes par des combinaisons récursives de caractères d'un alphabet  $\Sigma$  ainsi que de trois opérateurs fondamentaux : la concaténation ( $\cdot$ ), l'union ( $|$ ) et l'étoile de Kleene ( $*$ ). Pour deux expressions rationnelles  $R_1$  et  $R_2$ , la concaténation  $R_1 \cdot R_2$  reconnaît toute concaténation d'une chaîne reconnue par  $R_1$  et d'une chaîne reconnue par  $R_2$ , l'union  $(R_1|R_2)$  reconnaît toute chaîne reconnue par  $R_1$  ou  $R_2$ , et  $(R_1)^*$  reconnaît toutes répétitions d'une chaîne reconnue par  $R_1$ , y compris l'absence de répétition, c'est-à-dire la chaîne vide. L'utilisation des expressions régulières a gagné en popularité dans les années 1970 grâce à leur mise en œuvre efficace dans les outils Unix tels que **awk**, **grep**, ou **sed**. Elles sont devenues un outil crucial dans de nombreux domaines tels que l'analyse du trafic internet [114, 110], les bases de données, l'exploration de données [82, 72, 73], les réseaux informatiques [107], et la recherche de protéines [75]. Nous étudions les expressions régulières dans le Chapitre 1.

Malheureusement, Backurs et Indyk [212] suivis par Bringmann, Grønlund, et Larsen [238] ont prouvé des bornes inférieures conditionnelles qui impliquent que certaines expressions régulières ne peuvent probablement pas être recherchées en temps fortement sous linéaire. Comme alternative plus simple, Fischer et Paterson [14] ont introduit la recherche de motifs avec « don't care » où un symbole don't care (aussi appelé wildcard ou gap), dénoté '?', peut apparaître à la fois dans le motif et dans le texte, et correspond à n'importe quel autre caractère de l'alphabet. Ce modèle a été directement appliqué dans la base de données de protéines PROSITE [106] où les caractères génériques sont pris en

---

charge. Plus généralement, les space seeds [96], un concept similaire où seules certaines positions doivent correspondre, ont été utilisées dans la recherche d'homologie [85], l'alignement [151], l'assemblage [196], et la métagénomique [197]. Les motifs avec don't cares sont parfois [152] décrits comme  $P = P_1 g_1 P_2 g_2 \dots g_\ell P_{\ell+1}$  où  $P_1, P_2, \dots, P_{\ell+1}$  sont des motifs sur l'alphabet  $\Sigma$  et  $g_1, g_2, \dots, g_\ell$  sont le nombre de ? entre deux motifs. Naturellement, cette question a ensuite été étendue au problème de la recherche de motifs avec espacement variable [155, 175] où la longueur des espaces peut varier dans des intervalles  $[a_i, b_i]$  pour  $i \in [1, \ell]$ . Les espacements de longueur variable sont également pris en charge par la base de données PROSITE [106]. Différentes variantes du problème ont été étudiées [219, 129, 63], y compris une version plus simple avec seulement deux motifs  $P_1$  et  $P_2$  et un seul espace [108, 132] et le cas spécial  $P_1 = P_2$  [86, 115].

En 2016, Navarro et Thankatchan [228] ont proposé une variante naturelle : étant donné un motif unique  $P$  et un intervalle  $[a, b]$ , on doit déclarer toutes les occurrences *consécutives* de  $P$  commençant aux positions  $(i, j)$  (consécutif signifiant aucune autre occurrence entre  $i$  et  $j$ ) tel que  $j - i$  appartient à  $[a, b]$ . Depuis, les occurrences consécutives ont été étudiées dans plusieurs publications [348, 364, 347]. Récemment, Bille et al. [349] ont proposé une combinaison des modèles de recherche de motifs consécutifs et de recherche de motifs avec espacement : la recherche de motifs consécutifs avec espacement. Dans ce modèle que nous étudions dans les Chapitres 2 et 3, on nous donne deux motifs  $P_1, P_2$  et un intervalle  $[a, b]$ , et il faut renvoyer toutes les occurrences consécutives de  $P_1$  suivies de  $P_2$  espacées d'une distance dans  $[a, b]$ .

**Mesures de Similarité.** De nombreuses applications de chaînes de caractères doivent composer avec la présence de bruit dans les données d'entrée, ce qui rend difficile la recherche de correspondances exactes. Les modèles tels que « don't care » et « variable length gap matching » présentés précédemment définissent leur match de manière à prendre en compte les données d'entrée bruitées, mais une autre approche consiste à travailler avec des distances et des mesures de similarité. La quantification du degré de similarité et de dissimilarité de deux chaînes de caractères est par exemple nécessaire en bio-informatique [57], en analyse musicale [41] et en détection de plagiat [116]. Une mesure de distance quantifie la dissimilarité entre les chaînes de caractères, tandis qu'une mesure de similarité quantifie le degré de ressemblance entre les chaînes de caractères. La plupart des distances peuvent être dérivées en mesures de similarité, et la plupart des mesures de similarité ont une distance correspondante. Dans cette section, nous alternons donc les deux termes en fonction de la forme la plus courante dans la littérature.

Différents types de bruit peuvent apparaître dans les données, tels que le remplacement, l'insertion ou la suppression de caractères, ou encore l'étirement ou la réorganisation de sections du texte. Par conséquent, diverses mesures de similarité peuvent être définies pour tenir compte des différents types de bruit, l'objectif étant toujours que quelques modifications ne changent pas radicalement la distance par rapport à d'autres chaînes de caractères. L'une des distances les plus simples sur les chaînes de caractères est la distance de Hamming : pour deux chaînes  $X$  et  $Y$  avec  $|X| = |Y|$ , il s'agit du nombre de différences entre  $X$  et  $Y$ . La distance de Hamming est également définie comme le nombre de substitutions nécessaires pour transformer  $X$  en  $Y$ . Lorsque l'objectif est plutôt d'avoir une mesure de similarité robuste aux insertions et aux suppressions dans les chaînes  $X$  et  $Y$ , on peut considérer la longueur de la plus longue sous-séquence commune entre  $X$  et  $Y$  : le plus grand  $\ell$  tel qu'il existe des positions  $i_1 < \dots < i_\ell$  et  $j_1 < \dots < j_\ell$

---

telles que  $X[i_p] = Y[j_p]$  pour tout  $p \in [1, \ell]$ . Notez que la plus longue sous-séquence commune est à la base d'outils de comparaison tels que `diff` qui sont ensuite appliqués dans des systèmes de contrôle de version tels que git. Pour la distance de Levenshtein [8] (également appelée distance d'édition par la suite), les opérations autorisées sont les substitutions, les insertions et les suppressions, toutes avec un coût de 1. Cette définition peut être généralisée pour permettre aux coûts de différer pour chacune des opérations (distance d'édition pondérée) ou même pour que le coût dépende du caractère qui est ajouté, supprimé ou substitué (distance d'édition pondérée en fonction de l'alphabet). Cette métrique est l'une des plus connues, en raison de l'importance de la recherche d'alignements globaux (alignements de deux chaînes complètes avec substitutions, insertions et suppressions) en bio-informatique [57]. Malheureusement, Backurs et Indyk [254] ont prouvé une borne inférieure conditionnelle (basée sur SETH) qui suggère qu'il est peu probable que la distance d'édition soit calculable en temps fortement sous-quadratique. Pour tenter de contourner cette limite inférieure, dans le Chapitre 5, nous considérons la plus longue sous-chaîne commune (LCS) avec environ  $k$  différences comme une version approximative d'une mesure résiliente aux substitutions : LCS avec  $k$  différences. Dans le problème LCS avec  $k$  différences, étant donné un entier  $k$  et deux chaînes  $X$  et  $Y$ ,  $\text{LCS}_k(X, Y)$  est la longueur maximale d'une sous-chaîne (qui doit être continue, contrairement à une sous-séquence) de  $X$  qui apparaît dans  $Y$  avec au plus  $k$  différences. Mais là encore, Kociumaka, Radoszewski et Starikovskaya [294] ont montré (sous réserve de SETH) qu'il existe  $k = \Theta(\log n)$  tel que LCS avec  $k$  différences ne peut être résolu en temps fortement sous-quadratique, ils ont donc introduit LCS avec environ  $k$  différences dans le but de rendre le problème plus facile par approximation. Dans le Chapitre 5, nous étudions ce problème, où nous recevons une constante  $\varepsilon > 0$ , et nous devons retourner une sous-chaîne de  $X$  de longueur au moins  $\text{LCS}_k(X, Y)$  qui apparaît dans  $Y$  avec au plus  $(1 + \varepsilon) \cdot k$  différences. Nous fournissons deux nouveaux algorithmes avec des compromis spatio-temporels différents et évaluons l'efficacité pratique de l'un d'entre eux par rapport à la solution de programmation dynamique quadratique pour LCS avec  $k$  différences.

Outre la comparaison directe de deux chaînes de caractères, les distances sont également utilisées pour définir des problèmes de recherche approximative de motifs [32, 39] où, pour un entier donné  $k$ , un motif  $P$  et un texte  $T$ , on doit trouver toutes les positions  $j$  telles qu'il existe une sous-chaîne  $T[i..j]$  qui est à une distance maximale de  $k$  par rapport à  $P$ . Pour un aperçu des résultats obtenus avant 2001 sur la recherche approximative pour les distances d'édition, de Hamming et de la plus longue séquence commune, voir [74]. Nous étudions l'appariement approximatif dans le Chapitre 6 pour une distance populaire pour les séquences temporelles, qui est moins courante pour les chaînes de caractères : la « Dynamic Time Warping (DTW) distance » [18]. Pour les chaînes de caractères, la distance DTW peut être décrite comme suit : dupliquer certains caractères dans le but d'obtenir des chaînes de longueur égale et de minimiser la somme des distances entre les caractères situés aux mêmes positions, cette somme étant la distance DTW.

**Détections de Répétitions.** Après avoir abordé les modèles de recherches de motifs et les mesures de similarité, nous passons maintenant à une autre tâche centrale du traitement des chaînes de caractères : la détection de répétitions. La localisation des fragments répétés dans une chaîne de caractères peut être utilisée pour détecter les données dupliquées qui doivent être supprimées ou pour compresser leur représentation. Par exemple, la

---

décomposition d'une chaîne en fragments maximaux apparus précédemment est à la base de la factorisation de Lempel–Ziv [17]. Une fois détectées, les régions hautement répétitives d'une chaîne peuvent également être traitées différemment, permettant d'obtenir un algorithme plus efficace.

En musique [38], génomique [275], finance [112] et astronomie [20]<sup>1</sup>, de nombreuses données présentent des signes de répétitions et de phénomènes périodiques. Formellement, on dit qu'une chaîne  $T$  de longueur  $n$  est périodique avec une période  $p$  si pour tout  $0 \leq i < n - p$ ,  $T[i] = T[i + p]$ , et la plus petite période de  $T$  est simplement appelée la période de  $T$ . Mais souvent, les données d'entrée ne contiennent que des sous-chaînes périodiques (au lieu d'être entièrement périodiques), ce qui conduit naturellement au concept de « runs ». Les runs sont des sous-chaînes maximales périodiques, c'est-à-dire qu'une sous-chaîne  $T[i..j]$  est un run si elle est périodique, soit  $p$  sa période, et  $T[i - 1..j]$  et  $T[i..j + 1]$  (si elles sont bien définies, c'est-à-dire  $0 < i$  et  $j < |T| - 1$ ) ne sont pas périodiques de période  $p$ .

Un modèle plus simple souvent considéré est le carré : une sous-chaîne  $T[i..i + 2k - 1]$  telle que  $T[i..i + k - 1] = T[i + k..i + 2k - 1]$ , est appelé un carré ou un tandem. Cette forme de répétition est naturellement présente dans l'ADN et joue un rôle important dans l'établissement des empreintes génomiques [90, 244]. L'étude des carrés dans les chaînes de caractères remonte à 1906 avec les travaux de Thue [1] sur la construction d'un mot infini sans carré. En termes d'algorithme, la question la plus fondamentale est de tester si une chaîne de longueur  $n$  contient au moins un carré, et elle a été examinée pour la première fois par Main et Lorentz [27] qui ont conçu un algorithme en temps  $\mathcal{O}(n \log n)$ . Ils ont utilisé une approche en « diviser pour régner » pouvant être adaptée pour obtenir une représentation compacte de tous les carrés dans le même temps. Ils ont également montré que  $\Omega(n \log n)$  comparaisons (vérifier si deux caractères sont égaux) sont nécessaire pour tester la présence de carré. Cependant, leur preuve utilise des chaînes avec jusqu'à  $n$  caractères distincts et ils ont laissé explicitement ouverte la question de savoir si un algorithme plus rapide pouvait être conçu lorsque la taille de l'alphabet est restreinte. Dans le Chapitre 4, nous montrons qu'en effet, pour une chaîne sur un alphabet non ordonné de taille  $\sigma$ , il existe un algorithme pour tester si elle contient un carré en temps  $\mathcal{O}(n \log \sigma)$  n'utilisant que des tests d'égalité pour comparer les caractères. Nous montrons également que ce résultat est optimal pour les algorithmes déterministes et nous étendons notre solution pour permettre de renvoyer de tous les runs de la chaîne.

## Difficulté du Passage à l'Échelle

Nous avons vu jusqu'à présent comment les tâches de traitement des chaînes de caractères ont des applications cruciales pour d'autres domaines appliqués, mais un autre défi majeur dans la plupart des applications est le passage à l'échelle sur de grands jeux de données. Les jeux de données manuellement gérés restent généralement de petite taille. Par exemple, les pages anglaises de Wikipédia (uniquement le texte et les métadonnées) occupaient 20 gigaoctets dans un format compressé en 2022 [366]. En comparaison, toute forme d'archivage et d'historique des versions tend à être beaucoup plus volumineuse. Les

---

<sup>1</sup>Lorsque Jocelyn Bell a découvert pour la première fois des signaux provenant de pulsars (une étoile à neutrons en rotation qui crée un effet de phare), leurs régularités étaient si surprenantes que des spéculations ont été faites sur le fait qu'ils pourraient être des signaux d'une intelligence extraterrestre.



---

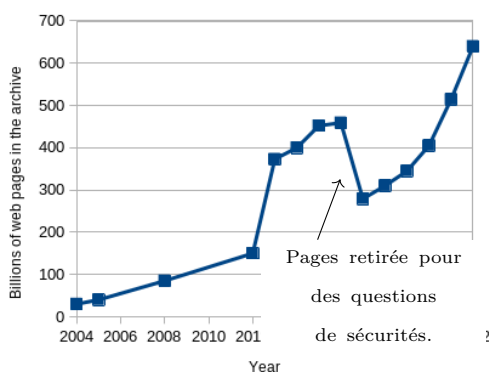
métadonnées de l'historique des révisions (sans le contenu des articles) pour les pages anglaises de Wikipédia occupaient à elles seules 75 gigaoctets en 2022. Il est parfois possible de limiter la redondance dans des données archivées, par exemple en utilisant un graphe indiquant les endroits où les données sont répétées. C'est l'approche adoptée par le projet Software Heritage [367], qui vise à archiver l'intégralité du code logiciel produit par l'humanité. La structure du graphe est particulièrement nécessaire dans ce projet pour refléter l'utilisation courante d'historiques de versions dans le développement de logiciels. Des efforts considérables de recherche et d'ingénierie [342] ont été déployés pour permettre une navigation efficace dans le graphe. Cependant, comme les dépôts de code sont indexés sur la base de leurs URL et de leurs métadonnées, il n'est actuellement pas possible d'effectuer une recherche spécifique pour trouver les occurrences d'un extrait de code particulier<sup>2</sup>. En 2023, le graphe occupe seulement 7 terabytes, mais avec les fichiers source, l'archive occupe proche de 1 petabyte [368]. Un autre exemple de grand projet d'archivage est l'Internet Archive, une organisation à but non-lucratif qui a commencé à sauvegarder des pages web en 1996 et qui détient aujourd'hui un historique pour plus de 800 milliards de pages web grâce à son programme : la Wayback Machine [369]. Cette archive occupe plus de 70 pétaoctets et continue de grandir rapidement (voir Figure 1). Là encore, les options de recherche sont limitées aux métadonnées des sites web et non aux contenus des pages web elles-mêmes.

De grandes archives de chaînes de caractères existent également en bio-informatique, mais la structure des séquences biologiques est très différente de celle de programmes ou de pages web, qui sont typiquement structurés, avec des liens, un grand alphabet et un nombre raisonnable de mots distincts. L'information génétique codée dans l'ADN peut être abstraite comme une simple chaîne sur l'alphabet des nucléotides {A, T, C, G}. Nous obtenons cette chaîne après séquençage et assemblage. Lorsqu'un génome est séquençé, le résultat est un ensemble de fragments, appelés *lectures*, extraits de la séquence originale. Les lectures peuvent contenir des erreurs de séquençage, notamment des insertions, des suppressions et des substitutions de nucléotides. La longueur typique et le taux d'erreur des lectures varient en fonction des techniques de séquençage. Dans tous les cas, la position originale de la lecture dans le génome est perdue pendant le séquençage et les lectures doivent être alignées et fusionnées afin de reconstruire le génome original. Ce problème est appelé l'assemblage de séquences. Pour rendre cette reconstruction possible, les lectures sont extraites en quantités telles que chaque position du génome original est couverte plusieurs fois. Cela rend les ensembles de lectures plus grands et plus redondants que le génome assemblé. Une difficulté supplémentaire réside dans le fait que l'ADN est composé de deux chaînes de nucléotides complémentaires (appariement A-T, C-G). Au cours du séquençage, ces chaînes complémentaires sont détachées et traitées dans des directions opposées, et les lectures proviennent des deux. Ainsi, lors de l'assemblage ou de l'alignement sur une référence, il est nécessaire de prendre en compte le complément inversé d'une lecture.

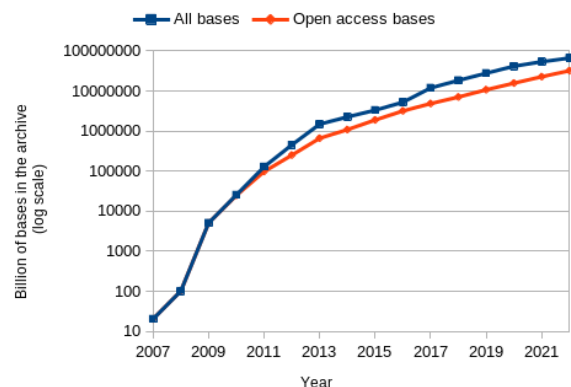
Les applications bio-informatiques travaillent avec des séquences qui sont généralement redondantes, soit en raison de régions répétées au sein d'un génome (redondance intra-génome), soit en considérant plusieurs génomes (de la même espèce) qui partagent

---

<sup>2</sup> Un exemple simple mais intéressant est [190] où l'auteur a recherché `"const double epsilon ="` (et les équivalents dans d'autres langages) sur tous les dépôts GitHub pour étudier la valeur que les programmeurs choisissent généralement pour epsilon.



(a) The Wayback Machine Archive



(b) The Sequence Read Archive

FIGURE 1 : Graphes de l'évolution des tailles de bases de données pour les archives the Wayback Machine [373] et the Sequence Read Archive [372]. Ces bases données sont non seulement déjà très grandes, mais aussi grandissent toujours rapidement.

des parties de leurs génomes (redondance inter-génome). L'exploitation de cette redondance est essentielle pour fournir des algorithmes efficaces, pour des problèmes tels que l'indexation d'une chaîne de caractères. Si l'ADN est stocké sous la forme d'une chaîne de nucléotides  $\{A, T, C, G\}$ , il n'utilise que 2 bits par base, mais nécessite un parcours linéaire de l'ensemble de la chaîne pour rechercher un motif. L'arbre des suffixes est une structure classique qui permet une analyse plus efficace des séquences, mais qui nécessite 10 octets par base [227], ce qui représente 30 gigaoctets pour un génome humain contenant 3,3 milliards de bases. Cela n'est pas envisageable pour les projets où des milliers de génomes ont été séquencés, comme le projet 1000 génomes [200] achevé en 2015 et le projet 100K génomes [370] achevé en 2018. Heureusement, les structures de données compactes qui exploitent la redondance pour réduire l'utilisation de l'espace offrent un compromis intéressant : pour un génome humain, elles permettent de représenter la séquence et son arbre à suffixes en utilisant seulement 4 gigaoctets [227].

Les algorithmes efficaces sont essentiels pour la bio-informatique, car depuis 2008, on a assisté à une diminution drastique du coût du séquençage combiné à une augmentation du débit, entraînant une augmentation massive des volumes d'ADN séquencés. À ce jour, l'Archive européenne des nucléotides (ENA) a accumulé plus de 50 pétaoctets [371] de données de séquençage. Alors que le NCBI Sequence Read Archive possède plus de 73 pétabases [372] de données, dont 38 pétabases en libre accès, et continue de croître, voir Figure 2. Cependant, comme pour le Software Heritage project et internet archive, dans l'ENA et le NCBI, les données sont indexées uniquement sur la base de leurs métadonnées. Dans le Chapitre 7, nous proposons un index compact spécialement conçu pour les ensembles de lectures courtes, qui tire parti de l'entrée redondante pour obtenir une meilleure compression.

---

## L'Utilisation de Sketches

Jusqu'à présent, nous avons présenté les deux principaux défis au cœur du traitement de chaînes de caractères moderne : permettre des requêtes pertinentes (et parfois complexes) adaptées à des applications spécifiques tout en maintenant des performances qui permettent de passer à l'échelle sur de grands volumes de données. Cette thèse propose de nouveaux compromis, théoriques et pratiques, entre les requêtes complexes et efficaces. Pour cela, nous nous reposons sur l'utilisation de **sketches**. Dans cette thèse, un sketch est une compression avec ou sans perte qui ne conserve que les caractéristiques essentielles des données d'entrée nécessaires pour répondre à une requête donnée, offrant ainsi un potentiel prometteur pour le passage à l'échelle. Parmi les exemples de sketches pour la compression avec perte, on peut citer les empreintes de Karp–Rabin [35] (voir Préliminaires 4.3) qui occupent un espace constant et permettent de vérifier si deux chaînes de caractères sont égales avec une probabilité élevée. Mais les empreintes ne contiennent pas en elles-mêmes suffisamment d'informations pour reconstruire les données d'origine. Pour la compression sans perte, un exemple est la factorisation de Lempel–Ziv [17], une compression très efficace en pratique utilisée dans des formats de compression tels que **png** ou **zip**, qui permet toujours de reconstruire la chaîne de caractères originale, mais dans le pire des cas, la factorisation de Lempel–Ziv peut occuper autant d'espace que l'entrée d'origine.

## Contributions

La **Partie I** se concentre sur une étude théorique des requêtes complexes. Nous commençons par la recherche d'expressions régulières dans le modèle des données en streaming. Nous supposons que l'on nous donne une expression régulière  $R$ , et un texte en streaming  $T$  de longueur  $n$ . Pour le problème d'*appartenance* à une expression rationnelle, nous devons déterminer, après avoir vu  $T$  entièrement, s'il est reconnu par l'expression rationnelle  $R$ , tandis que pour la *recherche*, nous devons répondre, à chaque position  $r$ , s'il existe une position  $l$  telle que la sous-chaîne  $T[l..r]$  est reconnue par  $R$ . Dans le **Chapitre 1**, notre principale contribution est d'identifier  $d$ , le nombre de symboles d'union et d'étoiles de Kleene dans  $R$ , comme le paramètre clé qui permet un algorithme de streaming efficace en espace. Auparavant, Bille et Thorup [140]<sup>3</sup> avaient déjà utilisé ce paramètre pour proposer des algorithmes permettant de résoudre l'appartenance et la recherche d'une expression régulière de longueur  $m$  en  $\mathcal{O}(m)$  espace et  $\mathcal{O}(n(\frac{d \log w}{w} + \log d))$  temps, où  $w$  est la taille du mot machine. Mais il restait à savoir si  $d$  pouvait être utilisé pour une solution efficace en terme d'espace. Nous répondons à cette interrogation en fournissant des algorithmes randomisés Monte Carlo (le temps d'exécution est déterministe, mais les algorithmes peuvent se tromper avec une faible probabilité) permettant de résoudre les problèmes d'appartenance et de recherche d'une expression régulière en espace  $\mathcal{O}(d^3 \text{polylog } n)$  et un temps par caractère en  $\mathcal{O}(nd^5 \text{polylog } n)$  (Théorème 1.27).

Voici un bref résumé de la façon dont nous prouvons notre résultat : nous commençons par définir les *chaînes atomiques* qui sont les « mots » apparaissant dans l'expression régulière. Elles ne contiennent que des caractères de  $\Sigma$  et il y en a  $\Theta(d)$ . Par exemple, pour

---

<sup>3</sup>Ils considèrent en fait  $k$ , le nombre de chaînes de caractères apparaissant dans  $R$ , mais  $k = \Theta(d)$ .

---

$R = \text{GAT}(\text{TA}|\text{O})(\text{CAT})^*$  l'ensemble des chaînes atomiques est  $\{\text{GAT}, \text{TA}, \text{O}, \text{CAT}\}$ . La base de notre approche consiste à stocker efficacement certaines occurrences des préfixes des chaînes atomiques dans le texte  $T$ . Ces occurrences stockées sont ensuite liées pour tester s'il existe une correspondance « partielle » de  $R$  (Définition 1.5). Sur les régions périodiques du texte, il peut y avoir trop d'occurrences pour toutes les stocker. Nous choisissons donc de ne stocker que quelques-unes de ces occurrences qui peuvent être très éloignées les unes des autres, avec juste une longue sous-chaîne périodique entre les deux. Pour reconstruire une correspondance partielle, nous devons vérifier si cette longue sous-chaîne périodique correspond à une exécution de l'automate de Thompson [9]. Nous formulons cela comme la recherche d'un chemin de poids spécifique dans un multigraphe. Nous résolvons ensuite efficacement ce problème de graphe en le traduisant en un circuit utilisant des portes d'addition et de convolution qui peuvent être évaluées de manière efficace en terme d'espace à l'aide d'un système général [145, 237]. En outre, nous améliorons ce système en supprimant sa dépendance à l'Hypothèse de Riemann étendue (Théorème 1.33). Les sketches utilisés dans ce chapitre sont des empreintes de Karp–Rabin (Préliminaires 4.3) utilisées pour détecter les occurrences des préfixes des chaînes atomiques dans l'algorithme de recherche de motifs (Théorème 1.9) que nous utilisons.

Dans le **Chapitre 2**, nous commençons notre étude de la recherche de motifs consécutifs avec espacement. Dans ce problème, on nous donne deux motifs  $P_1, P_2$ , et un intervalle  $[a, b]$  et nous devons trouver toutes les paires de positions  $(i, j)$  dans un texte  $T$  telles qu'une occurrence de  $P_1$  commence à la position  $i$ , une occurrence de  $P_2$  commence à la position  $j$ , il n'y a pas d'occurrence de  $P_1$  ou  $P_2$  commençant dans l'intervalle  $[i + 1, j - 1]$ , et enfin  $j - i \in [a, b]$ . Bille et al. [349] ont introduit ces requêtes et ont donné une borne inférieure conditionnelle indiquant que pour les index (texte traité en amont et les motifs donné ensuite sous forme de requêtes) de taille  $\tilde{O}(|T|)$  (la notation  $\tilde{O}$  cache les facteurs polylogarithmiques), l'obtention d'un temps de requête plus rapide que  $\tilde{O}(|P_1| + |P_2| + \sqrt{|T|})$  contredirait « the Set Disjointness conjecture », même si  $a = 0$  est fixé. En outre, ils ont fourni une borne supérieure non-triviale qui utilise  $\tilde{O}(|T|)$  d'espace et  $\tilde{O}(|P_1| + |P_2| + |T|^{2/3} \text{occ}^{1/3})$  de temps pour rapporter toutes les  $\text{occ}$  occurrences.

Nous supposons  $a = 0$  fixé, et que le texte  $T$  de taille  $n$  est donné comme un programme linéaire  $G$  de taille  $g$ . Un programme linéaire est une grammaire sans contexte générant exactement une chaîne de caractères. Par exemple, la grammaire avec les non-terminaux  $\{A, B, C, D\}$  et les règles  $\{A \rightarrow BC, B \rightarrow \mathbf{b}, C \rightarrow DD, D \rightarrow \mathbf{d}\}$  génère la chaîne de caractères  $\mathbf{bdd}$ . Nous avons choisi ce formalisme, car il permet de capturer la populaire factorisation de Lempel–Ziv à un facteur logarithmique près : une factorisation de Lempel–Ziv de taille  $z$  peut être transformée en un programme linéaire de taille  $\mathcal{O}(z \log n)$  [80, 91]. Notre contribution est de créer un index prenant un espace polynomial dans la taille de la grammaire qui rapporte les occurrences consécutives à distance dans  $[0, b]$  en temps optimal, à des facteurs polylogarithmiques près. Pour rapporter les occurrences consécutives (sans contraintes sur la distance entre  $P_1$  et  $P_2$ ), notre index utilise un espace en  $\mathcal{O}(g^2 \log^4 |T|)$  où  $g$  est la taille de la grammaire (voir Corollaire 2.16). Nous nous appuyons sur une construction efficace d'arbre préfixes compacts (voir Préliminaires 4.2) qui tire parti du fait que les chaînes sont des préfixes et des suffixes de chaînes générées par des non-terminaux. Cette implémentation utilise les empreintes de Karp–Rabin (voir Préliminaires 4.3). Les arbres préfixes compacts sont ensuite augmen-

tés à l'aide d'une décomposition en chemins de poids lourds. Nous réutilisons ensuite cette structure pour notre résultat principal : le Théorème 2.1, avec un index qui rapporte des occurrences consécutives à distance dans un intervalle  $[0, b]$  en utilisant l'espace  $\mathcal{O}(g^5 \log^5(|T|))$ . Le programme linéaire donné en entrée est le principal sketch utilisé dans ce travail, mais l'index que nous construisons forme également un sketch de la grammaire spécifique à la recherche d'occurrences consécutives. L'index du Théorème 2.1 contourne la borne inférieure dans le cas des textes hautement compressibles (tels que  $g^5 \ll n^2$ ). Il s'agit d'un résultat non-trivial puisque certains problèmes ne peuvent échapper à une forte dépendance vis-à-vis de la taille de la chaîne non compressée. Cependant, nous nous attendons à ce que notre complexité en espace soit loin d'être optimale et nous laissons les améliorations ainsi que le cas général avec  $0 \leq a \leq b \leq |T|$  comme des questions ouvertes.

Motivés entre autre par les contraintes d'espace de notre index en  $\tilde{\mathcal{O}}(g^5)$ , dans le **Chapitre 3**, nous abordons le problème dual : la recherche de motifs consécutifs. Dans ce problème, les motifs et le texte arrivent et sont traités en même temps. Notons que pour un texte non compressé, la recherche de motifs consécutifs peut être résolue par un algorithme de recherche linéaire classique, en temps  $\mathcal{O}(|T| + |P_1| + |P_2| + \text{occ})$ , en gardant simplement la trace des occurrences les plus récentes de  $P_1$  et  $P_2$ . Nous montrons qu'une complexité similaire peut être atteinte lorsque le texte est hautement compressible : toutes les occurrences consécutives peuvent être rapportées en temps  $\mathcal{O}(g + |P_1| + |P_2| + \text{occ})$  (voir le Théorème 3.1) où  $g$  est la taille du texte compressé sous forme de grammaire. Nous dérivons ensuite de ce résultat des algorithmes pour la recherche d'occurrences consécutives avec espacement (Corollaire 3.2) et pour la recherche des  $k$  occurrences consécutives les plus proches (Corollaire 3.3). Notre résultat est basé sur l'encodage efficace de « l'information frontalière » récemment introduit par Ganardi et Gawrychowski [353]. Pour un motif donné  $P$ , les informations  $P$ -frontalières d'une chaîne  $S$  stockent les sous-chaînes apparaissant à la fois dans  $P$  et  $S$ . Elles sont choisies pour capturer uniquement les informations nécessaires à la détection de nouvelles occurrences de  $P$  qui pourraient survenir lors de la concaténation d'une chaîne à  $S$ . Les auteurs montrent comment utiliser cet encodage pour déterminer en  $\mathcal{O}(g + |P|)$  temps si  $P$  apparaît dans le texte compressé. Nous étendons leur approche pour rapporter toutes les occurrences croisant la frontière (Lemme 3.10). Nous répétons ensuite cette technique à un deuxième niveau avec des « informations frontalières secondaires » et analysons soigneusement tous les cas pour obtenir le Théorème 3.1. Ici encore, le sketch principal est la grammaire sur laquelle nous travaillons.

Tous les chapitres précédents s'appuient fortement sur la détection de la périodicité pour le design des algorithmes, et il semblait donc naturel d'étudier ce problème dans le **Chapitre 4**. Nous montrons comment rapporter tous les carrés en temps optimal dans le modèle le plus abstrait où ils peuvent être définis : les alphabets généraux (non ordonnés) où la seule opération autorisée est un test d'égalité entre deux caractères. Nous considérons d'abord le problème de la détection des carrés, puis nous étendons notre approche pour rapporter les carrés et runs. En 1984, Main et Lorentz [27] ont conçu un algorithme en  $\mathcal{O}(n \log n)$  temps pour la détection de carrés dans un texte  $T$  de taille  $n$  sur un alphabet général non ordonné. Ils ont également fourni une borne inférieure correspondante pour les chaînes ayant  $\Omega(n)$  symboles distincts, mais ont laissé ouverte la question de savoir si un algorithme plus rapide était possible si la taille de l'alphabet  $\sigma = |\Sigma|$  était restreinte.

---

Nous commençons par prouver que le problème nécessite  $\Omega(n \log \sigma)$  comparaisons même si la taille de l'alphabet est connue (Théorème 4.1). En outre, dans le Théorème 4.9, nous montrons que le calcul de toute approximation pertinente du nombre de caractères distincts nécessite  $\Omega(n\sigma)$  opérations. Pour les alphabets ordonnés généraux (lorsqu'un ordre est donné), Crochemore [30] a utilisé la factorisation  $f$  (liée à la factorisation de Lempel–Ziv) pour donner un algorithme de détection des carrés fonctionnant en temps  $\mathcal{O}(n \log \sigma)$ . En très résumé, la factorisation  $f$  et la factorisation Lempel–Ziv (factorisation LZ) détectent les fragments répétitifs dans le texte et peuvent être calculées efficacement à l'aide d'un arbre à suffixes ou d'un tableau à suffixes. Cependant, nous montrons que pour les alphabets généraux non ordonnés, ces factorisations nécessitent  $\Omega(n\sigma)$  opérations pour être calculées (corollaire de la borne inférieure sur l'approximation de l'alphabet). Au lieu de cela, nous introduisons la factorisation de Lempel–Ziv  $\Delta$ -approchée qui agit comme un sketch capturant uniquement les carrés suffisamment longs (d'une longueur d'au moins  $8\Delta$ ), par opposition à la factorisation  $f$  et à la factorisation LZ qui capturent tous les carrés. Nous présentons notre algorithme final par étapes. Nous supposons d'abord que la taille de l'alphabet est connue et nous nous concentrons sur l'obtention d'une borne supérieure sur le nombre de comparaisons, puis nous supprimons l'hypothèse de la connaissance de la taille de l'alphabet, et enfin nous fournissons un algorithme global efficace fonctionnant en temps  $\mathcal{O}(n \log \sigma)$ .

La **Partie II** explore l'utilisation d'approximations pour réduire encore la taille des sketches et fournir des algorithmes encore plus efficaces. Chaque chapitre fournit une implémentation pratique de ses algorithmes pour des applications en bio-informatique. Plus tôt, nous avons détaillé l'importance des mesures de similarité pour de nombreuses applications. En bio-informatique, la distance d'édition est sans doute la mesure de similarité la plus populaire, mais Backurs et Indyk [254] ont prouvé une borne inférieure conditionnelle (basée sur SETH) suggérant qu'il est peu probable que la distance d'édition soit calculable en temps fortement sous-quadratique. La nécessité de surmonter cet obstacle a conduit à l'étude d'algorithmes approximatifs pour la distance d'édition. Chakraborty et al. [255] ont donné le premier résultat avec un algorithme d'approximation à facteur constant qui calcule la distance d'édition entre deux chaînes de longueur  $n$  en temps  $\tilde{\mathcal{O}}(n^{2-2/7})$ . Depuis notre publication [317] (présentée dans le Chapitre 5), une série de travaux ont été publiés sur l'approximation de la distance d'édition [311, 318], le résultat le plus fort étant [306] avec une approximation à facteur constant en temps  $n^{1+\varepsilon}$  pour tout  $\varepsilon > 0$  (où la constante d'approximation dépend uniquement de  $\varepsilon$ ). Néanmoins, ces algorithmes ont tendance à être assez techniques et même ceux qui sont censés être plus simples, comme [305], ne semblent pas avoir été implémentés et évalués dans la pratique. Dans le **Chapitre 5**, nous adoptons une autre approche en considérant une mesure de similarité différente censée être à la fois robuste aux changements légers et suffisamment simple pour permettre un calcul efficace. Nous considérons la plus longue sous-chaîne commune (abrégée LCS par la suite) avec environ  $k$  différences, qui est une version approximative de LCS avec  $k$  différences. Rappelons que, pour un entier  $k$  et deux chaînes  $X$  et  $Y$ ,  $\text{LCS}_k(X, Y)$  est la longueur maximale d'une sous-chaîne de  $X$  qui apparaît dans  $Y$  avec au plus  $k$  différences. Pour une constante  $\varepsilon > 0$ , le problème LCS avec environ  $k$  différences doit retourner une sous-chaîne de  $X$  de longueur au moins  $\text{LCS}_k(X, Y)$  qui apparaît dans  $Y$  avec au plus  $(1 + \varepsilon) \cdot k$  différences. Ce problème a été introduit par Kociumaka,

---

Radoszewski et Starikovskaya [294] après qu’ils aient montré qu’il existe  $k = \Theta(\log n)$  tel que **LCS with  $k$  Mismatches** ne peut pas être résolu exactement en temps fortement sous-quadratique (conditionné par SETH). Dans le Théorème 5.1, nous fournissons deux algorithmes : l’un supposant un alphabet de taille constante s’exécutant en temps et en espace en  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$ , et l’autre sans contraintes sur l’alphabet s’exécutant en temps  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$  et linéaire en espace. Le premier résultat repose sur une structure de données pour la recherche de plus proches voisins [194] comme boîte noire, et nous ne l’évaluons pas dans la pratique. En revanche, notre deuxième contribution est plus simple et nous confirmons son caractère pratique par une évaluation expérimentale. En outre, dans le « Fait 5.2 », nous montrons une borne inférieure conditionnelle pour LCS avec environ  $k$  différences (avec une construction similaire à la preuve de la borne inférieure de **LCS with  $k$  Mismatches**). Dans nos algorithmes, nous nous appuyons sur les empreintes de Karp–Rabin et sur un sketch estimant la distance de Hamming basée sur la réduction de dimensions, tous deux détaillés dans la section 2.

En poursuivant notre exploration des mesures de similarité et des distances, le Chapitre suivant se concentre sur la distance DTW (Dynamic Time Warping). Pour la distance DTW, il faut « doubler » les caractères des deux chaînes jusqu’à ce que les chaînes soient de même longueur, puis additionner les distances entre les caractères situés aux mêmes positions. Pour proposer un algorithme efficace pour cette distance, dans le **Chapitre 6**, nous considérons l’une des formes les plus simples de sketches : l’encodage par plages (run-length encoding). L’encodage par plages d’une chaîne  $S$  de longueur  $N$  est défini comme  $\text{RLE}(S) = (c_1, l_1)(c_2, l_2)\dots(c_n, l_n)$  où  $(c_i, l_i)$  représente le caractère  $c_i \in \Sigma$  répété  $l_i$  fois pour  $i \in [1, n]$ , et tel que  $\sum_{i \in [1, n]} l_i = |S|$ . Ce sketch est particulièrement pertinent pour la DTW car les séries de caractères égaux ont tendance à être alignées malgré les variations de longueur. C’est pourquoi Froese et al. [285] ont déjà utilisé le nombre de runs dans les chaînes de caractères pour donner un algorithme calculant la distance DTW entre deux chaînes de caractères avec un temps d’exécution  $\mathcal{O}(mN + nM)$ , où  $M, N$  sont les longueurs des chaînes de caractères, et  $m, n$  sont les tailles de leurs encodages par plage.

Notre contribution est la suivante : lorsque les distances entre les caractères de  $\Sigma$  sont des entiers, pour un motif  $P$  avec  $m$  runs et un texte  $T$  avec  $n$  runs, nous montrons qu’il existe un algorithme en  $\mathcal{O}(n + m)$  temps qui calcule toutes les positions  $j$  où la distance DTW entre  $P$  et un suffixe de  $T[..j]$  est au plus de 1. Puis, plus généralement, pour un entier  $k$ , nous fournissons un algorithme en temps  $\mathcal{O}(knm)$  qui calcule toutes les positions  $j$  où la distance DTW entre  $P$  et un suffixe de  $T[..j]$  est au maximum de  $k$ . Notre intérêt et nos recherches sur DTW sont également motivés par des applications potentielles à l’analyse de données biologiques produites par le séquençage de troisième génération. Pour cette technologie, l’ADN passe à travers un nanopore à une vitesse irrégulière, ce qui tend à créer des erreurs dans la longueur des plages du même nucléotide (homopolymères) [331]. Nous détaillons cette application potentielle dans la section 6. Depuis notre publication, Boneh, Golan, Mozes, et Weimann ont mis en ligne une prépublication [361] avec un temps en  $\tilde{\mathcal{O}}(n^2)$  pour le calcul de la distance DTW entre deux chaînes avec  $n$  runs, ce qui est optimal à des facteurs logarithmiques près. Ils suivent l’approche que Clifford et al. [282] ont utilisée pour montrer un résultat similaire pour la distance d’édition : ils représentent et traitent les entrées et les sorties à l’aide d’une fonction linéaire par morceaux.

---

Pour le **Chapitre 7**, nous quittons l'étude des mesures de similarité pour nous concentrer sur le problème pratique de l'indexation des ensembles de lectures de séquençage. De plus, dans ce Chapitre, la perspective sur l'approximation est différente. Au lieu d'étudier la correspondance approximative où l'on rapporte toutes les sous-chaînes à une distance donnée du motif, nous proposons un index compact qui a l'inconvénient de rapporter des faux positifs : des occurrences du motif qui ne sont pas entièrement incluses dans une lecture. Une lecture est une séquence de paires de bases, généralement courte (la longueur précise dépend de la technique de séquençage), obtenue lors du séquençage de l'ADN. Pour pouvoir réassembler l'ensemble de la séquence d'ADN, chaque position de la séquence est généralement couverte par plusieurs lectures. Le nombre moyen de lectures couvrant une position est appelé couverture de séquençage. La couverture de séquençage standard pour les lectures courtes dans le but d'assembler un génome se situe aujourd'hui entre 30 et 50, ce qui rend les ensembles de lectures très répétitifs, en particulier lorsque l'ADN séquençé provient d'un seul individu. Pour indexer une chaîne répétitive unique, l'index FM [99] basé sur la transformation de Burrows-Wheeler (BWT) [48] est l'une des structures de données les plus importantes, et elle a été appliquée dans plusieurs outils bio-informatiques pour l'alignement des lectures courtes [134, 164, 135]. L'amélioration la plus récente du FM-index est le  $r$ -index de Gagie et al. [313] qui occupe un espace proportionnel à  $r$ , le nombre de plages dans la BWT de la chaîne indexée. Plus précisément, la structure de données occupe  $\mathcal{O}(r \log \log n)$  d'espace tout en étant capable de compter et de localiser toutes les occurrences d'un motif en un temps optimal (à un facteur logarithmique près). Cela fait du nombre de plages dans la BWT un paramètre important pour les structures économes en espace et  $r$  est souvent considéré comme une mesure de la répétitivité du texte. Malheureusement, l'extension de la BWT à une collection de chaînes de caractères n'est pas simple. Nous recommandons au lecteur la publication de Cenzato et Lipták [351] qui présente toutes les variantes existantes pour construire une structure de type BWT pour une collection de chaînes et l'impact sur le nombre de plages dans la transformation qui en résulte.

Notre travail dans le Chapitre 7 propose de tirer parti d'une information très couramment associée aux lectures : l'alignement sur un génome de référence. Nous construisons un arbre dont le tronc principal est la référence et les lectures se ramifient à partir du tronc depuis la position sur laquelle elles sont alignées, puis nous calculons la généralisation de la BWT pour les arbres : la transformée de Burrows-Wheeler étendue (XBWT) [130]. Intuitivement, notre transformation fournit un contexte aux lectures qui permet un meilleur tri et limite le nombre de ruptures dans les plages de la chaîne transformée. Formellement, nous montrons que si les lectures sont presque identiques à la référence et que les différences sont situées vers la fin des lectures (ce qui est le cas en pratique pour les lectures courtes), le nombre de plages dans la XBWT peut être limité en fonction du nombre de plages dans la BWT de la référence. Nous testons ensuite notre approche en pratique et montrons qu'elle permet d'obtenir moins de runs que la BWT avec l'heuristique co-lexicographique populaire (15% moins de run que BWT+colex pour un ensemble de lectures du chromosome 19 humain). Le principal inconvénient de notre index est que nous cherchons dans l'arbre de branchement et donc, lorsque nous comptons le nombre d'occurrences d'un motif, nous ne pouvons pas éviter de compter les occurrences qui ne sont pas entièrement contenues dans une lecture (qui sont partiellement ou entièrement dans le tronc principal correspondant à la référence). Une autre contribution de notre tra-



---

vail est que, dans le but de permettre un passage à l'échelle de la construction de l'index, nous adaptons la technique de découpage sans préfixe pour la construction de la BWT inventé par Boucher et al. [280] à la construction de la XBWT. Dans la construction de la BWT par découpage sans préfixe, la chaîne d'entrée est découpée en phrases qui se chevauchent comme suit : nous maintenons un hash d'une fenêtre glissante en utilisant les empreintes de Karp–Rabin, et chaque fois que le hash est égal à zéro, nous terminons la phrase en cours et en commençons une nouvelle. Cette technique, nous permet de créer un ensemble de phrases où aucune n'est un préfixe d'une autre. La chaîne est ensuite décomposée en un *dictionnaire* et un *parse*. Le dictionnaire associe chaque phrase à un métacaractère, et le parse est une chaîne de métacaractères. Cette construction utilise des sketches (empreintes de Karp–Rabin) mais crée également un sketch par le biais de cette représentation compacte de la chaîne. L'intuition est que les sections répétées de la chaîne se traduiront par des phrases répétées qui peuvent être représentées par les mêmes métacaractères dans le parse, ce qui permet au dictionnaire de rester raisonnablement petit. La BWT de la chaîne d'entrée est alors construite à partir de la BWT du parse (où l'ordre entre les métacaractères est l'ordre lexicographique de la phrase qu'ils représentent) et de la BWT des phrases.

L'adaptation de cette technique aux arbres et à la construction de la XBWT n'est pas triviale car chaque branche peut créer de nouvelles phrases. Nous avons implémenté notre index et l'avons évaluée expérimentalement en termes de nombre d'exécutions dans la XBWT résultante. La mise en œuvre des requêtes et l'analyse des faux positifs font partie des travaux futurs.

J'espère que cet aperçu des contributions a clairement mis en évidence le fait que les sketches sont des outils précieux dans de nombreux contextes et applications. Je m'attends à ce qu'ils continuent de se développer dans de nombreux travaux futurs, tant en théorie qu'en pratique.

# Acknowledgement - Remerciements

---

I am grateful to Thierry Lecroq and Simon Puglisi for accepting to review this thesis and to Bastien Cazaux, Élise Prieur-Gaston, and Stéphane Vialette for accepting to be part of my jury. Merci également à Mireille Régnier et Eric Rivals pour leur participation à mon comité de suivi de thèse.

Mes remerciements vont ensuite à Pierre Peterlongo et Tatiana Starikovskaya pour leurs encadrements. Merci d’avoir été compréhensifs et aidants quand j’ai exprimé mes difficultés de santé mentale. Merci de m’avoir offert un environnement scientifique très riche et libre. Pierre, merci d’avoir accepté d’encadrer ma thèse, un peu loin de ta recherche habituelle. J’ai beaucoup apprécié de pouvoir côtoyer ta recherche pratique, simple, efficace et très inspirante. Tatiana, merci de m’avoir transmis un peu de tes très grandes connaissances et compétences de chercheuse et de m’avoir mené vers des projets ambitieux. Malgré votre travail commun sur mes tics de langage, je ne vous promets pas de ne pas dire “truc” où “inte\g\er” pendant ma soutenance de thèse...

Merci à mes deux équipes d’accueil, Talgo à l’ENS à Paris et Symbiose (Genscale) à l’IRISA à Rennes. Il y a bien pire problème que de se retrouver à être triste de quitter un environnement et contente d’en retrouver un autre. À Talgo, je suis reconnaissante d’avoir pu côtoyer<sup>4</sup> Chien-Chung, Pierre A., Kevin, Monika, Guillaume, Gabriel, Quentin, et Benoit. Plus largement, je remercie Michäel<sup>5</sup>, Yann, Yoan et Juliette d’avoir partagé de joyeuses pauses cafés avec moi pendant mon stage de M2 et doctorat, elles m’auront fait beaucoup de bien dans ces journées denses ! Merci également à Lise-Marie, Linda, Fanny et Tiffany pour leur aide administrative précieuse. Merci au DI et la Fondation de l’ENS pour avoir financé certains de mes voyages de recherche et mon extension de thèse de 2 mois. Merci également à Laurent Boyer pour m’avoir permis d’enseigner le Python au L2 MIASH de Paris 1, ce fut un plaisir même avec des cours en distanciel et semi-distanciel.

À Symbiose je remercie toute la grande équipe, dont j’ai énormément apprécié l’ambiance joviale, solidaire, et chaleureuse. Des mercis du fond du cœur vont à Lucas<sup>6</sup>, Matthieu<sup>7</sup>, Nicolas<sup>8</sup>, et Victor<sup>9</sup> pour m’avoir intégré à leur équipe de réalisation de Science en cour[t]s<sup>10</sup>, pour m’avoir hébergé, conseillé et distraite dans les moments difficiles. Merci pour ces très beaux moments d’amitiés, vous êtes tous les bienvenus à Bagneux pour que je vous rende un peu la pareille. Merci à l’enthousiaste Karel pour m’avoir fait découvrir ses passionnantes utilisations de l’informatique théorique appliqué à la bio-informatique et pour tous ses encouragements qui me sont allés droit au cœur ! Un merci particulier à Marie, Jacques, Emanuelle, Clara, Téo, Camille, Olivier, Khonogan, Sandra, Kerian, Baptiste, Khodor, Rolland, Meven, Victor M, Léo et Luca, dont j’ai pu apprécier la compagnie.

I am also deeply grateful for the opportunity of working with Teresa Anna Steiner<sup>11</sup>,

---

<sup>4</sup>Et d’avoir pu partager quelques jeux de société !

<sup>5</sup>En particulier, pour les gouters-sport du DI et les encouragements mutuels sur l’écriture !

<sup>6</sup>Très fière d’être ta jumelle de thèse.

<sup>7</sup>Toujours à l’écoute et réconfortant !

<sup>8</sup>J’espère m’être un peu inspiré de ta précision pour ce manuscrit.

<sup>9</sup>J’admire énormément ton engagement politique.

<sup>10</sup>Festival de courts-métrages de vulgarisation scientifique Rennais, que nous avons ensuite organisé !

<sup>11</sup>I cherish our friendship and found our collaborations extremely motivating.

Pawel Gawrychowski, and Jonas Ellert. I very much enjoyed our shared time in Paris and I hope I can see you all there<sup>12</sup> another time. My gratitude also goes to my former internship supervisors for their guidance : Jakub Radosweski, Travis Gagie, Gonzalo Navarro, and Roberto Grossi. Many thanks to the co-authors who I have not mentioned already : Giulia Bernardini, Alessio Conte, Anne Driemel, Tomasz Kociumaka, Grigorios Loukides, Giovanni Manzini, Nadia Pisanti, Solon P. Pissis, Wojciech Rytter, Giulia Punzi, Arseny Shur, Leen Stougie, Michelle Sweering, and Tomasz Walen. I am also grateful for the community of the CPM and SPIRE conferences who welcomed me with open arms as soon as I arrived.

Merci à ma famille pour tous leurs soutiens et encouragements. Merci à mes parents de m'avoir transmis un peu de leurs passions pour la recherche<sup>13</sup> et tellement plus. Merci à Geneviève, ma sœur adorée pour son soutien indéfectible et pour notre sororité tellement importante. Merci à mes grands-parents, Andrée, Gwen, Marc, y compris ceux qui ne sont plus parmi nous, Daniel, Francis, Paulette, et Jean-Pierre, ils ont tant contribué à faire de moi l'adulte que je suis. Merci à mon chat turbulent, Betisou, de m'avoir inspiré la version BD pour enfant de cette thèse (en fin de manuscrit).

Merci à Éloi pour tout son soutien, son amour, et pour avoir accepté de m'épouser. Je t'aime et je souhaite que nous ayons encore de nombreuses années à se reposer l'un sur l'autre. Merci aussi à ma belle-famille pour leur accueil chaleureux, leur soutien et leur intérêt pour ma thèse qui n'aura cessé de me surprendre.

Merci à tous mes amis de m'avoir écoutée, réconfortée, conseillée, encouragée et distraite (et pour tout le reste) pendant 3 ans, la vie serait tellement moins douce sans vous. En particulier, merci à Antoine P. pour m'avoir beaucoup écouté râler et pour avoir toujours été de très bon conseils. Merci à Paul d'être un ami incroyable, à toute épreuve, présent et calme en toute circonstances. Merci à mes amis de collège-lycée d'être toujours là après tant d'années : Diane, Enguerrand, Johanna, Mathieu, Goshia, et Martin. Merci à mes amis rencontrés par l'ENS, Rémi, Lucas G., Lucas P., mon mentor Jill-Jênn, et Tomoko. Merci à tous mes amis rencontrés par Prologin<sup>14</sup>, Antoine M.<sup>15</sup>, Florian, Joël, Matthieu, Maya E., Thibault, Victor, et Victoria. Vous comptez tous énormément pour moi.

---

<sup>12</sup>Possibly at the zoo.

<sup>13</sup>Attention, vous n'allez plus pouvoir me répondre "Passe ta thèse d'abord!"...

<sup>14</sup>Association dont je suis membre depuis 2018, qui organise un concours d'informatique à l'échelle nationale et des stages d'informatique pour les collégiennes et lycéennes.

<sup>15</sup>Merci pour la correction des typos, le soutien, de m'avoir poussé à prendre du repos et plein de courage pour ta thèse!

# Table of Contents

---

<b>Introduction</b>	<b>1</b>
1 Context . . . . .	1
1.1 Matching Models . . . . .	3
1.2 Similarity Measures and Distances . . . . .	5
1.3 Repetition Detection . . . . .	6
1.4 Scalability Issues . . . . .	8
2 Sketching . . . . .	10
3 Contributions . . . . .	13
4 Preliminaries . . . . .	20
4.1 The Strong Exponential Time Hypothesis . . . . .	20
4.2 Tries and Suffix Trees . . . . .	20
4.3 Karp–Rabin Fingerprint . . . . .	22
4.4 The Fine–Wilf’s Periodicity Lemma . . . . .	22
 <b>I Complex Queries</b>	 <b>25</b>
<b>1 Streaming Regular Expression Membership and Pattern Matching</b>	<b>27</b>
1 Introduction . . . . .	28
1.1 Our Results . . . . .	30
2 Preliminaries . . . . .	31
3 Technical Overview . . . . .	33
4 Membership and Pattern Matching . . . . .	39
4.1 Anchors . . . . .	39
4.2 Algorithms . . . . .	43
5 Proof of Theorem 1.13 . . . . .	48
5.1 Finding Primes . . . . .	49
5.2 Walks in a Weighted Graph . . . . .	52
 <b>2 Compressed Indexing for Consecutive Occurrences</b>	 <b>57</b>
1 Introduction . . . . .	57
2 Preliminaries . . . . .	60
2.1 Grammars . . . . .	60
2.2 Compact Tries . . . . .	62
3 Relevant, Extremal, and Predecessor Occurrences in a Non-terminal . . . . .	64
3.1 Proof of Theorem 2.15 . . . . .	65
4 Compressed Indexing for Close Co-occurrences . . . . .	69
4.1 Combinatorial Observations . . . . .	70
4.2 Index . . . . .	72
4.3 Query . . . . .	73

<b>3</b>	<b>Compressed Consecutive Pattern Matching</b>	<b>81</b>
1	Introduction . . . . .	81
2	Preliminaries . . . . .	82
2.1	Grammars . . . . .	83
3	Boundary Information . . . . .	83
4	Compressed Consecutive Pattern Matching . . . . .	85
4.1	Computing Boundary Information and Crossing Occurrences . . . . .	85
4.2	Reporting Co-occurrences . . . . .	87
5	Gapped and Top- $k$ Consecutive Pattern Matching . . . . .	89
6	Proof of Lemma 3.10 . . . . .	90
<b>4</b>	<b>Run Reporting Over General Alphabets</b>	<b>93</b>
1	Introduction . . . . .	93
2	Preliminaries . . . . .	98
3	Lower Bounds . . . . .	99
3.1	Approximating the Alphabet Size . . . . .	100
3.2	Testing Square-Freeness . . . . .	101
4	Upper Bound . . . . .	104
4.1	Sparse Suffix Trees and Difference Covers . . . . .	105
4.2	Detecting Squares with a Delta-Approximate LZ Factorisation . . . . .	106
4.3	Simple Algorithm for Detecting Squares . . . . .	108
4.4	Improved Algorithm for Detecting Squares . . . . .	108
5	Algorithm . . . . .	110
5.1	Constructing the Suffix Tree and Factorisation . . . . .	111
5.2	Final Improvement . . . . .	114
6	Computing Runs . . . . .	119
6.1	Copying Runs From Previous Occurrences . . . . .	122
6.2	Final Improvement for Computing Runs . . . . .	122
<b>II</b>	<b>Approximate Matching for Bioinformatics</b>	<b>125</b>
<b>5</b>	<b>Approximating Longest Common Substring with <math>k</math> Mismatches</b>	<b>127</b>
1	Introduction . . . . .	127
2	Preliminaries . . . . .	129
2.1	The Twenty Questions Game . . . . .	130
3	LCS with Approximately $k$ Mismatches . . . . .	131
3.1	Proof of Theorem 5.9 . . . . .	132
4	Experiments . . . . .	135
<b>6</b>	<b>Pattern Matching under DTW Distance</b>	<b>139</b>
1	Introduction . . . . .	139
2	Preliminaries . . . . .	141
3	Linear Algorithm for $k = 1$ . . . . .	143
4	Main Result: $\mathcal{O}(kmn)$ -time Algorithm . . . . .	144
5	Approximation Algorithm . . . . .	147
6	Experiments . . . . .	150

<b>7</b>	<b>Compressing and Indexing Aligned Readset</b>	<b>153</b>
1	Introduction . . . . .	154
2	Concepts . . . . .	155
	2.1 BWT and FM-index . . . . .	156
	2.2 EBWT . . . . .	156
	2.3 Wheeler Graphs and XBWT . . . . .	157
3	Our Contribution . . . . .	158
4	XBWT via Prefix Free Parsing . . . . .	161
	4.1 Construction of the Dictionary and the Parse . . . . .	162
	4.2 XBWT of the Parse . . . . .	163
	4.3 Building the Final XBWT . . . . .	163
5	Experiments . . . . .	164
6	Application to the JST . . . . .	166
7	Proof Sketch for Theorem 7.3 . . . . .	168
	7.1 Generalized Toehold Lemma . . . . .	168
	7.2 Generalized $\phi$ . . . . .	169
	7.3 Discussion . . . . .	170
	<b>Conclusion</b>	<b>171</b>
	<b>Bibliography</b>	<b>173</b>
	<b>List of Publications</b>	<b>199</b>



# Introduction

---

When answering the classic question “What is your PhD about?” to family and friends, I always start with the “Ctrl + F” function in their favourite text editor or web browser. This quickly highlights one of the applications of the exact pattern-matching problem. If I feel especially ambitious in my explanations, I will attempt to give the intuition of the naive  $\mathcal{O}(nm)$  algorithm. Picture a young child, aligning the string against every position of the text and comparing character by character because the child has yet to learn how to read. To show a glimpse of a more complex solution, I comment on how, depending on the pattern, the child may try to skip portions of the text. But even my grandparents immediately know that efficient search in a text has been possible for decades and that it cannot be my real research subject.

## 1 Context

The exact pattern matching problem has been extensively studied, with in particular the famous Knuth–Morris–Pratt algorithm<sup>16</sup> published in 1977 [16] after being independently discovered by Morris-Pratt in a technical report in 1970 and Knuth in 1973. Since then, this has become one of the classic textbook algorithms, and Charras and Lecroq published a detailed handbook [94] on the various solutions to exact pattern matching which have also been thoroughly compared in practice [143, 170].

In order to present the context of this thesis, we start with a few basic definitions. We assume the reader to be familiar with the little  $o$ , big  $\mathcal{O}$ ,  $\Omega$ , and  $\Theta$  notations for complexities. For ease of readability, we additionally use the notation  $\tilde{\mathcal{O}}$  that hides poly-logarithmic factors. A string of length  $n$  is a sequence  $T[0] \dots T[n-1]$  of characters from a finite alphabet  $\Sigma$  of size  $\sigma$ . The substring  $T[i..j]$  is the string  $T[i] \dots T[j]$ . We also use the notation  $T[i..j)$  and  $T(i..j]$  which stand for  $T[i..j-1]$  and  $T[i+1..j]$ , respectively. We call the substrings of the form  $T[0..i]$  *prefixes* and use the notation  $T[..i]$ , analogously *suffixes* refer to substrings  $T[j..n-1]$  and are denoted  $T[j..]$ . Given two strings  $P$  and  $T$ , we say that  $P$  occurs in  $T$  at position  $i$  if  $i + |P| \leq |T|$  and  $P = T[i..i + |P| - 1]$ . Note that the notation for substrings can differ slightly from chapter to chapter<sup>17</sup>.

Throughout this thesis, we assume the standard unit-cost word RAM model with words of size  $\theta(N)$  for an input of size  $N$ . In this introduction, *data structures* are seen as algorithms where the complexity analysis is split into two parts: the *construction* and the *query*. The construction is generally more expensive and meant to be performed once, whereas the queries are meant to be fast and performed multiple times with varying inputs, thus the focus is often on improving the space and time needed for the queries.



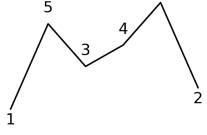
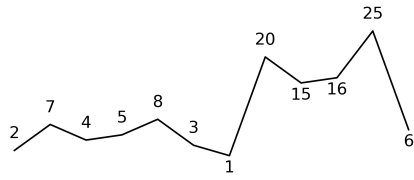
Matching	Pattern	Text with occurrences underlined																														
<b>Regular expression</b> [3]	$P = \text{GAT}(\text{TA} \mid \text{O})(\text{CAT})^*$	<u>GATTAATGATOCATCATA</u>																														
Don't care [14]	$P = \text{GAT}??\text{CAT}$ The ? match any other characters.	<u>GATTACATAGATOACATAC</u>																														
<b>Gapped consecutive</b> [349]	$P_1 = \text{GATTA}$ $P_2 = \text{TAC}$ $a = 2, b = 6$ Consecutive occurrences $(i, j)$ of $(P_1, P_2)$ and s.t. $j - i \in [a, b]$	AGGG <u>GATTACT</u> TAC, distance between $P_1$ and $P_2$ $6 - 3 = 3 \in [a, b]$																														
Degenerate [33]	$P = \left\{ \begin{array}{c} \text{G} \\ \text{C} \end{array} \right\} \text{ATTA}$	<u>GATTAATCCATTACC</u> ATGAAT Some positions can match several characters.																														
Generalized degenerate [252]	$P = \left\{ \begin{array}{c} \text{G} \\ \text{C} \end{array} \right\} \text{AT} \left\{ \begin{array}{c} \text{TA} \\ \text{AT} \end{array} \right\}$	<u>GATTAATCCATTACC</u> ATGAAT Some positions match a set strings of a fixed length.																														
Elastic degenerate [339]	$P = \left\{ \begin{array}{c} \text{G} \\ \text{C} \end{array} \right\} \text{A} \left\{ \begin{array}{c} \text{TTA} \\ \text{T} \end{array} \right\}$	<u>GATTAATCCATTACC</u> ATGAAT Some positions match a set of strings (length can vary).																														
Abelian/Jumbled [95]	$P = \text{GATTACAT}$ Characters can be in reordered.	AGAG <u>TATGAT</u> CAGT																														
Weighted [50]	Each position is given a character probability distribution. GATTA has a cumulative probability $> 0.07$ .	<table><tr><td></td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td></tr><tr><td>A</td><td>0</td><td><u>1</u></td><td>0.25</td><td>0</td><td><u>0.75</u></td></tr><tr><td>C</td><td>0.25</td><td>0</td><td>0.25</td><td>0</td><td>0.25</td></tr><tr><td>G</td><td><u>0.75</u></td><td>0</td><td>0.25</td><td>0.5</td><td>0</td></tr><tr><td>T</td><td>0</td><td>0</td><td><u>0.25</u></td><td><u>0.5</u></td><td>0</td></tr></table>		0	1	2	3	4	A	0	<u>1</u>	0.25	0	<u>0.75</u>	C	0.25	0	0.25	0	0.25	G	<u>0.75</u>	0	0.25	0.5	0	T	0	0	<u>0.25</u>	<u>0.5</u>	0
	0	1	2	3	4																											
A	0	<u>1</u>	0.25	0	<u>0.75</u>																											
C	0.25	0	0.25	0	0.25																											
G	<u>0.75</u>	0	0.25	0.5	0																											
T	0	0	<u>0.25</u>	<u>0.5</u>	0																											
Order preserving [182, 171]	$P = 1 \ 5 \ 3 \ 4 \ 6 \ 2$  Characters must have the same relative order.	<u>2 7 4 5 8 3 1 20 15 16 25 6</u> 																														
Parametrized [45]	$P = \text{GATTACAT}$ Characters can be renamed with a bijection	OPO <u>POGG</u> ODOGO A:O, C:D, G:P, T:G																														

Table 1: Example for various matching models, in bold those we study in this thesis.

## 1.1 Matching Models

However, in general, the need for text processing goes far beyond the exact pattern-matching problem. To illustrate this claim, we present alternative models of matching with their motivations and specify those we study in the following chapters. Figure 1 also provides an example for each matching model.

One of the oldest and most classic models for complex queries is regular expressions search, introduced by Kleene in 1951 [3]. The regular expression formalism offers a concise description for sets of strings through recursive combinations of characters from an alphabet  $\Sigma$  along with three fundamental operators: concatenation ( $\cdot$ ), union ( $|$ ), and Kleene star ( $*$ ). For two regular expressions  $R_1$  and  $R_2$  the concatenation  $R_1 \cdot R_2$  matches any concatenation of a string matching  $R_1$  and a string matching  $R_2$ , the union  $(R_1|R_2)$  allows matching any string matching either  $R_1$  or  $R_2$ , and  $(R_1)^*$  matches any number of repetitions of a string matching  $R_1$ , including no repetition, i.e. the empty string. The use of regular expression gained popularity in the 1970s through their efficient implementation in Unix tools such as `awk`, `grep`, or `sed`. They have become a crucial tool in many fields such as internet traffic analysis [114, 110], databases, data mining [82, 72, 73], computer networks [107], and protein search [75]. Another way to describe regular expressions is through the Thompson automaton construction [9]. This automaton can then be simulated efficiently to test whether a string  $T$  is recognized by a regular expression  $R$  in  $\mathcal{O}(|R| \times |T|)$  time. A series of works [44, 119, 127, 102, 140] has focused on improving the time complexity of regular expression search. However, they only managed to shave off polylogarithmic factors from the  $\mathcal{O}(|R| \times |T|)$  complexity, and a recent fine-grained complexity approach brought an explanation for this. Backurs and Indyk [212] followed by Bringmann, Grønlund, and Larsen [238] considered a subclass of regular expressions called “homogeneous”. A regular expression is “homogeneous” if, in the tree representation of the expression, the operators at each level are equal. For example,  $R = (P_1|P_2|\dots|P_d)^*$  is homogeneous and searching it corresponds to the Word break problem [153, 378]. The authors of [212, 238] showed that for every homogeneous regular expression there is either a solution in near-linear time or it requires  $\Omega((|R| \times |T|)^{1-o(1)})$  time conditioned on the Strong Exponential Time Hypothesis [70]. The only exception is the Word break problem which can be solved in  $\mathcal{O}(|T|(|R| \log |R|)^{1/3} + |R|)$ -time and has a matching lower bound up to polylogarithmic factors. Abboud and Bringmann [251] further detailed those lower bounds using an even finer-grained complexity approach. These results give a good understanding of the time complexity of regular expression in the classical setting, however, multiple practical applications need to work with a stream of input. We specify what we mean by a stream later on in this introduction. Therefore, in Chapter 1, we provide a new space-efficient streaming algorithm for regular expression membership and pattern matching.

Although the versatility of regular expressions makes them widely used in practice across fields, they are notoriously difficult to write for users. As a simpler alternative, Fischer and Paterson [14] introduced the “don’t care” pattern matching where a don’t care (also called wildcard or gap) symbol, denoted  $?$ , can occur in both the pattern and the text, and matches any other character of the alphabet. This model has been

<sup>16</sup>The elegance of this algorithm is what first drew me in this area of research as a bachelor student!

<sup>17</sup>I chose not to unify notations between chapters (corresponding to publications) so the substrings  $T[i..j]$  is denoted  $T[i\dots j]$  in Chapter 2 and 3 and by  $T[i, j]$  in Chapter 5.

directly applied in the PROSITE [106] database of proteins where wildcards are supported. More generally, space seeds [96], a similar concept where only some positions have to be matched, have been used in homology search [85], alignment [151], assembly [196], and metagenomics [197]. Patterns with don't cares are sometimes [152] described as  $P = P_1 g_1 P_2 g_2 \dots g_\ell P_{\ell+1}$  where  $P_1, P_2, \dots, P_{\ell+1}$  are patterns over the alphabet  $\Sigma$  and  $g_1, g_2, \dots, g_\ell$  are the lengths of the maximal stretches of ?. Naturally, this question was later extended to the problem of string matching with variable length gap [155, 175] where the length of the gaps can vary in intervals  $[a_i, b_i]$  for  $i \in [1, \ell]$ . Note that variable length gaps are also supported by the PROSITE [106] database. Different variants of the problem have been studied [219, 129, 63], including a simpler version with just two patterns  $P_1$  and  $P_2$  and a single gap [108, 132] and the special case  $P_1 = P_2$  [86, 115].

In 2016, Navarro and Thankachan [228] proposed a natural variant to pattern matching with a variable length gap, where given a single pattern  $P$  and an interval  $[a, b]$ , one must report all consecutive occurrences of  $P$  starting at positions  $(i, j)$  (consecutive meaning no other occurrence in between  $i$  and  $j$ ) such that  $j - i$  belongs to  $[a, b]$ . Since, consecutive occurrences have been studied in several publications [348, 364, 347]. Recently Bille et al. [349] proposed a combination of the gapped and consecutive lines of research: gapped consecutive matching where we are given two patterns  $P_1$  and  $P_2$  as well as an interval  $[a, b]$  and must report all consecutive occurrences of  $P_1$  and  $P_2$  with distance in  $[a, b]$ . We study gapped consecutive pattern matching in various settings in Chapters 2 and 3, a summary of the contribution is given in Section 3.

Although an in-depth non-standard matching listing is out of scope for this manuscript, for completeness, we detail other models found in the literature, and Table 1 provides examples for each of the models. The modelling of flexible and diverse DNA sequences [10] lead to the model of degenerate strings [33] (also called indeterminate), where each position corresponds to a subset of  $\Sigma$ . For Bioinformatic applications, degenerate strings are most often used as a pattern to search for (like in the example given in Table 1). This model has recently been extended in two directions: elastic degenerate strings [339] where each position is a subset of strings over  $\Sigma$  and generalized degenerate strings [252] where each position is a subset of strings of  $\Sigma^k$ , and the length  $k$  can vary from position to position. Alternatively, when each position is assigned a random variable with values in  $\Sigma$  the strings are called weighted (or uncertain) and can be represented by a weight matrix [50], see Table 1 for an example. Then, the cumulative probability that a string occurs at a position is the product of the probabilities of the corresponding characters at each position, and a match is often defined by a threshold on that cumulative probability. This model has been used in molecular biology<sup>18</sup> in “profiles” that represent multiple aligned strings [34]. In the model of Abelian matching, a string (or a substring) is entirely identified by the characters it contains (with multiplicities), disregarding their order. It stems from the automatic discovery of clusters of genes in genomes where they can occur in a different order but still linked to the same function [95], but the same concept has also been used in the context of using mass spectrometry for DNA assembly [88] where the strings without order are called compomers. This model is also known as jumbled and permutation pattern matching, and several other names, see [141]. The order-preserving model [182, 171] takes a somewhat opposite approach and says that two strings over an integer alphabet

<sup>18</sup>Though they study a slightly different score:  $\log \frac{p(x,i)}{p(x)}$  where  $p(x,i)$  is the probability that the  $i$ th character is equal to  $x$  and  $p(x)$  is the overall frequency of  $x$  in the weighted string.

match if they have the same relative shape:  $\forall i, j \in [0, n-1], X[i] < X[j] \Leftrightarrow Y[i] < Y[j]$ . This matching model aims at capturing the trend detection needed in the stock market and music melody matching problems [182]. Another application-driven model is parametrized strings or “p-string” introduced by Baker [45], where two strings match if we can transform one into the other by applying a function renaming the parameters, meant to detect code duplication.

## 1.2 Similarity Measures and Distances

Many string applications have to work around noise in the input data, which makes it difficult to search for exact matches. Some of the models presented earlier define their match to take into account noisy input, but another important approach is to work with distances and similarity measures. Quantifying the degree of similarity and dissimilarity of two strings is for example needed in Bioinformatics [57], music analysis [41], and plagiarism detection [116]. A distance measure quantifies the dissimilarity between strings, while a similarity measure quantifies the degree of resemblance between strings. There is a natural link between the two terms, and in this section, we alternate between them depending on which form is most common in the literature.

Various types of noise can appear in the input, examples include single characters or entire words being replaced, inserted, or deleted, or some sections of the text being stretched or reordered. As a consequence, various similarity measures can be defined to account for the different types of noise, as the goal is always that a few noisy modifications do not modify drastically the distance to other strings. One of the simplest distances on strings is the Hamming distance: For two strings  $X$  and  $Y$  with  $|X| = |Y|$ , it is the number of mismatches between  $X$  and  $Y$ . Alternatively, the Hamming distance is defined as the number of substitutions needed to transform  $X$  into  $Y$ . When instead the goal is to have a similarity measure robust to insertions and deletions in the strings  $X$  and  $Y$ , one can consider the length of the Longest Common Subsequence between  $X$  and  $Y$ : the largest  $\ell$  such that there exist positions  $i_1 < \dots < i_\ell$  and  $j_1 < \dots < j_\ell$  such that  $X[i_p] = Y[j_p]$  for all  $p \in [1, \ell]$ . The longest common subsequence has been used as the basis of comparison programs like `diff` which are then applied in version control systems like git. For the Levenshtein distance [8] (also called the edit distance hereafter) the operations allowed are substitutions, insertions, and deletions, all with cost 1. A formal definition is given in Table 2, this definition can be generalized to allow costs to differ for each of the operations (weighted edit distance) or even for the cost to depend on which character is being added, removed or substituted (alphabet-weighted edit distance). This metric is one of the most well-known, due to the importance of finding global alignments (alignments of two full strings with substitutions, insertions, and deletions) in Bioinformatics [57]. Unfortunately, Backurs and Indyk [254] proved a conditional lower bound (based on SETH) which suggests that the edit distance between two strings is unlikely to be computable in strongly subquadratic time. As an attempt to circumvent this lower bound, in Chapter 5 we consider the Longest Common Substring (LCS) with Approximately  $k$  Mismatches an approximate version of a measure resilient to substitutions: LCS with  $k$  Mismatches. In the LCS with  $k$  Mismatches problem, given an integer  $k$  and two strings  $X$  and  $Y$ ,  $\text{LCS}_k(X, Y)$  is the maximal length of a substring (must be continuous, unlike a subsequence) of  $X$  that occurs in  $Y$  with at most  $k$  mismatches. But again, Kociumaka,

Radoszewski, and Starikovskaya [294] showed (conditioned on SETH) that there is  $k = \Theta(\log n)$  such that **LCS with  $k$  Mismatches** cannot be solved in strongly subquadratic time, thus they introduced **LCS with Approximately  $k$  Mismatches** to make the problem easier through approximation. In Chapter 5 we study this problem, where we are given a constant  $\varepsilon > 0$ , and we need to return a substring of  $X$  of length at least  $\text{LCS}_k(X, Y)$  that occurs in  $Y$  with at most  $(1 + \varepsilon) \cdot k$  mismatches. We provide two new algorithms with different space-time trade-offs and evaluate the speed of one of them in practice compared to the quadratic dynamic programming solution for **LCS with  $k$  Mismatches**. In Section 3, we further detail our contributions.

Apart from the direct comparison of two strings, distances are also used to define approximate matching problems [32, 39] where for a given integer  $\tau$ , a pattern  $P$ , and a text  $T$ , one must report all positions  $j$  such that there is a substring  $T[i..j]$  that is at distance at most  $\tau$  from  $P$ . For a survey on the results before 2001 on approximate matching for the Edit, Hamming and Longest Common Subsequence distances, see [74]. We study approximate matching in Chapter 6 with regard to a popular distance for temporal sequences, which is less common for strings: the Dynamic Time Warping (DTW) distance [18]. For strings, the DTW distance can be described as follows: duplicate some characters to obtain strings of equal length and to minimize the sum of the distances between the characters at the same positions, this sum is the DTW distance. Computing the edit distance (for a given metric space over the characters) has been reduced to computing DTW [295] (over the same metric space, just with an added null character). All similarity measures and distances mentioned above are illustrated in Figure 2.

### 1.3 Repetition Detection

Having discussed pattern-matching models and similarity measures, we now move to another central task in string processing: repetition detection. Locating repeated fragments inside a string can be useful to detect duplicated data to be removed or to compress its representation. For example, decomposing a string into maximal fragments that appeared earlier (and appending the next symbol) is the basis of the Lempel–Ziv factorization [17]. Once detected, highly repetitive regions of a string can also be treated differently (see [137] for example) allowing for a more efficient algorithm overall.

In music [38], genomic [275], finance [112] and astronomy [20]<sup>19</sup>, a number of datasets present signs of repetitions and periodic phenomenon. Formally, we say that a string  $T$  of length  $n$  has period  $p$  if for all  $0 \leq i < n - p$ ,  $T[i] = T[i + p]$ , and the smallest period of  $T$  is simply called the period of  $T$ . If the period of  $T$  is smaller than  $|T|/2$  we say that the string is periodic. But often, input data only contains periodic substrings (rather than being entirely periodic), which naturally leads to the concept of runs. Runs are maximal substrings that are periodic with a given period, i.e. a substring  $T[i..j]$  is a run if it is periodic with period  $p$ , and  $T[i - 1..j]$  and  $T[i..j + 1]$  (if well-defined, i.e.  $0 < i$  and  $j < |T| - 1$ ) are not periodic with period  $p$ .

A simpler model often considered is squares: substrings  $T[i..i + 2k - 1]$  such that  $T[i..i + k - 1] = T[i + k..i + 2k - 1]$ , it is referred to as a square or a tandem. This form of repetition

---

<sup>19</sup>When Jocelyn Bell first discovered signals coming from pulsars (a rotating neutron star that creates a lighthouse effect), their regularity was so surprising that speculations were made on the fact that they might be signal from extraterrestrial intelligence.

Similarity measure	Example
<p>Hamming distance</p> $\text{HD}(X, Y) = \begin{cases} +\infty & \text{if }  X  \neq  Y , \\  \{i \text{ s.t. } X[i] \neq Y[j]\} , & \text{else.} \end{cases}$	<p><math>X = \text{AAAATG}</math>  <math>\quad \quad \quad    ** </math>  <math>Y = \text{AAATCG}</math>  <math>\text{HD}(X, Y) = 2</math></p>
<p>Length of the Longest Common Subsequence</p> $\text{LCSeq}(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \text{LCSeq}(X_{i-1}, Y_{j-1}) + 1 & \text{if } X[i] = Y[j], \\ \max(\text{LCSeq}(X_{i-1}, Y_j), \text{LCSeq}(X_i, Y_{j-1})) & \text{else.} \end{cases}$	<p><math>X = \text{AAAAGC}</math>  <math>Y = \text{AATC}</math>  <math>\text{LCseq}(X, Y) = 3</math></p>
<p>Levenshtein/Edit distance</p> $\text{ED}(X_i, Y_j) = \begin{cases} \max(i, j) & \text{if } i = 0 \text{ or } j = 0, \\ \min \begin{pmatrix} \text{ED}(X_{i-1}, Y_{j-1}) + d(X[i], Y[j]), \\ \text{ED}(X_{i-1}, Y_j) + 1, \\ \text{ED}(X_i, Y_{j-1}) + 1 \end{pmatrix} & \text{else,} \end{cases}$	<p><math>X = \text{AAAATG}</math>  <math>\quad \quad \quad    \quad  *</math>  <math>Y = \text{AA--TC}</math>  <math>\text{ED}(X, Y) = 3</math></p>
<p><b>Dynamic Time Warping distance</b></p> $\text{DTW}(X_i, Y_j) = \begin{cases} 0 & \text{if } i = 0 \text{ and } j = 0, \\ +\infty & \text{else if } i = 0 \text{ or } j = 0, \\ \min \begin{pmatrix} \text{DTW}(X_{i-1}, Y_{j-1}), \\ \text{DTW}(X_{i-1}, Y_j), \\ \text{DTW}(X_i, Y_{j-1}) \end{pmatrix} + d(X[i], Y[j]) & \text{else,} \end{cases}$	<p><math>X = \overbrace{\text{AAAA}}^{\quad} \text{TG}</math>  <math>\quad \quad \quad    \quad  *</math>  <math>Y = \text{AA TC}</math>  <math>\text{DTW}(X, Y) = 1</math></p>
<p>Length of the Longest Common Substring</p> $\text{LCS}(X, Y) = \max\{l + 1, \exists i, j \text{ s.t. } X[i..i + l] = Y[j..j + l]\}$	<p><math>X = \text{AAAATG}</math>  <math>Y = \text{AATTCG}</math>  <math>\text{LCS}(X, Y) = 3</math></p>
<p><b>Length of the Longest Common Substring with <math>k</math> Mismatches</b></p> $\text{LCS}_k(X, Y) = \max\{l + 1, \exists i, j \text{ s.t. } \text{HD}(X[i..i + l], Y[j..j + l]) \leq k\}$	<p><math>X = \text{AAAATG}</math>  <math>Y = \text{AATTCG}</math>  <math>\text{LCS}_2(X, Y) = 5</math></p>

Table 2: Example for various similarities, in bold those we study in this thesis, for a given metric  $d : \Sigma \times \Sigma \rightarrow \mathbb{Z}^+$  over the alphabet. For two given strings  $X$  and  $Y$  and two integers  $0 \leq i < |X|$  and  $0 \leq j < |Y|$ , for compactness,  $X_i$  and  $Y_j$  denote the prefixes of  $X$  and  $Y$  of length  $i$  and  $j$  respectively.

naturally occurs in DNA and plays an important role in genomic fingerprinting [90, 244]. The study of squares in strings goes back to 1906 with the work of Thue [1] on the construction of an infinite square-free word. In terms of algorithm, the most basic question is testing whether a string of length  $n$  contains at least one square, and it was first considered by Main and Lorentz [27] who designed an  $\mathcal{O}(n \log n)$  time algorithm using a divide and conquer approach to report a compact representation of all squares. At the same time, they showed that  $\Omega(n \log n)$  comparisons (checking if two characters are equal) were necessary, but their proofs used strings with up to  $n$  distinct characters. Therefore, they left as an open question whether a faster algorithm could be designed when the alphabet size is restricted. And indeed, we show in Chapter 4 that for a string over an unordered alphabet of size  $\sigma$ , there is an algorithm for testing whether it contains a square in  $\mathcal{O}(n \log \sigma)$  time using only equality tests to compare characters. We also show that this result is optimal for deterministic algorithms, and we extend it to reporting all runs in the string.

Many other forms of string regularities have been considered in the literature. Although they are out of the scope of this thesis, we list a few of them for completeness. Palindromes are strings that are identical when read backward or forward, formally, a string  $T$  of length  $n$  is a palindrome if  $T[n-1] \dots T[1]T[0] = T[0]T[1] \dots T[n-1]$ . Palindromes are found in music [374, 375] and genomes [345] where they are linked to the regulation of gene replication and implicated in the evolution and development of diseases such as cancer and neurodegenerative disorders. Another way to view repeated fragments is through the lenses of string covers [52]. A string  $C$  is a cover of a string  $T$  if  $T$  occurs in a string constructed by overlapping and concatenating occurrences of  $C$ . A substring of  $T$  is called a seed if it is a cover  $T$  and a cover of minimal length is called a quasiperiod of  $T$ . Those notions generalize the concept of periods in the sense that they additionally allow for superposition of the repetitions, but other approaches have been considered as generalizations. Cadences [235] in a string are characters that are repeated at regular intervals. Formally, given a string  $T[0, n-1]$ , a cadence is a pair  $(i, d)$  such that  $0 \leq i, d < n$  and for every  $k \geq 0$  such that  $i + kd < n$  we have  $T[i] = T[i + kd]$ . Intuitively, a cadence defines a periodicity that is woven regularly inside an aperiodic string. Approximate periodicity has been considered as well. A series of publications [77, 193, 138, 273, 253] studied the problem of deciding whether a given string is close to any periodic string under a given metric. Interestingly, a number of those definitions can be formulated through substring equations and Gawrychowski et al. [314] gave an algorithm that from a set of substring equations over strings of length  $n$  can produce a “generic” solution, meaning that it contains a maximal number of different characters and from which every solution can be generated through character renaming. They gave an algorithm producing a generic solution in  $\mathcal{O}(n)$  which generalizes solutions to various reconstruction problems such as reconstructing a string from its maximal palindromes or runs.

## 1.4 Scalability Issues

So far, we discussed how string processing tasks have crucial applications to other domains, but another major challenge in most applications is scaling up to process large datasets. Highly curated datasets generally remain quite small. For example, the English

pages of Wikipedia (just the text and metadata) take up 20 gigabytes in a compressed format as of 2022 [366]. In comparison, any form of archival and version history tends to grow much bigger. Just the metadata of the revision's history (without the content of the articles) for the Wikipedia English pages occupies 75 gigabytes as of 2022. It is sometimes possible to limit the redundancy in the archival data, for example by using a graph which tracks where the data is repeated multiple times. This is the approach taken by the Software Heritage [367] project, which aims at keeping an archive of all the software code produced by humanity. The graph structure is especially necessary for this project to reflect the standard use of version history management in software development. Substantial research and engineering efforts [342] were made to provide efficient navigation of the graph. However, since the code repositories are indexed based on their URLs and metadata, it is not currently feasible to perform a search specifically for occurrences of a particular code snippet<sup>20</sup>. As of 2023, the graph is limited to 7 terabytes but with the source files the space usage approaches 1 petabyte [368]. Another example of a large archival project is the internet archive, a non-profit which started saving web pages in 1996 and now holds the history of more than 800 billion web pages through their program: the Wayback Machine [369]. This archive takes up more than 70 petabytes, and is still growing quickly (see Figure 2). Here, again, the search options are limited to the metadata of the websites and not the content of the web pages themselves.

Large archives also exist in Bioinformatics, but the structure of biological sequences is very different from that of code or webpages, which are typically structured, with links, a large alphabet, and a reasonable number of distinct words. The genetic information encoded in DNA can be abstracted as just a string over the nucleotide alphabet  $\{\text{A}, \text{T}, \text{C}, \text{G}\}$ . We obtain this string after sequencing and assembling. When a genome is sequenced, the output is a set of fragments, called *reads*, extracted from the original sequence. Reads can contain sequencing errors, including nucleotide insertions, deletions, and substitutions. The typical length and error rate of the reads vary depending on the sequencing techniques. In any case, the original position of the read in the genome is lost during sequencing and the reads have to be aligned and merged in order to reconstruct the original genome, this problem is called sequence assembly. To make this reconstruction possible, the reads are extracted in such quantities that each position of the original genome is covered multiple times. This makes readsets larger and more redundant than the assembled genome. An additional difficulty is that DNA is composed of two chains of complementary nucleotides (A-T, C-G pairing). During sequencing, those complementary chains are detached and processed in opposite directions, and reads come from both chains, thus, when assembling or aligning to a reference it is necessary to consider the reverse complement of a read.

Bioinformatics applications work with sequences that are generally redundant either from repeated regions in a genome (intra-genome redundancy) or by considering several genomes (from the same species) that share portions of their genomes (inter-genome redundancy). Exploiting this redundancy is key to providing efficient algorithms, an example of that is the problem of indexing a single string. If the DNA is stored just as a string over the nucleotide alphabet  $\{\text{A}, \text{T}, \text{C}, \text{G}\}$ , it uses just 2 bits per base but requires

<sup>20</sup> A simple but interesting example is [190] where the author searched "`const double epsilon =`" (and equivalents in other languages) on all GitHub repositories to study the value programmers typically chose for epsilon.



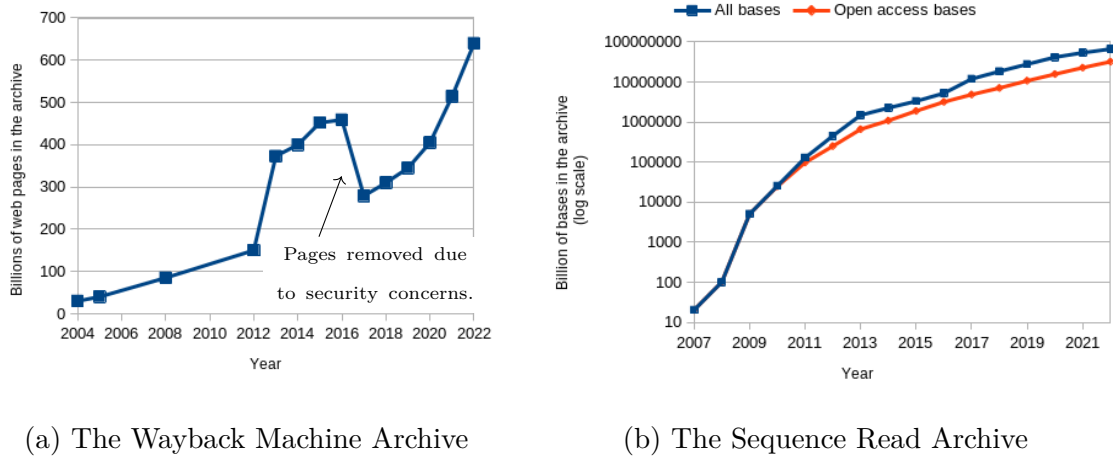


Figure 2: Plots of the database growth for the Wayback Machine [373] and the Sequence Read Archive [372]. The databases are big but also still quickly growing.

linearly scanning the whole string to search for a pattern. Alternatively, the suffix tree is a classical structure that enables more efficient sequence analysis but necessitates 10 bytes per base [227], which amounts to 30 gigabytes for a human genome containing 3.3 billion bases. This would not be practical for projects where thousands of genomes have been sequenced such as the 1000 Genomes project [200] completed in 2015 and the 100K Genomes project [370] completed in 2018. Fortunately, compact data structures that exploit redundancy to decrease space usage offer an interesting trade-off: for a human genome it allows representing the sequence and its suffix tree using just 4 gigabytes [227].

Scalable algorithms are crucial for Bioinformatics as there has been a drastic decrease in sequencing cost and an increase in throughput since 2008 which both lead to higher volumes of DNA being sequenced. So far, the European Nucleotide Archive has accumulated more than 50 petabytes [371] of sequencing data. While the NCBI Sequence Read Archive has more than 73 petabases [372] of data including 38 petabases in open access and is still growing, see Figure 2. However, like for the software heritage project and internet archive, in the ENA and the NCBI, the data is indexed solely based on its metadata. In Chapter 7, we propose a compact index specifically designed for sets of short reads, that takes advantage of the redundant input to achieve better compression and scalability.

## 2 Sketching

So far, we presented the two core challenges at the heart of modern text processing: enabling relevant (and sometimes complex) queries suited to specific applications while also maintaining performances that can scale to large volumes of data. This thesis offers new theoretical and practical trade-offs between complex and scalable queries, and we do so through the use of **sketches**. In this thesis, a sketch is a lossless or lossy compression that keeps only the essential characteristic of the input needed to answer a given query, offering promising scalability potential. Examples of sketches for lossy compression include Karp–Rabin fingerprints [35] (see Preliminaries 4.3) which occupy constant space

and allow checking whether two strings match with high probability, but in themselves do not contain enough information to reconstruct the original data. For lossless compression, an example is the Lempel–Ziv factorization [17], a very efficient compression in practice used in compression formats such as `png` or `zip`, that always allows reconstructing the original string, but in the worst case, the Lempel–Ziv factorization can occupy as much space as the original input. There exist a myriad of sketches and how to choose one entirely depends on the problem’s characteristic and space limitations, but they can be grouped under three general approaches (that can sometimes be combined):

- **Compressed input:** highly redundant data can sometimes be represented by a sketch of a manageable size. Consequently, algorithms that can directly operate on the sketch become much more efficient. This is a very natural approach as the data is almost always shared in a compressed format, the difficulty is working with the sketch’s property and structure. However, it is important to note that not all problems can be solved faster with this approach. For example, Abboud et al. [233] showed that to compute the longest common subsequence of two strings of uncompressed size  $N$ , given as grammar of size  $n$ , there is a time lower bound  $\Omega(nN)^{1-o(1)}$  assuming the Strong Exponential Time Hypothesis (SETH, see Preliminaries 4.1). Very recently, Ganesh et al. [354] also showed a time lower bound  $\Omega(N^{k-1}n)$  conditioned on SETH to compute the median edit distance and length of the longest common subsequence of  $k$  strings. In other words, sometimes, even if the input is given in compressed form, we cannot avoid a high dependency on the uncompressed size. In this thesis, Chapters 2 and 3 both take as input a sketch, more precisely a grammar-compressed text and their complexities are given as a function of the size of the compressed input.
- **Streaming algorithms:** there, the data is considered so large that it can only be handled as a stream. For the pattern-matching problem, the pattern and the length of the text are known and can be preprocessed in advance. Then the characters of the text arrive one by one and can only be accessed later if they are stored explicitly. This model focuses on small space complexity and accounts for all the space used, including the space required to store the input. Thus, sketches are crucial to keep the necessary information about the data already seen while limiting space usage. Chapter 1 studies the regular expression membership and pattern-matching problems in this model. Streaming algorithms also relate to the practical notion of efficient second-memory algorithms. One of the challenges when dealing with large inputs is the quantity of main memory used, as most computers are still limited to gigabytes of RAM. To circumvent this limitation, programs may resort to working directly from secondary memory as disk space scales at a much cheaper cost than RAM. Random access to disk is in general quite inefficient, however, contiguous reads on recent SSD can read gigabytes (between 2.2 and 3.4 Gb) of data per second which is comparable to the speed of most RAM. Therefore, an algorithm that uses no or few random accesses (including streaming algorithms) can be executed directly on disk which allows it to scale to large inputs much more easily. We use this approach of streaming on secondary memory to limit main memory usage in Chapter 7. The construction of the index is split into phases that read and process the input as a stream from disk.

- **Approximation algorithms:** When dealing with very large datasets, it is not always needed to provide precise answers to queries. Allowing for some approximation in the results can enable shortcuts and help bypass lower bounds. There, using sketches in the form of lossy compression inherently introduces approximation but allows for more efficient processing. The entire second part of this thesis is dedicated to this approach. In particular, in Chapter 5, we treat the problem of the Longest Common Subsequence with approximately  $k$  mismatches with a probabilistic algorithm that answers correctly with high probability.

Before presenting the main contributions of this thesis, I would like to present MinHash, a sketch that has been very successfully applied in Bioinformatics to the large amounts of genomic data we mentioned in Section 1.4. I will also mention a few other popular sketches in Bioinformatics for completeness. In this field, a sketch generally refers to lossy compression producing a small and approximate summary of the data, and does not include lossless compression as we do. Most of the sketching techniques in Bioinformatics rely on the  $k$ -mer decomposition: the sequence (or sequences) are represented by their substrings of length  $k$  (called  $k$ -mers)<sup>21</sup>. Naturally, similar strings will tend to share a lot  $k$ -mers. The Jaccard index measures this similarity. For two  $k$ -mer sets  $A$  and  $B$ , it is simply defined as  $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$ . Unfortunately, dealing with complete  $k$ -mer sets quickly becomes infeasible as the input's size grows. The *MinHash* sketch [54], originally developed for duplicate detection in web pages and images, allows computing efficiently an estimator of the Jaccard Index. For a given set of  $S$  hash functions  $(h_1, h_2, \dots, h_S)$ , the sketch of a  $k$ -mer set  $A$  is defined as:

$$\text{MinHash}(A) = (\min_{a \in A} h_1(a), \min_{a \in A} h_2(a), \dots, \min_{a \in A} h_S(a))$$

It is a locality-sensitive hash: similar inputs will have similar sketches with high probability. Therefore, for two  $k$ -mer sets the number of shared hashes in  $\text{MinHash}(A)$  and  $\text{MinHash}(B)$  is on average a good estimator of the shared elements. The first tool implementing this concept for  $k$ -mer set was Mash [230] and the speed at which it could compare two sets made it a game changer. However, it is important to note that it uses a variant of MinHash sometimes called KMV sketching or Bottom MinHash [78] that uses a single hash function and is defined for a  $k$ -mer set  $A$  as  $\text{mash}(A) = (h(a_1), h(a_2), \dots, h(a_S))$  where  $h(a_1) < h(a_2) < \dots < h(a_S)$  are the  $S$  the smallest elements of  $h(a)_{a \in A}$ . The estimator of the Jaccard index is then computed as follows:

$$J_{\text{mash}}(A, B) = \frac{|\text{mash}(A) \cap \text{mash}(B) \cap \text{mash}(A \cup B)|}{|\text{mash}(A \cup B)|}$$

and the authors show that it is an unbiased estimator. Since the publication of Mash, multiple variants have been considered, but not all the corresponding estimators are unbiased [346]. For visual illustrations of the MinHash variants, I recommend Camille Marchet's blog post [376]. MinHash and its variants are sketches for set similarity, but other sketches are used for other tasks: Bloom filters can be seen as sketches for set membership ("Is my element in the set?"), count-min sketches have been used for element

---

<sup>21</sup>In general, a  $k$ -mer and its reverse complement are seen as equivalent and represented by the smallest of the two according to the lexicographic order (called the canonical  $k$ -mer).

frequency (“In how many sets does my element occur?”), and HyperLogLog sketches estimate set cardinality (“How many distinct elements are in my set ?”). Those are examples taken from the survey on sketching for genomics by Will P.M. Rowe [301], and for more details see also the survey by Marçais et al. [298] on “Sketching and Sublinear Data Structures in Genomics”.

### 3 Contributions

In this manuscript, each chapter corresponds to an independent publication, with a specific problem, a review of the state of the art, and a set of contributions. The choice of not unifying notations and merging those publications was motivated by the variety of subjects and techniques used, and how they were conceived as independent projects (with varying sets of co-authors) during the PhD. Nevertheless, this section intends to provide an overview of the main contributions, some insights into the techniques used, and how they relate to the over-arching concept of sketches.

**Part I** focuses on a theoretical study of complex queries. We start with regular expression search in the streaming setting. We assume to be given a regular expression  $R$ , a streaming text  $T$  of length  $n$  over an alphabet  $\Sigma$ . For regular expression *membership* we need to determine, after having seen  $T$  entirely, whether it matches a regular expression  $R$ , while for *pattern matching*, we must answer, at each position  $r$ , whether there exists a substring  $T[l..r]$  recognized by  $R$ . In **Chapter 1**, our main contribution is to identify  $d$ , the number of union symbols and Kleene stars in  $R$ , as the key parameter that enables a space-efficient streaming algorithm. Previously, Bille and Thorup [140]<sup>22</sup> already used this parameter to propose algorithms to solve membership and pattern matching of a regular expression of length  $m$  in  $\mathcal{O}(m)$  space and  $\mathcal{O}(n(\frac{d \log w}{w} + \log d))$  time, where  $w$  is the size of the machine word. But it remained unclear if  $d$  could be used for a space-efficient solution. This parameter was already known in the streaming model for the two special cases of regular expressions: dictionary matching and don’t care matching. In dictionary matching, we search for occurrences of a set of patterns  $\{P_1, P_2, \dots, P_d\}$  of total length at most  $m$ , which corresponds to pattern matching for the regular expression  $(P_1|P_2|\dots|P_d)$ , a series of results [137, 176, 199, 243, 270] led to the development of a randomized Monte-Carlo algorithm in  $\mathcal{O}(d \log m)$  space and  $\mathcal{O}(\log \log |\Sigma|)$  time per character. For don’t care pattern matching, we are given a pattern  $P_1?P_2\dots?P_d$  where  $P_i$ ,  $i \in [1, d]$ , are strings (possibly empty) over an alphabet  $\Sigma$  of total length at most  $m$ . Don’t care pattern matching can be expressed as matching  $R = P_1(1|2|\dots|\sigma)P_2(1|2|\dots|\sigma)\dots(1|2|\dots|\sigma)P_d$ , and Golan, Kopelowitz, and Porat [292] showed that this problem can be solved by a randomized Monte-Carlo algorithm in  $\mathcal{O}(d \log m)$  space and  $\mathcal{O}(d + \log m)$  time per character.

Formally, we provide space-efficient randomized Monte-Carlo algorithms (meaning the execution time is deterministic, but the algorithms can err with a small probability) that solve regular expression membership and pattern matching in  $\mathcal{O}(d^3 \text{polylog } n)$  space and  $\mathcal{O}(nd^5 \text{polylog } n)$  time per character of  $T$  (Theorem 1.27). Here is a bird’s eye view of how we prove our result: we start by defining *atomic strings* which are the “words” appearing in the regular expression. They only contain characters of  $\Sigma$  and there are  $\Theta(d)$  of them. For example, for  $R = \text{GAT}(\text{TA}|\text{O})(\text{CAT})^*$  the set of atomic strings is  $\{\text{GAT}, \text{TA}, \text{O}, \text{CAT}\}$ .

<sup>22</sup>They actually consider  $k$  the number of strings appearing in  $R$  but  $k = \Theta(d)$ .

The basis of our approach is to efficiently store some specific occurrences of prefixes of the atomic strings in the text  $T$ . Those stored occurrences are then linked to test if there is a “partial” match of  $R$  (Definition 1.5). Over periodic regions of the text, there can be too many occurrences to store them all. Thus, we choose to store only a few of those occurrences that can be very far apart, with just a long periodic substring in between. To reconstruct a partial match, we must check if that long periodic substring corresponds to a run of the Thompson automaton [9]. We formulate it as finding a path of a specific weight in a multi-graph. We then efficiently solve this graph problem by translating it into a circuit using addition and convolution gates that can be evaluated in a space-efficient manner using a general framework [145, 237]. Additionally, we improve that framework by removing its dependency on the Extended Riemann Hypothesis (Theorem 1.33). Here the sketches are Karp–Rabin fingerprints (Preliminaries 4.3) used to detect matches of the prefixes of atomic strings, but it is black-boxed in the streaming pattern matching algorithm (Theorem 1.9) that we use. My personal contribution to this work was focused on formalizing and proving key properties of “anchor” positions (Definition 1.14) which serve to efficiently choose which occurrences are stored and this concept is key for achieving the desired space complexity. I also participated in developing the streaming algorithm.

In **Chapter 2**, we begin our study of gapped consecutive matching. In this problem, we are given patterns  $P_1, P_2$ , an interval  $[a, b]$ , and must report all pairs of positions  $(i, j)$  in a text  $T$  such that an occurrence of  $P_1$  starts at position  $i$ , an occurrence of  $P_2$  starts at position  $j$ , there are no occurrences of  $P_1$  or  $P_2$  starting in the interval  $[i + 1, j - 1]$ , and finally  $j - i \in [a, b]$ . Bille et al. [349] introduced those queries and gave a conditional lower bound stating that for indexes of size  $\tilde{O}(|T|)$  (the  $\tilde{O}$  notation hides polylogarithmic factors) achieving a query time faster than  $\tilde{O}(|P_1| + |P_2| + \sqrt{|T|})$  would contradict the Set Disjointness conjecture, even if  $a = 0$  is fixed. Additionally, they provided a non-trivial index that uses  $\tilde{O}(|T|)$  space and  $\tilde{O}(|P_1| + |P_2| + |T|^{2/3} \text{output}^{1/3})$  time. We assume  $a = 0$  is fixed, and that the text  $T$  of size  $n$  is given as a straight-line program  $G$  of size  $g$ , which is a context-free grammar generating exactly one string. For example the grammar with the non-terminals  $\{A, B, C, D\}$  and rules  $\{A \rightarrow BC, B \rightarrow \mathbf{b}, C \rightarrow DD, D \rightarrow \mathbf{d}\}$  generates the string  $\mathbf{bdd}$ . We choose this formalism as it captures the popular Lempel–Ziv factorization up to a logarithmic factor: a Lempel–Ziv factorization of size  $z$  can be transformed into a straight line program (SLP) of size  $\mathcal{O}(z \log n)$  [80, 91]. Our contribution is to create an index taking space polynomial in the size of the grammar that reports consecutive occurrences with distances in  $[0, b]$  in optimal time up to poly log factors. To report consecutive occurrences without constraints on the distance between  $P_1$  and  $P_2$ , our index uses  $\mathcal{O}(g^2 \log^4 |T|)$  space, where  $g$  is the size of the SLP (see Corollary 2.16). We rely on an efficient construction of compact tries (see Preliminaries 4.2) which takes advantage of the strings being prefixes and suffixes of strings generated by non-terminals. This implementation uses Karp–Rabin fingerprints (see Preliminaries 4.3) and the compacted tries are then augmented using a heavy path decomposition. We then reuse this structure for our main result: Theorem 2.1, with an index that can report consecutive occurrences with distances in an interval  $[0, b]$  using  $\mathcal{O}(g^5 \log^5(|T|))$  space. Apart from the Karp–Rabin fingerprints in the compact tries, the straight-line program input is the main sketch used in this work, but the index we construct also forms a grammar-based sketch specific to consecutive occurrences. The index of Theorem 2.1 circumvents the lower bound for highly compressible texts (such that  $g^5 \ll n^2$ ), which is a non-trivial result since some problems

cannot avoid a high dependency on the size of the uncompressed string (as detailed in Section 2). However, we expect our space complexity to be far from optimal and leave improvements as well as the general case with  $0 \leq a \leq b \leq |T|$  as open questions<sup>23</sup>. I contributed to every part.

Partially motivated by the limitations of the space usage in  $\tilde{\mathcal{O}}(g^5)$ , in **Chapter 3**, we address the dual problem: consecutive pattern matching, where the patterns and the text are processed simultaneously. Note that for an uncompressed text, consecutive pattern matching can be solved by a classic online matching algorithm, just keeping track of the most recent occurrences of  $P_1$  and  $P_2$  in  $\mathcal{O}(|T| + |P_1| + |P_2| + \text{occ})$  time. We show that a similar complexity can be achieved when the text is highly compressible: all consecutive occurrences can be reported in  $\mathcal{O}(g + |P_1| + |P_2| + \text{occ})$  time (see Theorem 3.1) where  $g$  is the size of the grammar compressed text. We then derive from this result algorithms for gapped consecutive matching (Corollary 3.2) and the  $k$ -closest consecutive occurrences (Corollary 3.3). Our result is based on the efficient “boundary information” encoding recently introduced by Ganardi and Gawrychowski [353]. For a given pattern  $P$ , the  $P$ -boundary information of a string  $S$  stores substrings occurring both in  $P$  and  $S$ . They are chosen to capture just the information needed to detect new occurrences of  $P$  that could occur when concatenating a string to  $S$ . The authors show how to use this encoding to determine in  $\mathcal{O}(g + |P|)$  time whether  $P$  occurs in the compressed text. We slightly extend their approach to report all occurrences crossing the boundary (Lemma 3.10). Then we repeat this technique on a second level with “secondary boundary information” and carefully analyse all cases to obtain Theorem 3.1. Here, again, the main sketch is the straight-line program we work on, and I contributed to all aspects of the publication.

Intrinsically, all previous chapters rely heavily on periodicity detection, thus it felt only natural to study this problem in **Chapter 4**. We show how to report all runs in optimal time in the most abstract model where they can be defined: general (unordered) alphabets where the only operation allowed is an equality test between two characters. We first consider the problem of square detection, and then extend our approach to square and run reporting. In 1984, Main and Lorentz [27] designed an  $\mathcal{O}(n \log n)$  time algorithm for square detection in a text  $T$  of size  $n$  over a general unordered alphabet. They also provided a matching lower bound for strings that have  $\Omega(n)$  distinct symbols but left as an open question whether a faster algorithm was possible if the size of the alphabet  $\sigma = |\Sigma|$  is restricted. We start by proving that the problem requires  $\Omega(n \log \sigma)$  comparisons even if the size of the alphabet is known (Theorem 4.1). Additionally, in Theorem 4.9, we show that computing any relevant approximation of the number of distinct characters requires  $\Omega(n\sigma)$  operations. For general ordered alphabets (where an order is given), Crochemore [30] used the  $f$ -factorization (related to popular Lempel–Ziv factorization) to give an algorithm for square detection running in  $\mathcal{O}(n \log \sigma)$  time. Roughly speaking, the  $f$ -factorization and Lempel–Ziv factorization (LZ-factorization) detect repetitive fragments in the text and can be computed efficiently using a suffix tree or suffix array. However, we show that on general unordered alphabets, those factorizations require  $\Omega(n\sigma)$  operations to be computed (as a corollary of the lower bound on approximating the alphabet). Instead, we introduce the  $\Delta$ -approximate LZ-factorization which acts as a

---

<sup>23</sup>We started writing a solution for the general case where the distance between the consecutive occurrences has to be in  $[a, b]$ , but it was very technical and had an unreasonable space-complexity of  $\tilde{\mathcal{O}}(g^{15})$ ...

sketch capturing only sufficiently long squares (of length at least  $8\Delta$ ), as opposed to the  $f$ -factorization and the LZ-factorization that capture all squares. We present our final algorithm in steps. We first assume that the alphabet size is known and focus on giving an upper bound on the number of comparisons, then we proceed to remove the assumption on knowing the alphabet size, and finally, we provide an overall efficient algorithm running in time  $\mathcal{O}(n \log \sigma)$ . In this work, I participated in formalizing and clarifying the proofs throughout the research process.

**Part II** explores the use of approximation to further reduce the size of sketches and provide more efficient algorithms. Each chapter provides a proof of concept implementation for Bioinformatics applications. Earlier in this introduction, we detailed the importance of similarity measures for numerous applications. In Bioinformatics, the edit distance is arguably the most popular similarity measure, however, Backurs and Indyk [254] proved a conditional lower bound (based on SETH) which suggests it is unlikely to be computable in strongly subquadratic time. The need to overcome this quadratic-time barrier led to the study of approximate algorithms for the edit distance. Chakraborty et al. [255] gave the first breakthrough result with a constant-factor approximation algorithm that computes the edit distance between two strings of length  $n$  in time  $\tilde{\mathcal{O}}(n^{2-2/7})$ . Since our publication [317] (presented in Chapter 5), a series of works have been published on approximating the edit distance [311, 318] with the strongest result being [306] with a constant factor approximation in time  $n^{1+\varepsilon}$  for any  $\varepsilon > 0$  (where the approximation constant depends solely on  $\varepsilon$ ). Nevertheless, these algorithms tend to be quite technical and even those meant to be simpler such as [305] do not seem to have been implemented and evaluated in practice. In **Chapter 5**, we take another approach by considering a different similarity measure meant to be both robust to small changes and simple enough to allow for efficient computation. We consider the Longest Common Substring (abbreviated LCS hereafter) with Approximately  $k$  mismatches which is an approximate version of **LCS with  $k$  Mismatches**. Recall that, for an integer  $k$  and two strings  $X$  and  $Y$ ,  $\text{LCS}_k(X, Y)$  is the maximal length of a substring of  $X$  that occurs in  $Y$  with at most  $k$  mismatches. For a constant  $\varepsilon > 0$ , the **LCS with Approximately  $k$  Mismatches** problem needs to return a substring of  $X$  of length at least  $\text{LCS}_k(X, Y)$  that occurs in  $Y$  with at most  $(1 + \varepsilon) \cdot k$  mismatches. This problem has been introduced by Kociumaka, Radoszewski, and Starikovskaya [294] after they showed that there is  $k = \Theta(\log n)$  such that **LCS with  $k$  Mismatches** cannot be solved exactly in strongly subquadratic time (conditioned on SETH). In Theorem 5.1, we provide two algorithms: one assuming a constant size alphabet running in  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  time and space, and one in  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$  time and linear space without constraints on the alphabet. The first result relies on an Approximate Nearest Neighbour data structure [194] as a black box, and we do not evaluate it in practice. In contrast, our second contribution is simpler, and we confirm its practicality through an experimental evaluation. Additionally, in Fact 5.2, we show a conditional lower bound for **LCS with Approximately  $k$  Mismatches** (with a construction similar to the proof for the lower bound of **LCS with  $k$  Mismatches**). In our algorithms, we rely on Karp–Rabin fingerprints and a sketch estimating the Hamming distance based on dimension reduction, both detailed in Section 2. I participated in all aspects of the work, modified our lower bound reduction to work over a binary alphabet and implemented the practical evaluation of our algorithm.

Continuing our exploration of similarity measures and distances, the next chapter fo-

cuses on the Dynamic Time Warping (DTW) distance. Recall that for the DTW distance, one must “warp” the two strings: double some characters until the strings are of equal length and then sum the distances between the characters at the same positions. To propose an efficient algorithm for this distance, in **Chapter 6**, we consider one of the simplest forms of sketching: run-length encoding. The run-length encoding of a string  $S$  of length  $N$  is a concise representation  $\text{RLE}(S) = (c_1, l_1)(c_2, l_2) \dots (c_n, l_n)$  where  $(c_i, l_i)$  represents the character  $c_i \in \Sigma$  repeated  $l_i$  times for  $i \in [1, n]$ , and such that  $\sum_{i \in [1, n]} l_i = N$ . This sketch is especially relevant for DTW as the runs of equal characters tend to be aligned despite variations in length. That is why the number of runs in the strings has already been used by Froese et al. [285] who gave an algorithm computing the DTW distance between two strings with running time  $\mathcal{O}(mN + nM)$ , where  $M, N$  are the lengths of the strings, and  $m, n$  are the sizes of their run-length encodings.

Our contribution is as follows: when the distances between characters of  $\Sigma$  are integers, for a pattern  $P$  with  $m$  runs and a text  $T$  with  $n$  runs, we show that there is an  $\mathcal{O}(n + m)$ -time algorithm that computes all locations  $j$  where the DTW distance between  $P$  and a suffix of  $T[..j]$  is at most 1. More generally for an integer  $k$ , we provide an algorithm with  $\mathcal{O}(knm)$ -time that computes all locations  $j$  where the DTW distance between  $P$  and a suffix of  $T[..j]$  is at most  $k$ . Our interest and research on DTW are also motivated by potential applications to analysing biological data produced by the third generation sequencing where DNA is passed through a nanopore at an uneven speed which tends to create errors in the length of stretches of the same nucleotide (homopolymers) [331]. We detail this potential application in Section 6. I contributed to every part of that work except for the approximation algorithms (given as corollary of our second algorithm) which were added by Tatiana Starikovskaya, I only helped in checking correctness. Since our publication, Boneh, Golan, Mozes, and Weimann uploaded a preprint [361] with a  $\tilde{\mathcal{O}}(n^2)$ -time computation of the DTW distance between two strings with  $n$  runs, which is optimal up to log factors. They follow the approach that Clifford et al. [282] used to show a similar result for the edit distance: they represent and manipulate the distances computed with a piecewise-linear function.

For **Chapter 7**, we shift our focus from the study of similarity measures to the practical problem of readsets indexing. Additionally, in this chapter, the perspective on approximation is different, instead of studying approximate matching where one reports all substrings at a given distance from the pattern, we propose a compact index that has the downside of reporting false positives: occurrences of the pattern that are not fully included in a read.

Recall that a read is a sequence of base pairs, generally short (the precise length depends on the sequencing technique), obtained when sequencing DNA. To be able to re-assemble the entire DNA sequence, each position of the sequence is generally covered by multiple reads. The average number of reads covering a position is called the sequencing coverage. The standard sequencing coverage for short reads to assemble a genome is now between 30 and 50, making the readsets highly repetitive, especially when the DNA sequenced comes from a single individual. To index a single repetitive string, the FM-index [99] based on the Burrows-Wheeler transform (BWT) [48] is one of the most important data structures<sup>24</sup>, and it has been applied in several Bioinformatics tools for

<sup>24</sup>The authors of both the BWT and FM-index received in 2022 the *ACM Paris Kanellakis Theory and Practice Award* for those breakthroughs.



short read alignment [134, 164, 135]. The most recent improvement on the FM-index is the  $r$ -index by Gagie et al. [313] that takes space proportional to  $r$ , the number of runs in the BWT of the indexed string. More precisely, for an indexed string of length  $n$ , the data structure occupies  $\mathcal{O}(r \log \log n)$  space while still being able to count and locate all occurrences of a pattern in optimal time (up to a logarithmic factor). This makes the number of runs in the BWT an important parameter for space efficiency and is often seen as a measure of the repetitiveness of the text.

Unfortunately, extending the BWT to a collection of strings is not straightforward. To better understand the context of Chapter 7, let me give an overview of the current solutions and their impact on the resulting index structures. A possible approach is to concatenate all strings of the collection with a distinct separator character between them and then build the FM-index of the resulting string. However, with this approach the string can be very large, making the BWT challenging to build and update (adding a string to the index), and one character is added to the alphabet per string in the collection which becomes costly for large collections. Some methods avoid this increase in the alphabet size by having a repeated separator symbol and comparing two separators using their position in the concatenated string (this is done in tools such as **BEETL** [159] and **RopeBWT** [185]). **BigBWT** [280] uses another method, there are two separator symbols: one is put between the strings and the other just at the end of the concatenated string as a way to order the separators. But both of those approaches create a result where two different letters followed by the same suffix (up to a dollar) will be ordered depending on the input order of their respective strings in the collection. This can cause some variation in the number of runs in the resulting transform. A common heuristic (used by **BEETL** [159] and **RopeBWT** [185]) is to sort the strings in the collection according to their colexicographic<sup>25</sup> order so that the strings with similar suffixes are grouped together. Mantaci et al. [117] generalized the concept of the BWT to collections of strings. The EBWT is a permutation of characters in the strings of a collection according to the lexicographic order of the suffixes that immediately follow those characters, considering each string as cyclic<sup>26</sup>. In the literature, several variants of the EBWT are commonly used. The first variant is the EBWT of the collection where each string is appended a shared end-of-string symbol (This is the definition we take as standard in Chapter 7). The second variant is the EBWT of the collection where each string is appended a unique end-of-string symbol.

Those nuances between definitions of the BWT for a concatenation of strings and EBWT with or without distinct end-of-string symbols may seem minor, but they do impact the transformed string, and Cenzato and Lipták [351] showed that the final number of runs can drastically change depending on the precise definition of the BWT for a given collection of strings (among the variant we described above). They compared the number of runs produced by each variant to the number of runs in the BWT where the end-of-string symbols follow the order given by Bentley et al. [279], which minimize the numbers of runs in the resulting BWT. The construction of this order has been implemented in practice by [362]. Their experiments demonstrate the efficiency of the heuristic ordering the end-of-string symbols according to the colexicographic order of the strings in practice (which we also highlighted in Chapter 7) and they provide a theoretical upper bound. I

---

<sup>25</sup>Lexicographic order on the reversed strings, i.e.  $X \prec_{\text{colex}} Y$  if and only if  $\text{rev}(X) \prec_{\text{lex}} \text{rev}(Y)$ .

<sup>26</sup>This is the Omega order ( $\prec_{\omega}$ ) order defined by Mantaci et al. [117].

strongly encourage reading their survey, it is a very helpful reminder that the devil is in the details.

Our work in Chapter 7 is prior to the publication of the survey of Cenzato and Lip-ták [351] and the construction of an optimal order by Bentley et al. [279], and we take a different approach to optimizing the number of runs for the BWT. We propose to take advantage of information very commonly associated with reads: alignment to a reference genome. We build a tree where the main trunk is the reference and reads branch out of the trunk at the position they are aligned to, and then compute the generalization of the BWT for trees: the eXtended Burrows–Wheeler transform (XBWT) [130]. Intuitively, this provides a context to the reads that allows for better sorting and limits the number of runs breaking in the transformed string. Formally, we show that if the reads are almost identical to the reference and that the differences are located towards the end of the reads (which is the case in practice for short reads), the number of runs in the XBWT can be bounded depending on the number of runs in the BWT of the reference. We then test our approach in practice and show that it does obtain fewer runs than the BWT with the colexicographic heuristic (15% less for a human chromosome 19 read set). The main drawback of our index is that we search in the branching tree thus when counting the number of occurrences of a pattern, we cannot avoid counting occurrences that are not fully contained in a read (that are partially or entirely in the main trunk corresponding to the reference).

Another contribution of our work is that, with the goal of a scalable construction of the index in mind, we adapt the technique of a prefix-free parsing construction of the BWT of Boucher et al. [280] to the construction of the XBWT. In the prefix-free parsing construction of the BWT, the input string is parsed into overlapping phrases as follows: we maintain the hash of a sliding window using Karp–Rabin fingerprints, and whenever the hash is equal to zero we end the current phrase and begin a new one. This technique, called context-triggered piecewise hashing, allows us to create a set of phrases that are prefix-free. The string is then decomposed using a *dictionary* and a *parse*. The dictionary associates each phrase with a metacharacter, and the parse is a string of metacharacters in the same order as the corresponding phrases occur in the input string. This construction uses sketches (Karp–Rabin fingerprints) but also creates a sketch through this compact representation of the string. The intuition is that repeated sections of the string are translated into repeated phrases that can be represented by the same metacharacters in the parse, allowing the dictionary to remain reasonably small. The BWT of the input string is then constructed starting from the BWT of the parse (where the order between the metacharacters is the lexicographic order of the phrase they represent) and the BWT of the phrases. Adapting this technique to labelled trees and the construction of the XBWT is non-trivial because each branch may create new phrases. My contribution was focused on this adaptation, the implementation, and the experimental evaluation of our proposed data structure in terms of the number of runs in the XBWT. The implementation of the queries and analysis of false positives are left as future work.

I hope this overview of the contributions has highlighted clearly that sketching is a valuable tool in a variety of settings and applications. I expect that they will continue to appear in many future works both in theory and in practice.

## 4 Preliminaries

This section aims at presenting some recurring concepts needed to understand the following chapters.

### 4.1 The Strong Exponential Time Hypothesis

The Strong Exponential Time Hypothesis (SETH) has already been mentioned several times in this introduction as the basis of many important conditional lower bounds. Conditional lower bounds are a crucial tool to understand the problems' complexity and whether the current upper bounds are optimal. In this thesis we never work directly with SETH, but for completeness here is the hypothesis's statement:

THE STRONG EXPONENTIAL TIME HYPOTHESIS [71]

The satisfiability of a formula in conjunctive normal form (a conjunction of clauses, where each clause is a disjunction of literals) with  $N$  variables and  $\mathcal{O}(N)$  clauses cannot be solved in time  $2^{N(1-o(1))}$ .

### 4.2 Tries and Suffix Trees

Let  $\mathcal{S} = S_1, S_2, \dots, S_k$  be a collection of strings over an alphabet  $\Sigma$ . Its trie [2, 4, 5] is a rooted tree with edge labels in  $\Sigma$ . For any path, we say the path *spells* the string obtained by concatenating the label of the edges of the path. Likewise, when referring to a node within the tree, we define its label as the string spelled by the unique path from the root to that node. The trie is a tree such that no two nodes spell the same string, each leaf spells a string of  $\mathcal{S}$  and each  $T_i \in \mathcal{S}$  is spelled by a leaf or an internal node. In both cases, we mark the nodes spelling the  $T_i \in \mathcal{S}$  with a special end-of-string character.

The more efficient compacted trie is obtained by contracting into a single edge each maximal path from a node that is either marked (by an end-of-string character) or branching (with at least two children) to its lowest ancestor that is either branching or marked. This edge is then labelled with the string spelled by the path it replaced. Let  $S$  be a string spelled by a node of the non-compacted trie. If it is also spelled by a node of the compacted trie we say the node is *explicit* else it is *implicit*. Those definitions are illustrated in Figure 3. Note that for any two nodes  $i$  and  $j$ , their lowest common ancestor spells the longest common prefix between the strings spelled by  $i$  and  $j$ . Notice that all nodes are either branching or marked and that all marked nodes spell distinct strings of  $\mathcal{S}$ . Hence, there are  $\mathcal{O}(k)$  nodes in the compacted trie. For each edge, there exists  $(i, s, e)$  such that its label is equal to  $S_i[s..e]$ , thus we can avoid storing the label explicitly and just store the reference  $(i, s, e)$  to the strings of  $\mathcal{S}$ . Thus,  $\mathcal{O}(k)$  space is sufficient to store the compacted trie.

A suffix tree of a string  $T$  of length  $n$  is the compacted trie containing the suffixes of  $T\$$  (where  $\$$  is a special symbol not in  $\Sigma$ ) i.e. over the collection  $\{T[j..]\$ \text{ for } j \in [0, n-1]\}$ . It can be stored in  $\mathcal{O}(n)$  space and for a linearly-sortable alphabet can be constructed in  $\mathcal{O}(n)$  time [56]. For more details on the suffix tree see Chapters 5 to 9 of [57]. We illustrate this definition in Figure 3c. Tries and suffix trees are used in Chapters 1 to 4.



### 4.3 Karp–Rabin Fingerprint

For  $p \geq |\Sigma|$  a prime number and an integer  $b \geq |\Sigma|$ , the Karp–Rabin fingerprint [35] of a string  $S$  is defined as  $\varphi_p(S) = \sum_{i=0}^{|S|-1} S[i]b^{|S|-i-1} \bmod p$ . It is easy to see that if two strings  $S_1$  and  $S_2$  are equal then  $\varphi_p(S_1) = \varphi_p(S_2)$ . The reverse is not true, two distinct strings  $S_1$  and  $S_2$  can have equal fingerprints, but we can bound the probability of such an event by choosing the prime  $p$  at random below an integer  $I$ . If  $I = |S_1|^2|S_2|$  the probability of a false match is  $\mathcal{O}(\frac{1}{|S_1|})$ . For a detailed analysis see Chapter 4.4 of [57]. This fingerprint has a convenient “sliding” property in the sense that for a string  $T$  and two integers  $i, m$  such that  $i + m < |T|$ , we can express efficiently the fingerprint of  $T[i + 1..i + m]$  as a function of the fingerprint of  $T[i..i + m - 1]$ ,  $T[i]$  and  $T[i + m]$ :

$$\varphi_p(T[i + 1..i + m]) = (\varphi_p(T[i..i + m - 1]) \times b - T[i]b^m + T[i + m]) \bmod p$$

This allows Karp–Rabin fingerprints to be used for an online matching algorithm for a pattern  $P$  running in  $\mathcal{O}(|P| + |T|)$  that can err with a small probability (Monte-Carlo). In practice,  $p$  is often chosen as a large prime number around  $2^{27}$  to still fit into a 32-bit integer, and in the implementation of Chapter 7,  $b = 256$  to accommodate for the UTF-8 where each character is encoded on 8 bits.

In Chapters 2 and 5, we use a variant of Karp–Rabin fingerprint defined by Porat and Porat [137]: for a prime number  $p \geq |\Sigma|$  and  $r \in \mathbb{F}_p$  (the finite field of integers modulo  $p$ ), the fingerprint of a string  $S$  is defined as  $\varphi_{p,r}(S) = \sum_{i=0}^{|S|-1} S[i]r^i \bmod p$ . The collision analysis of this variant is simpler: for a given integer  $n$ , if  $p$  is chosen to be  $\Theta(\max(|\Sigma|, n^k))$  and  $r$  is chosen at random in  $\mathbb{F}_p$ ,  $|S_1|, |S_2| \leq n$ , then  $\varphi_{p,r}(S_1) - \varphi_{p,r}(S_2) = 0$  can be seen as a polynomial over  $\mathbb{F}_p$ . This polynomial has degree greater than zero (because  $S_1 \neq S_2$ ), and at most  $\max(|S_1|, |S_2|) < n$  thus it can have at most  $n$  roots. Thus, the probability that the fingerprints  $S_1$  and  $S_2$  collide is less than  $\frac{n}{n^k} = \frac{1}{n^{k-1}}$ .

Note that the precise definition can vary slightly depending on the specific property needed. In Chapter 2, the fingerprint of a string  $S$  is defined as a triple  $(r^{|S|-1} \bmod p, r^{-|S|+1} \bmod p, \varphi_{p,r}(S))$ , to have the property that given three strings  $X, Y, Z$  such that  $XY = Z$ , and the fingerprints of two of the strings, we can deduce in constant time the third fingerprint (Fact 2.13).

### 4.4 The Fine–Wilf’s Periodicity Lemma

Recall that a string  $S$  has period  $p$  if for all  $i \in [0, N - p - 1]$ ,  $S[i] = S[i + p]$ . There are two versions of the lemma, we give the proof only for the simpler version:

**Lemma 1** (Fine–Wilf’s Strong Periodicity Lemma [7]). *If  $p$  and  $q$  are both periods of a string  $S$  such that  $p + q \leq |S| + \gcd(p, q)$ , then  $\gcd(p, q)$  is also a period of  $S$ .*

**Lemma 2** (Fine–Wilf’s Weak Periodicity Lemma [7]). *If  $p$  and  $q$  are both periods of a string  $S$  such that  $p + q \leq |S|$ , then  $\gcd(p, q)$  is also a period of  $S$ .*

*Proof.* The proof is immediate when  $p = q$ . Without loss of generality, we can assume  $p > q$  and our goal is to show that  $p - q$  is also a period of  $S$ . Let  $i \in [0, N - (p - q) - 1]$ , if  $i + p < |S|$ , then  $S[i] = S[i + p] = S[i + p - q]$ , else it implies that  $i - q \geq 0$  thus  $S[i] = S[i - q] = S[i + p - q]$ . We showed that  $p - q$  is also a period of  $S$  and by a recurrence following Euclid’s algorithm, we obtain that  $\gcd(p, q)$  is also a period of  $S$ .  $\square$

This is a basic tool in the field of combinatorics on words, and it has been extended to strings with don't cares [62, 120, 79, 97, 76, 180, 357], Abelian periods [103, 168], parametrized periods [118], order-preserving periods [224, 316], approximate periods [138, 154, 193]. In this thesis, in Chapters 1 to 3 of Part I, we use the following corollary:

**Corollary 3.** *If there are at least three occurrences of a string  $Y$  in a string  $X$ , where  $|X| \leq 2|Y|$ , then the occurrences of  $Y$  in  $X$  form an arithmetic progression equal to the period of  $Y$ .*

*Proof.* Let  $i < j < k$  be the starting positions of occurrences of  $Y$  in  $X$ . Because  $|X| \leq 2|Y|$ , at least two of the substrings  $X[i..i + |Y|)$ ,  $X[j..j + |Y|)$ , and  $X[k..k + |Y|)$ , overlap by more than  $|Y|/2$ . Without loss of generality we can assume that it is the case for  $X[i..i + |Y|)$  and  $X[j..j + |Y|)$ , meaning  $X[j..i + |Y|)$  is a substring that is both a prefix and a suffix of  $Y$ , this implies a period of  $j - i < |Y|/2$ . Let  $p$  be the period of  $Y$ ,  $j - i$  has to be a multiple of  $p$ , else the (weak) periodicity lemma ( $p + j - i \leq |Y|$ ) would contradict the minimality of  $p$ . Next, because  $|X| \leq 2|Y|$ ,  $X[j..j + |Y|)$  and  $X[k..k + |Y|)$  must overlap by at least  $p$  positions, i.e.  $j + |Y| - 1 - k \geq p$ .  $X[k..j + |Y|)$ , is both a prefix and a suffix of  $|Y|$  which implies a period of  $k - j \leq |Y| - p$ , by the same argument of the minimality of  $p$  we have that  $k - j$  must be a multiple of  $p$  and thus the substring  $X[i..k + |Y|)$  is a run of period  $p$  and each position  $i + np$  for  $0 \leq n \leq \lfloor \frac{k-1+|Y|-i}{p} \rfloor$  is the start of an occurrence of  $Y$ .  $\square$



# Part I

## Complex Queries

---





## Chapter 1

# Streaming Regular Expression Membership and Pattern Matching

---

### Publication

This chapter corresponds to the following publication: Bartłomiej Dudek, Paweł Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya, “Streaming Regular Expression Membership and Pattern Matching”, in: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022), Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, ed. by Joseph (Seffi) Naor and Niv Buchbinder, SIAM, 2022, pp. 670–694, DOI: 10.1137/1.9781611977073.30.

Regular expression search is a key primitive in myriads of applications, from web scrapping to bioinformatics. A regular expression is a formalism for compactly describing a set of strings, built recursively from single characters using three operators: concatenation, union, and Kleene star. Two basic algorithmic problems concerning such expressions are membership and pattern matching. In the regular expression membership problem, we are given a regular expression  $R$  and a string  $T$  of length  $n$ , and must decide whether  $T$  matches  $R$ . In the regular expression pattern matching problem, the task to find the substrings of  $T$  that match  $R$ .

By now we have a good understanding of the complexity of regular expression membership and pattern matching in the classical setting. However, only some special cases have been considered in the practically relevant streaming setting: dictionary matching and wildcard pattern matching. In the dictionary matching problem, we are given a dictionary of  $d$  strings of length at most  $m$  and a string  $T$ , and must find substrings of  $T$  that match one of the dictionary strings. In the wildcard pattern matching problem, we are given a string  $P$  of length  $m$  that contains  $d$  wildcards, where a wildcard is a special symbol that matches any character of the alphabet, and a string  $T$ , and must find all substrings of  $T$  that match  $P$ . Both problems can be solved in the streaming model by a randomised Monte Carlo algorithm that uses  $\mathcal{O}(d \log m)$  space [Golan and Porat (ESA 2017), Golan, Kopelowitz and Porat (Algorithmica 2019)].

In the general case, we cannot hope for a streaming algorithm with space complexity smaller than the length of  $R$  for either variant of regular expression search. The main contribution of this paper is that we identify the number of unions and Kleene stars, denoted by  $d$ , as the parameter that allows for an efficient streaming algorithm. This parameter has been previously considered in the classical setting, and it has been observed that in practice it is significantly smaller than the length of  $R$ . We design general randomised Monte Carlo algorithms for both problems that use  $\mathcal{O}(d^3 \text{polylog } n)$  space in the streaming setting.

A crucial technical ingredient of our algorithms is an adaptation of the general framework for evaluating a circuit with addition and convolution gates in a space-efficient manner [Lokshtanov and Nederlof (STOC 2010), Bringmann (SODA 2017)], initially designed as a key component of a pseudopolynomial time algorithm for the subset sum problem. We show how to replace the Extended Generalised Riemann Hypothesis in [Bringmann (SODA 2017)] by an application of the Bombieri–Vinogradov theorem to achieve the same bounds (but unconditionally), which might be of independent interest.

# 1 Introduction

The fundamental notion of regular expressions was introduced back in 1951 by Kleene [3]. Regular expression search is one of the key primitives in diverse areas of large scale data analysis: computer networks [107], databases and data mining [82, 72, 73], human-computer interaction [162], internet traffic analysis [114, 110], protein search [75], and many others. As such, this primitive is often the main computational bottleneck in these areas and in the pursuit for efficiency has been implemented in many programming languages: Perl, Python, JavaScript, Ruby, AWK, Tcl and Google RE2, to name a few.

A regular expression  $R$  is a sequence containing characters of a specified alphabet  $\Sigma$  and three special symbols (operators): concatenation ( $\cdot$ ), union ( $|$ ), and Kleene star ( $*$ ), and it describes a set of strings  $L(R)$  on  $\Sigma$ . For example, a regular expression  $R = (a|b)^*c$  specifies a set of strings  $L(R)$  on the alphabet  $\Sigma = \{a, b, c\}$  such that their last character equals  $c$ , and all other characters are equal to  $a$  or  $b$ . (See formal definition in Section 2). In this work, we consider two classical formalisations of regular expressions search, regular expression membership and pattern matching. In the regular expression membership problem, we are given a string  $T$  of length  $n$ , and must decide whether  $T \in L(R)$  for a given regular expression  $R$ . In the regular expression pattern matching problem, we must find all positions  $1 \leq r \leq n$  such that for some  $1 \leq \ell \leq r$ , the substring  $T[\ell..r] \in L(R)$ .

Assume that  $T$  is read-only, and let  $m$  be the length of the regular expression. The classical algorithm by Thompson [9] allows to solve both problems in  $\mathcal{O}(nm)$  time and  $\mathcal{O}(m)$  space by constructing a non-deterministic finite automaton accepting  $L(R)$ . Galil [28] noted that while the space bound of Thompson’s algorithm is optimal in the deterministic setting, the time bound could probably be improved. Since then, the effort has been mainly focused on improving the time complexity of regular expression search. The first breakthrough was achieved by Myers [44], who showed that both problems can be solved in  $\mathcal{O}(mn/\log n + (n+m)\log n)$  time and  $\mathcal{O}(mn/\log n)$  space. Bille and Farach-Colton [119] reduced the space complexity down to  $\mathcal{O}(n^\varepsilon + m)$ , for an arbitrary constant  $\varepsilon > 0$ . This result was further improved by Bille and Thorup [127] who showed an algorithm with running time  $\mathcal{O}(nm(\log \log n)/\log^{3/2} n + n + m)$  time that uses  $\mathcal{O}(n^\varepsilon + m)$  space. The idea of the algorithms by Myers [44], Bille and Farach-Colton [119], and Bille and Thorup [127] is to decompose Thompson’s automaton into small non-deterministic finite automata and tabulate information to speed up simulating the behaviour of the original automaton when reading  $T$ . A slightly different approach was taken by Bille [102] who showed that the small non-deterministic finite automata can be simulated directly using the parallelism built-in in the Word RAM model. For  $w$  being the size of the machine word, Bille showed  $\mathcal{O}(m)$ -space algorithms with running times  $\mathcal{O}(n \frac{m \log w}{w} + m \log w)$  for  $m > w$ ,  $\mathcal{O}(n \log m + m \log m)$  for  $\sqrt{w} < m \leq w$ , and  $\mathcal{O}(\min\{n + m^2, n \log m + m \log m\})$  for  $m \leq \sqrt{w}$ . Finally, Bille and Thorup [140] identified a new parameter affecting the complexity of regular expression search, which is particularly relevant to this paper. Namely, they noticed that in practice a regular expression contains  $d \ll m$  occurrences of the union symbol and Kleene stars, and showed that regular expression membership and pattern matching can be solved in  $\mathcal{O}(m)$  space and  $\mathcal{O}(n \cdot (\frac{d \log w}{w} + \log d))$  time<sup>1</sup>.

It is easy to see, however, that in the general case the time complexity of the algorithms

---

<sup>1</sup>Formally, they consider a parameter  $k$  equal to the number of strings in  $R$ , but it is not hard to see that  $k = \Theta(d)$ .

above remains close to “rectangular”, with some polylogarithmic factors shaved. Recently, fine-grained complexity provided an explanation for this. Backurs and Indyk [212] followed by Bringmann, Grønlund, and Larsen [238] considered a subclass of regular expressions which they refer to as “homogeneous”. Intuitively, a regular expression is homogeneous, if the operators at the same level of the expression are equal. Assume that the alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ . To give a few examples, the following regular expressions are homogeneous:  $R_1 = (P_1|P_2|\dots|P_d)$ ,  $R_2 = P_1(1|2|\dots|\sigma)P_2(1|2|\dots|\sigma)\dots(1|2|\dots|\sigma)P_d$ , and  $R_3 = (P_1|P_2|\dots|P_d)^*$ , where  $P_i$ ,  $1 \leq i \leq d$ , are strings on  $\Sigma$ , i.e. concatenations of characters in  $\Sigma$ . [212, 238] considered both the membership and the pattern matching problems. A careful reader might notice that in the pattern matching setting the expression  $R_1$  corresponds to the famous dictionary matching problem [15] and  $R_2$  to pattern matching with wildcards (don’t cares) [14, 84, 81, 60, 111]. In the membership setting,  $R_3$  corresponds to the Word Break problem [153, 378]. As such, a seemingly simple class of homogeneous regular expressions covers many classical problems in stringology. The authors of [212, 238] provided a complete dichotomy of the time complexities for homogeneous regular expressions in both settings. Namely, they showed that in both settings, every regular expression either allows a solution in near-linear time, or requires  $\Omega((nm)^{1-\alpha})$  time, conditioned on the Strong Exponential Time Hypothesis [70]. The only exception is the Word Break problem in the membership setting, for which [238] showed an  $\mathcal{O}(n(m \log m)^{1/3} + m)$ -time algorithm and a matching combinatorial lower bound (up to polylogarithmic factors). Later, Abboud and Bringmann [251] took an even more fine-grained approach and showed that in general, regular expression pattern matching and membership cannot be solved in time  $\mathcal{O}(nm/\log^{7+\alpha} n)$  for any constant  $\alpha > 0$  under the Formula-SAT Hypothesis. Schepper [325] extended their result by revisiting the dichotomy for homogeneous regular expressions, and showed an  $\mathcal{O}(nm/2^{\Omega(\sqrt{\log \min\{n,m\}})})$  time bound for some regular expressions, and for the remaining ones an improved lower bound of  $\Omega(nm/\text{polylog } n)$ .

By now we seem to have a rather good understanding of the time complexity of regular expression membership and pattern matching. However, in multiple practical applications one needs to work with the input arriving as a stream, one character at a time, without the possibility of going back and retrieving any of the previous characters on demand. This motivates studying both problems in the streaming model of computation. In this model, we mostly focus on designing algorithms with small space complexity, and need to account for storing any information about the input. On the other hand, we allow for randomised algorithms, more specifically Monte Carlo algorithm returning correct answers with high probability (with respect to the length to the input). The field of streaming algorithms for string processing is relatively recent but, because of its practical interest, quickly developing. It started with the paper of Porat and Porat [137], who showed streaming algorithms for exact pattern matching and for the  $k$ -mismatches problem. This was followed by a series of works on streaming pattern matching [176, 199, 243, 213, 249, 270, 290, 283, 315, 323, 356], search of repetitions in streams [142, 241, 259, 288, 299, 300, 289], and recognising formal languages in streams [186, 265, 266, 264, 216, 267, 166, 215, 286, 327].

For a general regular expression membership and pattern matching, it is not hard to see that  $\Omega(m)$  bits of space are required by a reduction from Set Intersection. However, there are at least two interesting special cases of regular expression pattern matching that

admit better streaming algorithms. In the dictionary matching, we are given a dictionary of  $d$  strings of length at most  $m$  over an alphabet  $\Sigma$  and for each position  $r$  in  $T$  must decide whether there is a position  $\ell \leq r$  such that  $T[\ell..r]$  matches a dictionary string. A series of work [137, 176, 199, 243, 270] showed that this problem can be solved by a randomised Monte Carlo algorithm in  $\mathcal{O}(d \log m)$  space and  $\mathcal{O}(\log \log |\Sigma|)$  time per character of the text. In the  $(d - 1)$ -wildcard pattern matching the expression is  $R = P_1(1|2|\dots|\sigma)P_2(1|2|\dots|\sigma)\dots(1|2|\dots|\sigma)P_d$ , where  $P_i$ ,  $1 \leq i \leq d$  are strings of total length at most  $m$  over an alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ . Golan, Kopelowitz, and Porat [292] showed that this problem can be solved by a randomised Monte Carlo algorithm in  $\mathcal{O}(d \log m)$  space and  $\mathcal{O}(d + \log m)$  time per character. The  $d$ -wildcard problem is a special case of the  $k$ -mismatch problem which asks to compute Hamming distances between a pattern and all its alignments to a text for which the Hamming distance does not exceed the given threshold  $k$ . The most space efficient algorithm for the  $d$ -mismatch problem is by Clifford, Kociumaka and Porat [283] and implies an algorithm that uses  $\mathcal{O}(d \log \frac{m}{d})$  words of space and spends  $\mathcal{O}(\log \frac{m}{d}(\sqrt{d \log d} + \log^3 m))$  time per character which is also the most efficient for the  $d$ -wildcard problem.

In a related work, Ganardi et al. [265, 266, 264, 216] considered a variant of the regular expression membership problem, where the automaton describing the regular expression has constant size, and one must tell, for each position  $r$  of  $T$ , whether  $T[r - \ell + 1..r] \in L(R)$ , where  $\ell$  is an integer specified in advance (“window” size). As a culmination of their work, they showed that any randomised Monte Carlo algorithm for this variant of the regular expression membership problem takes either constant, or  $\Theta(\log \log \ell)$ , or  $\Theta(\log \ell)$ , or  $\Theta(\ell)$  bits of space, and provided descriptions of these complexity classes.

This brings the challenge of identifying a structural parameter of a regular expression that determines whether it admits better streaming algorithms. As mentioned earlier, Bille and Thorup [140] observed that in practice the number  $d$  of occurrences of the union symbol and Kleene stars is significantly smaller than the size  $m$  of the expression  $R$ . Furthermore, both the dictionary matching and the wildcard pattern matching can be casted as instances of the regular expression pattern matching, and streaming algorithms with space complexity of the form  $\text{poly}(d, \log n)$  are known. The main goal of this paper is to investigate whether this is also the case for the general regular expression membership and pattern matching.

## 1.1 Our Results

We consider the space complexity of regular expression membership and pattern matching in the streaming model of computation. As by now traditional in streaming string processing, we assume that we receive  $R$  and  $n$  first, preprocess them, and then receive the string  $T$  character by character. We do not account neither for the time nor for the space used during the preprocessing stage. In the membership problem, we must output the answer after having read  $T$  entirely, whereas in the pattern matching problem we must decide whether there is a substring  $T[\ell..r] \in L(R)$  at the moment when we receive the character  $T[r]$ .

Our main conceptual contribution is that we identify the small number  $d$  of occurrences of the union symbol and Kleene stars in  $R$  as allowing for space-efficient streaming algorithms for regular expression membership and pattern matching. More specifically, we

design randomised Monte Carlo algorithms that solve both problems using  $\mathcal{O}(d^3 \text{polylog } n)$  space and  $\mathcal{O}(nd^5 \text{polylog } n)$  time per character of the text (Theorem 1.27). While it was known that the value of  $d$  determines the space complexity in the two special cases of streaming dictionary matching and wildcard pattern matching, our approach works for any regular expression. We leave it as an open problem to obtain algorithms with  $\text{poly}(d, \log n)$  space complexity and  $\text{poly}(d, \log n)$  time complexity.

On a very high-level, our approach is based on storing carefully chosen subsets of occurrences of the strings appearing in  $R$ . As usual in the area, this is easier when the strings are not periodic, that is, any two occurrences of a string  $S$  in  $T$  must be more than  $|S|/2$  characters apart. Of course, this is not always the case, and the usual remedy is to treat periodic and aperiodic strings separately (more specifically, in streaming pattern matching algorithms one applies this reasoning on every prefix of length being a power of 2). The technical novelty of our algorithms is that we apply this reasoning on  $\mathcal{O}(\log n)$  levels, thus obtaining a hierarchical decomposition of a periodic string. Next, because not all occurrences are stored we need to recover the omitted information. Very informally, we need to decide whether a substring of  $T$  sandwiched between two occurrences of strings  $A_1, A_2$  is a label of some run from  $A_1$  to  $A_2$  in the compact Thompson automaton for  $R$ , where the period of the substring is equal to the period of some prefix of length  $2^k$  of one of the strings. The difficulty is that, while the substring has a simple structure, it could be very long, and it is not clear how to implement this computation in a space-efficient manner. We overcome this difficulty by recasting the problem in the language of evaluating a circuit with addition and convolution gates. This technique was introduced by Lokshtanov and Nederlof [145] for designing a space-efficient solution for the subset sum problem. Later, Bringmann [237] replaced complex numbers with computation modulo a prime number  $p$  to obtain a tighter bound on the time and space complexity. In more detail, he designed two solutions, one using the Extended Riemann Hypothesis and the other unconditional but with polynomially higher time and space. We revisit his approach and show that, in fact, one can replace the Extended Riemann Hypothesis by an application of the Bombieri–Vinogradov theorem to achieve the same bounds. We believe that this might be of independent interest. As a consequence of our improvement, we obtain an efficient randomised Monte Carlo algorithm for the following classical problem: given a directed multigraph  $G$  with non-negative integer weights on edges, its two nodes  $v_1, v_2$ , and a number  $x$ , decide whether there is a walk from  $v_1$  to  $v_2$  of total weight  $x$ . Our algorithm requires  $x \cdot \text{poly}(|G|, \log x)$  time and  $\text{poly}(|G|, \log x)$  space.

The rest of the paper is organised as follows. We first remind the necessary definitions in Section 2, and in Section 3 we give an overview of the main technical ideas we introduced in this paper. We describe the new algorithms for regular expression membership and pattern matching in Section 4.2. Finally, in Section 5 we describe how to replace the Extended Riemann Hypothesis with an application of the Bombieri–Vinogradov theorem in Bringmann’s framework and design a space-efficient algorithm for checking if there is a walk of specified weight between two nodes of a directed multigraph.

## 2 Preliminaries

We assume an integer alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$  with  $\sigma$  characters. A string  $Y$  is a sequence of characters numbered from 1 to  $n = |Y|$ . For  $1 \leq i \leq n$ , we denote the  $i$ -th

character of  $Y$  by  $Y[i]$ . For  $1 \leq i \leq j \leq n$ , we define  $Y[i..j]$  to be equal to  $Y[i] \dots Y[j]$ , called a *fragment* of  $Y$ . We call a fragment  $Y[1] \dots Y[j]$  a *prefix* of  $Y$  and use a simplified notation  $Y[.j]$ , and a fragment  $Y[i] \dots Y[n]$  a *suffix* of  $Y$  denoted by  $Y[i..]$ . We say that a fragment  $Y[i..j]$  contains a position  $k$  if  $i \leq k \leq j$ . We denote by  $\varepsilon$  the empty string.

We say that  $X$  is a *substring* of  $Y$  if  $X = Y[i..j]$  for some  $1 \leq i \leq j \leq n$ . The fragment  $Y[i..j]$  is called an *occurrence* of  $X$ . We say that an integer  $p$  is a period of  $Y$  if for each  $1 \leq i \leq |Y| - p$ ,  $Y[i] = Y[i + p]$ . The smallest period of  $Y$  is referred to as *the period* of  $Y$ . We say that  $Y$  is *periodic* with period  $\rho$  if  $\rho$  is the period of  $Y$  and  $\rho \leq |Y|/2$ . For the period  $\rho$  of  $Y$ , we define the string period of  $Y$  to be equal to  $Y[1.. \rho]$ .

For an integer  $k$ , we denote the concatenation of  $k$  copies of  $Y$  by  $Y^k$ . We say that a string  $X$  is *primitive* if  $X \neq Y^k$  for any string  $Y \neq X$  and any integer  $k$ . Note that the string period of a string is always primitive.

**Definition 1.1** (Regular expression). *We define regular expressions over  $\Sigma$  as well as the languages they match recursively. Let  $L(R)$  be the language matched by a regular expression  $R$ .*

- Any  $a \in \Sigma \cup \{\varepsilon\}$  is a regular expression and  $L(a) = \{a\}$ .

For two regular expressions  $A$  and  $B$ , we can form a new expression using one of the three symbols  $\cdot$  (concatenation),  $|$  (union), or  $*$  (Kleene star):

- $A \cdot B$  is a regular expression and  $L(A \cdot B) = \{XY, \text{ for } X \in L(A) \text{ and } Y \in L(B)\}$ ;
- $A | B$  is a regular expression and  $L(A | B) = L(A) \cup L(B)$ ;
- $A^*$  is a regular expression and  $L(A^*) = \bigcup_{k \geq 0} \{X_1 X_2 \dots X_k, \text{ where } X_i \in L(A) \text{ for } 1 \leq i \leq k\}$ .

**Definition 1.2** (Thompson automaton [9]). *For a regular expression  $R$  we define the Thompson automaton of  $R$ ,  $T(R)$ , recursively. This non-deterministic finite automaton (NFA) accepts all strings  $s \in L(R)$ .*

- If  $R = a \in \Sigma \cup \{\varepsilon\}$ ,  $T(R)$  is constructed as in Figure 1.1a;
- If  $R = A \cdot B$ ,  $T(R)$  is constructed as in Figure 1.1b. Namely, the initial state of  $T(A)$  becomes the initial state of  $T(R)$ , the final state of  $T(A)$  becomes the initial state of  $T(B)$ , and the final state of  $T(B)$  becomes the final state of  $T(R)$ ;
- If  $R = A | B$ ,  $T(R)$  is constructed as in Figure 1.1c. Namely, the initial state of  $T(R)$  goes via  $\varepsilon$ -transitions both to the initial state of  $T(A)$  and to the initial state of  $T(B)$ , and the final states of  $T(A)$  and  $T(B)$  go via  $\varepsilon$ -transitions to the final state of  $T(R)$ ;
- If  $R = A^*$ ,  $T(R)$  is constructed as in Figure 1.1d. Namely, the initial state of  $T(R)$  and the final state of  $T(A)$  go via  $\varepsilon$ -transitions both to the initial state of  $T(A)$ , and to the final state of  $T(R)$ .

**Definition 1.3** (Compact Thompson automaton). *Given a Thompson automaton  $T(R)$ , we define the compact Thompson automaton  $T_C(R)$  as the automaton obtained from  $T(R)$  by replacing every maximal path of transitions labelled by  $a_1, a_2, \dots, a_k \in \Sigma$  by a single transition labelled by  $a_1 a_2 \dots a_k$ . The non-empty labels of  $T_C(R)$  are called atomic strings, and the size of the (multiset) of the atomic strings is defined to be the size of  $R$ .*

Figure 1.2 gives an example of the Thompson automata for  $R = b(ab|b)^*ab$ . We note that in general the size of a regular expression is much smaller than the total number of characters in it and is bounded by twice the number of union and Kleene star symbols plus two. The size of a regular expression measures its “complexity”.

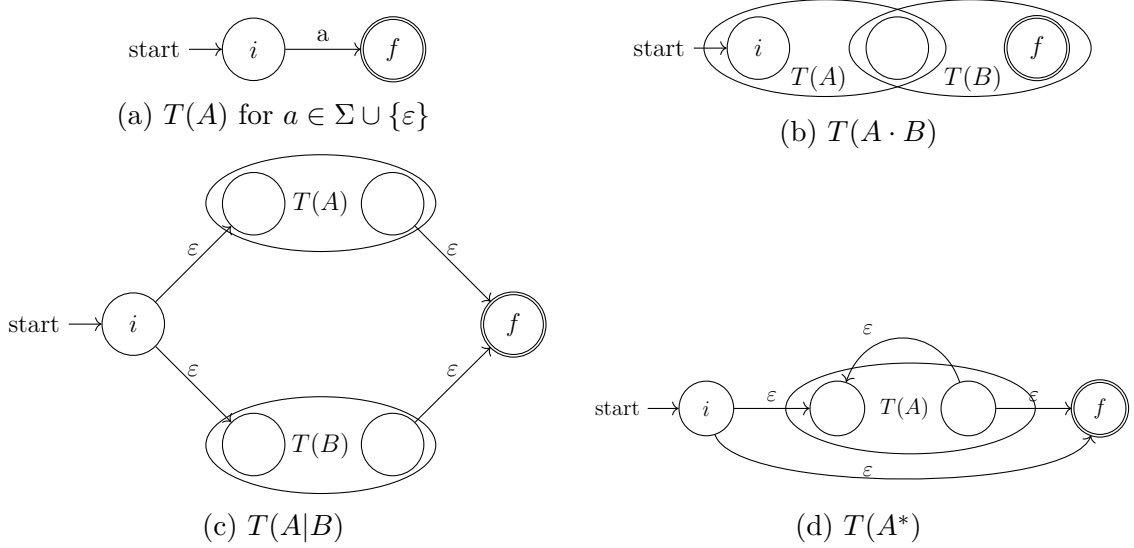


Figure 1.1: Thompson automaton. In each automaton,  $i$  and  $f$  are the initial and final states, respectively.

**Definition 1.4** (Occurrence of a regular expression). *We say that a fragment  $S[i..j]$  of a string  $S$ , where  $1 \leq i \leq j \leq |S|$ , is an occurrence of a regular expression  $R$ , if  $S[i..j] \in L(R)$ , or in other words if there is a walk from the initial state of  $T_C(R)$  to the final state of  $T_C(R)$  such that the concatenation of the labels of the transitions in this walk equals  $S[i..j]$ .*

We will also need a notion of a partial occurrence of  $R$ . Intuitively,  $S[i..j]$  is a partial occurrence of  $R$  if it is a prefix of a string in  $L(R)$ , but we will need a more precise definition.

**Definition 1.5** (Partial occurrence of a regular expression). *We say that a fragment  $S[i..j]$  of a string  $S$ , where  $1 \leq i \leq j \leq |S|$ , is a partial occurrence of a regular expression  $R$  ending with a prefix  $P$  of an atomic string  $A$ , if there is a walk from the initial state of  $T_C(R)$  to the endpoint of the transition corresponding to  $A$  such that the concatenation of the labels of the transitions in this walk equals  $S[i..j]A[|P| + 1..]$ .*

### 3 Technical Overview

In this section, we give an overview of the main technical ideas we introduced in this paper.

**Statement of the problems and the model of computation.** Let us start by giving the precise formulation of the regular expression membership and pattern matching problems and reminding the definition of the streaming model of computation.



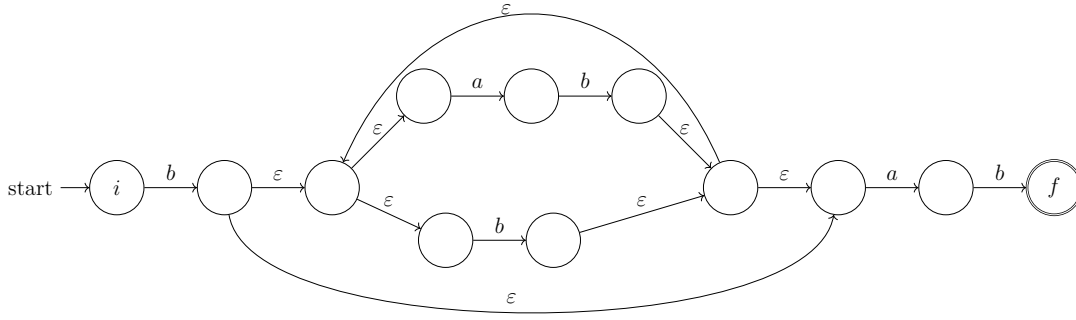
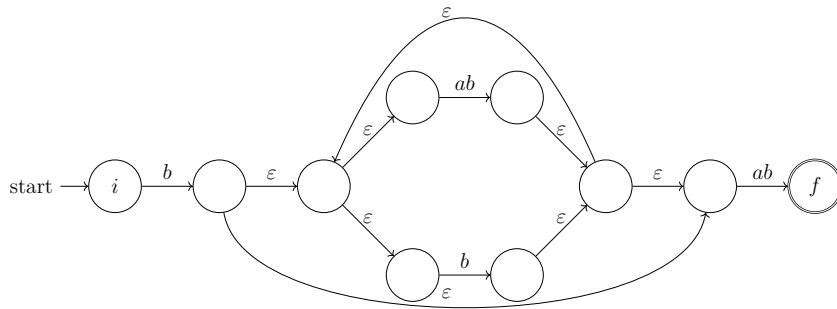

 (a)  $T(b(ab|b)^*ab)$ 

 (b)  $T_C(b(ab|b)^*ab)$ 

 Figure 1.2: The Thompson automaton of the regular expression  $b(ab|b)^*ab$ .

## REGULAR EXPRESSION MEMBERSHIP AND PATTERN MATCHING

Given a string  $T$  of length  $n$  over an alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ , where  $\sigma = n^{\mathcal{O}(1)}$ , and a regular expression  $R$  over  $\Sigma$  of size  $d$ . In the regular expression membership problem, we must decide whether  $T \in L(R)$ . In the regular expression pattern matching problem, we must find all positions  $1 \leq r \leq n$ , such that there exists a position  $1 \leq \ell \leq r$  such that  $T[\ell..r] \in L(R)$ .

We work in the streaming model of computation. As it is now standard in the streaming string processing algorithms, we assume to receive  $n$  and  $R$  first. We do not account neither for the time nor for the space we need to preprocess  $R$ . After having preprocessed  $R$ , we receive  $T$  as a stream, character by character. At the moment we receive the first character of  $T$ , the main phase of the algorithm starts. During the main phase, we account for *all* the space and time used.

**Definitions and tools.** Let  $A_1, A_2, \dots, A_d$  be the atomic strings of the regular expression  $R$ . We define  $\Pi = \{A_i[1.. \min\{2^j, |A_i|\}]: 1 \leq i \leq d, 0 \leq j \leq \lceil \log |A_i| \rceil\}$ . The prefixes of  $A_i$ 's that belong to  $\Pi$  are called *canonical*.

We can assume that all atomic strings have length at most  $n$ , otherwise they never appear in the text and we can ignore them. Formally, during the preprocessing phase we delete all transitions  $(u, v)$  from  $T_C(R)$  that are labelled by atomic strings of lengths larger than  $n$ . We also assume that  $d \leq n$ , otherwise we can use the following solution:

**Claim 1.6.** *Given a streaming text  $T$  of length  $n$  and a regular expression of size  $d \geq n$ .*

Assume that all atomic strings have length at most  $n$  each. There is a deterministic algorithm that solves the membership and the pattern matching problems for  $T$  and  $R$  in  $\mathcal{O}(d^2)$  space and  $\mathcal{O}(d^3)$  time per character of  $T$ .

*Proof.* First note that we can afford storing  $T$  in full. Second, we build a compact trie on the reverses of the atomic strings of  $R$ . The trie occupies  $\mathcal{O}(dn) = \mathcal{O}(d^2)$  space. Finally, let  $\mathcal{F}$  contain all atomic strings  $A$  such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A$  to the final state of  $T_C(R)$ .

Define an array  $D$  of length  $n + 1 = \mathcal{O}(d)$  such that  $D[0]$  contains a singleton set consisting of the starting state of  $T_C(R)$  and  $D[r]$ ,  $1 \leq r \leq n$ , stores all states  $u$  such that  $u$  is the end of some transition labelled by an atomic string and  $T[1..r]$  equals the concatenation of the labels of the transitions in some walk from the starting state of  $T_C(R)$  to  $u$ . Assume that we have constructed  $D[1..r]$ . To compute  $D[r+1]$ , we use the trie to find the atomic strings  $A_1, A_2, \dots, A_q$  equal to  $D[1..r+1], D[2..r+1], \dots$  or  $D[r+1]$  in  $\mathcal{O}(r+q)$  time. Note that  $q \leq d$ . For each atomic string  $A_i$ ,  $1 \leq i \leq q$ , labelling a transition  $(v, w)$ , we add  $w$  to  $D[r+1]$  if there is a state  $u$  in  $D[r+1 - |A_i|]$  such that there is an  $\varepsilon$ -transition path from  $u$  to  $v$ , which can be checked in  $\mathcal{O}(d)$  time and space. In total, the algorithm spends  $\mathcal{O}(d^3)$  time to process a character of  $T$  ( $q = \mathcal{O}(d)$ , and for each  $1 \leq i \leq q$  the set  $D[r+1 - |A_i|]$  contains  $\mathcal{O}(d)$  states). The algorithm reports that  $T \in L(R)$  if  $D[n]$  contains a state  $v$ , which is an endpoint of a transition labelled by some  $A \in \mathcal{F}$ .

In the regular expression pattern matching problem, we define an array  $D$  in the following way. As before,  $D[0]$  contains a singleton set consisting of the starting state of  $T_C(R)$ . For every  $1 \leq r \leq n$ ,  $D[r]$  stores the starting state of  $T_C(R)$  and all states  $u$  such that  $u$  is the end of some transition labelled by an atomic string and  $T[\ell..r]$ , for some  $\ell \leq r$ , equals the concatenation of the labels of the edges in some walk from the starting state of  $T_C(R)$  to  $u$ . Assume that we have constructed  $D[1..r]$ . To compute  $D[r+1]$ , we use the trie to find the atomic strings  $A_1, A_2, \dots, A_q$  equal to  $D[1..r+1], D[2..r+1], \dots$  or  $D[r+1]$  in  $\mathcal{O}(r+q) = \mathcal{O}(d)$  time. For each atomic string  $A_i$ ,  $1 \leq i \leq q$ , labelling a transition  $(v, w)$ , we add  $w$  to  $D[r+1]$  if there is a state  $u$  in  $D[r+1 - |A_i|]$  such that there is an  $\varepsilon$ -transition path from  $u$  to  $v$ , which can be checked in  $\mathcal{O}(d)$  time and space. In total, the algorithm spends  $\mathcal{O}(d^3)$  time to process a character of  $T$ . We report all positions  $r$  such that  $D[r]$  contains a state  $v$ , which is an endpoint of a transition labelled by some  $A \in \mathcal{F}$ .  $\square$

From now on, we assume that all atomic strings have length at most  $n$ , and that  $d \leq n$ . For a string  $P$  and a text  $T$ , denote by  $\text{occ}(P, T)$  the set of the ending positions of the occurrences of  $P$  in  $T$ . Our solutions for streaming regular expression membership and pattern matching are very similar, the main difference is how we define a witness:

**Definition 1.7** (Witness (Membership)). *Let  $P$  be a canonical prefix of an atomic string, and  $r \in \text{occ}(P, T)$ . We say that  $r$  is a witness if  $T[1..r]$  is a partial occurrence of  $R$  ending with  $P$ .*

**Definition 1.8** (Witness (Pattern matching)). *Let  $P$  be a canonical prefix of an atomic string, and  $r \in \text{occ}(P, T)$ . We say that  $r$  is a witness if there exists a position  $1 \leq \ell \leq r$  such that  $T[\ell..r]$  is a partial occurrence of  $R$  ending with  $P$ .*

We exploit the following algorithm, which we refer to as the pattern matching algorithm<sup>2</sup>:

**Theorem 1.9** (cf. [137, Theorem 2]). *Given a pattern of length at most  $n$  and a text  $T$  of length  $n$  over an alphabet of size  $n^{\mathcal{O}(1)}$ . There exists a randomised Monte Carlo streaming algorithm that uses  $\mathcal{O}(\log n)$  space and  $\mathcal{O}(\log n)$  time per character of the text. When it receives  $T[i]$ , it says whether  $i \in \text{occ}(P, T)$ . The algorithm is correct with high probability.<sup>3</sup>*

We also make use of the following well-known fact:

**Fact 1.10** (Fine and Wilf’s periodicity lemma [7]). *If a string  $X$  has two periods of length  $p$  and  $q$  and  $p + q \leq |X|$ , then  $X$  also has a period of length  $\gcd(p, q)$ .*

**Intuition: non-periodic case.** To give intuition behind our solutions, consider a very simple case when every canonical prefix is not periodic. We start with the following simple observation:

**Observation 1.11.** *By Fact 1.10, if  $P$  is not periodic, there can be at most two occurrences of  $P$  in a string of length  $\leq 2|P|$ .*

Therefore, if none of the strings in  $\Pi$  is periodic, we can use the following approach. For each  $P \in \Pi$  and  $T$ , we run the pattern matching algorithm and at any moment store the two most recent witnesses for  $P$  discovered by the algorithm (for membership, witness are defined as in Definition 1.7, and for pattern matching as in Definition 1.8). When the algorithm discovers a new position  $r \in \text{occ}(P, T)$ , we must decide whether it is a witness. Let  $P = A[1.. \min\{2^k, |A|\}]$ , where  $A$  is an atomic string.

If  $k = 0$ , we consider the starting node  $u$  of the transition in the compact Thompson automaton  $T_C(R)$  labelled by  $A$ . Suppose that there is an  $\varepsilon$ -transitions path from the endpoints of the transitions labelled by atomic strings  $A_{i_1}, A_{i_2}, \dots, A_{i_j}$  to  $u$ . We then check if  $(r - 1)$  is a witness for at least one of  $A_{i_1}, A_{i_2}, \dots, A_{i_j}$ . If it is, then  $r$  is a witness. Importantly, if  $r - 1$  is a witness for  $A_{i_{j'}}$ ,  $1 \leq j' \leq j$ , it is the most recent one and is stored in the memory of the instance of the pattern matching algorithm for  $A_{i_{j'}}$  and  $T$ . Suppose now that  $k \geq 1$ . We then must check whether  $(r - 2^{k-1})$  is a witness for  $A[1.. 2^{k-1}]$ . If it is, then  $r$  is a witness for  $P$ . Note that by Observation 1.11, if  $(r - 2^{k-1})$  is a witness for  $A[1.. 2^{k-1}]$ , it is one of the two most recent ones and will be stored by the pattern matching algorithm for  $A[1.. 2^{k-1}]$ .

Let  $\mathcal{F}$  contain all atomic strings  $A$  such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A$  to the final state of  $T_C(R)$ . In the regular expression pattern matching problem, we report all positions  $r$  such that  $r$  is a witness in  $\text{occ}(A, T)$  for some  $A \in \mathcal{F}$ . In the regular expression membership problem,  $T \in L(R)$  if  $n$  is a witness for  $\text{occ}(A, T)$ , for some  $A \in \mathcal{F}$ .

We do not provide the formal analysis of the algorithm, as we only give it for intuition, but it is easy to see that it uses  $\mathcal{O}(d^2 \log^2 n)$  space and  $\mathcal{O}(d \log^2 n)$  time per character of the text (recall that we do not account for the time spent during the preprocessing phase). As all atomic strings have length at most  $n$  and  $d \leq n$ , the algorithm is correct with high probability by Theorem 1.9.

<sup>2</sup>One could also use one of the streaming dictionary matching algorithms (see the introduction), but this does not change the final complexity and makes the description of the algorithm more complex.

<sup>3</sup>With high probability means with probability at least  $1 - 1/n^c$  for any predefined constant  $c > 1$ .

**General case: main technical contributions.** In general, unfortunately, some of the canonical prefixes are periodic we can no longer use Observation 1.11. However, the following generalisation holds:

**Observation 1.12.** *By Fact 1.10, if for a string  $P$  and a string  $X$ ,  $X \leq 2|P|$ , we have  $|\text{occ}(P, X)| > 2$ , then  $P$  is periodic and the set  $\text{occ}(P, X)$  can be represented as an arithmetic progression with difference  $\rho$ , where  $\rho$  is the period of  $P$ .<sup>4</sup>*

Observation 1.12 gives the idea behind our approach for the general case. By this observation, we obtain that every  $r \in \text{occ}(P, T)$ , where  $P$  is a canonical prefix of some atomic string periodic with period  $\rho$ , belongs to a fragment of form  $(\Delta(P))^k$ , where  $\Delta(P) = P[|P| - \rho + 1 ..]$  and  $k$  is an integer. Instead of storing the last two witnesses for each canonical prefix, we would like to store the witnesses in the last two fragments of form  $(\Delta(P))^k$ . However, the number of such witnesses can be large. Our main technical novelty is a compressed representation of such witnesses. We give a high-level overview of the approach we use for the membership problem, our solution to the regular expression pattern matching problem is similar. We show that for each fragment of form  $(\Delta(P))^k$  it suffices to store a small, carefully selected subset of witnesses that belong to this fragment. The remaining ones can be restored in small space at request.

Consider a witness  $r \in \text{occ}(P, T)$ , where  $P \in \Pi$ . By definition, there is a partition  $T[1..r] = T[\ell_1..r_1]T[\ell_2..r_2] \dots T[\ell_m..r_m]$  such that each fragment in the partition, except for the last one, is an atomic string, and the last one equals  $P$ . Furthermore, by Observation 1.12,  $r$  must belong to some fragment  $F = T[i..i + k\rho - 1] = (\Delta(P))^k$ . Let  $m'$  be the index of the first fragment such that  $r_{m'} \geq i$ . Consider the fragment  $W = T[\ell_{m''}..r_{m''}]$ ,  $m' \leq m'' \leq m$ , containing a position  $i + 2\rho - 1$  (we call this position an “anchor”). Note that  $W$  is a canonical prefix of some atomic string and  $r_{m''} \in \text{occ}(W, T)$  is a witness. If there are a few witnesses  $t \in \text{occ}(W, T)$  such that  $T[t - |W| + 1, t]$  contains the anchor  $i + 2\rho - 1$ , we can store them explicitly. Otherwise, there is a periodic fragment containing  $i + 2\rho - 1$ , and we can recurse for it by choosing a new anchor close to its starting point. We choose the definition of anchors (see Section 4.1) so that the recursion stops in a logarithmic number of steps and for some of the anchors there is a witness that we store explicitly for this anchor.

To summarize, the idea of the compact representation of witnesses that belong to a fragment of form  $(\Delta(P))^k$  is to choose a logarithmic set of anchors close to the starting point of the fragment, and for each of these anchors to store a constant number of witnesses for each canonical prefix in  $\Pi$ . Suppose now that  $r \in \text{occ}(P, T)$ , where  $P$  is a canonical prefix of an atomic string  $A$ ,  $r$  belongs to a fragment  $F = T[i..i + k\rho - 1] = (\Delta(P))^k$ . To decide whether it is a witness we use the following approach. From above we know that  $r$  is a witness iff there is a witness  $r' \in \text{occ}(A', T)$ , where  $A'$  is an atomic string, that we store in the compact representation of witness in  $F$ , and there is a path in  $T_C(R)$  from the ending node of the transition labelled by  $A'$  and to the starting node of the transition labelled by  $A$  such that the concatenation of the strings on the edges of the path equals  $T[r' + 1..r - |A|]$  (which is a substring of  $(\Delta(P))^k$ ). Unfortunately, it is not clear how to verify this condition in a straightforward way as we do not have random access neither to  $\Delta(P)$ , nor to the strings on the edges of  $T_C(R)$ . Instead, using anchors again, we show

<sup>4</sup>Note that when  $|\text{occ}(P, X)| \leq 2$ , we can represent  $\text{occ}(P, X)$  as at most two (degenerate) arithmetic progressions of length 1, we will use this fact to simplify the description of the algorithms.

that verifying this condition can be reduced to the following question, where  $G$  is a graph of size  $\text{poly}(d, \log n)$  (see Lemma 1.25 for details):

**WALKS IN A WEIGHTED GRAPH**

Given a directed multigraph  $G$  with non-negative integer weights on edges, two nodes and a number  $x$ , decide if there is a walk from the first node to the second one of total weight  $x$ .

**Walks in a weighted graph and circuits.** In Section 5, we show the following theorem:

**Theorem 1.13.** *There exists an algorithm which, given a directed multigraph  $G$  with non-negative integer weights on edges, its two nodes  $v_1$  and  $v_2$  and a number  $x$ , decides if there is a walk from  $v_1$  to  $v_2$  of total weight  $x$  in  $\mathcal{O}((|E(G)| + |V(G)|^3)x \text{ polylog } x)$  time and  $\mathcal{O}((|E(G)| + |V(G)|^3) \text{ polylog } x)$  space and succeeds with probability at least  $1/2$ .*

Let  $N = |V(G)|$ . For the simpler case when the graph is unweighted, we could use a folklore approach and compute the  $x$ -th power of the adjacency matrix in  $\mathcal{O}(N^3 \log x)$  time and  $\mathcal{O}(N^2)$  space. In order to handle arbitrary weights of edges, we compute the arrays  $C_k$  of bit-vectors of length  $x + 1$ , where  $C_k[u, v][d]$  stores a bit indicator of whether there exists a walk from  $u$  to  $v$  in  $G$  of at most  $2^k$  edges of total weight exactly  $d$ . The following formula holds:

$$C_k[u, v][d] = \bigvee_{\substack{w \in V(G) \\ i \in \{0, \dots, d\}}} C_{k-1}[u, w][i] \wedge C_{k-1}[w, v][d - i]$$

Using the fast Fourier transform to compute the convolutions, we obtain an algorithm with  $\mathcal{O}(N^3 x \log^2 x)$  time and space  $\mathcal{O}(N^2 x)$ .

In our application,  $x$  can be equal to  $n$ , and the approach above uses  $\Omega(n)$  space, which is prohibitive. In order to improve the space complexity, we represent the above computations as a circuit with binary OR and CONVOLUTION $_x$  gates operating on bit-vectors of length  $x + 1$ . Every element  $C_k[u, v]$  requires a separate gate and while computing its value we need to perform  $N$  convolutions, for every possible intermediate node  $w$ , so in total there are  $\mathcal{O}(N^3 \log x)$  gates. The CONVOLUTION $_x$  gates store only the first  $x + 1$  bits of the results, as we never need paths of total weight larger than  $x$ . We are interested only in a single bit of output of the circuit, namely  $C_{\lceil \log x \rceil}[v_1, v_2][x]$ . If there were only OR gates in the circuit, we could store only the  $x$ -th element at each gate. In order to handle also CONVOLUTION $_x$  gates, we use the discrete Fourier transform over a suitably chosen ring.

We use the technique introduced by Lokshtanov and Nederlof [145] and then modified Bringmann [237] to work with numbers modulo  $p$  instead of complex numbers. Informally, they show that if we operate on  $\mathbb{Z}_p^t$  (vectors of length  $t$  with elements in  $\mathbb{Z}_p$  for suitably chosen  $p$  and  $t$ ) instead of the bit-vectors, we can compute  $\text{out}(C)[x]$ , the  $x$ -th element of the output of the circuit  $C$  in  $\mathcal{O}(|C|t \text{ polylog } p)$  time and  $\mathcal{O}(|C| \log p)$  space (see details in Theorem 1.28). However, there are technical difficulties that we need to overcome to apply their technique to our solution. The approach of Bringmann [237] requires that  $t > x$  and  $\mathbb{Z}_p$  contains a  $t$ -th root of the unity. The main difficulty is to choose these numbers as small as possible as they directly affect the complexity of the algorithm. This question was also faced by Bringmann [237], who showed two variants of the framework, one using

the Extended Riemann Hypothesis and the other unconditional but with polynomially higher time and space, which is not good enough for our streaming application. By using Bombieri–Vinogradov theorem (see details in Theorem 1.33) and facts about counting primes in arithmetic progressions, we obtain an unconditional time bound comparable to that of Bringmann that assumes the Extended Riemann Hypothesis.

## 4 Membership and Pattern Matching

In this section, we address the problems of regular expression membership and pattern matching in the streaming model of computation. In Section 4.1, we introduce a notion of *anchors* that is the key to achieving the desired space complexity. In Section 4.2, we describe the algorithms.

### 4.1 Anchors

In this section we define the notion of anchors that will allow us to store all occurrences (partial or not) of regular expressions detected by the algorithm efficiently.

**Definition 1.14** (Anchors). *Consider a periodic string  $W$  with period  $\rho$ . Set the anchor  $a_0 = 2\rho$  and the period  $\rho_0 = \rho$ . Suppose that  $(a_r, \rho_r)_{r=0}^{q-1}$  are defined. We define  $(a_q, \rho_q)$  recursively. Consider the set  $S$  of fragments  $W[i..j]$  of  $W$  satisfying the following properties:*

1.  $W[i..j]$  contains  $a_{q-1}$  and is periodic with period  $\pi < \rho_{q-1}$ ;
2.  $i + 4\pi - 1 \leq a_{q-1}$  (there are at least four repetitions of the string period of  $W[i..j]$  before  $a_{q-1}$ );
3.  $a_{q-1} \leq j - 4\pi - r$ , where  $r = (j - i + 1) \pmod{\pi}$  (there are at least four repetitions of the string period of  $W[i..j]$  after  $a_{q-1}$ ).

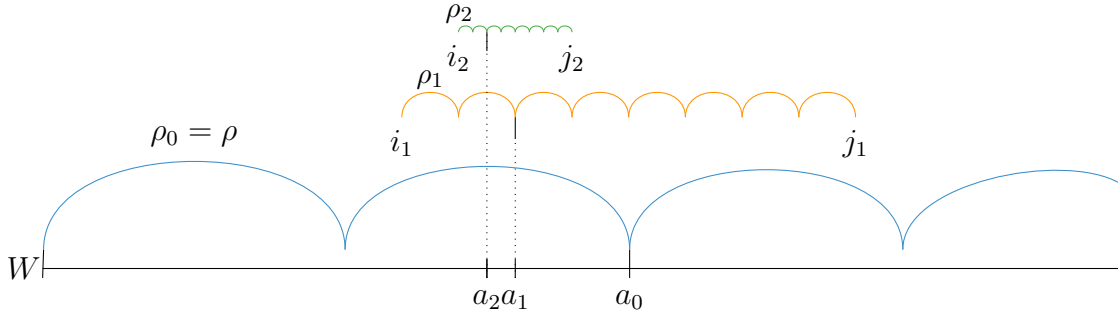
Let  $i_q = \min\{i : W[i..j] \in S\}$  and  $j_q = \max\{j : W[i_q..j] \in S\}$ . If  $W[i_q..j_q]$  is undefined, recursion stops. Otherwise,  $\rho_q$  is defined to be the period of  $W[i_q..j_q]$  and  $a_q := i_q + 2\rho_q - 1$ .

Let  $\mathcal{A}(W) = \{a_0, a_1, \dots, a_Q\}$ , where  $(a_Q, \rho_Q)$  is the last defined anchor-period pair. We call  $\mathcal{A}$  the generator set of anchors of  $W$ . Define  $\mathcal{A}^*(W) = \left[ \bigcup_{\Delta \in \mathbb{Z}_+} (\mathcal{A}(W) + \Delta \cdot \rho) \right] \cap [1, |W| - 2\rho]$ . We refer to  $\mathcal{A}^*$  simply as the set of anchors of  $W$ . For an illustration, see Fig. 1.3.

(In the next section we slightly abuse notation and extend the notion of the set of anchors to infinite strings in a natural way, i.e. for an infinite string  $W$  with period  $\rho$  the set  $\mathcal{A}^*(W) = \left[ \bigcup_{\Delta \in \mathbb{Z}_+} (\mathcal{A}(W) + \Delta \cdot \rho) \right]$ .) We first show that the generator set of anchors has logarithmic size:

**Lemma 1.15.** *Let  $W$  be a periodic string with period  $\rho$ . We have  $|\mathcal{A}(W)| = \mathcal{O}(\log \rho)$ .*

*Proof.* Let  $\mathcal{A}(W) = \{a_0, a_1, \dots, a_Q\}$ . For each  $0 \leq q \leq Q$ , let  $W[i_q..j_q]$  be the fragment associated with  $a_q$ , and  $\rho_q$  be its period. (In particular, for  $q = 0$  we have  $i_0 = 1$ ,  $j_0 = |W|$ ,


 Figure 1.3: Anchors of a periodic string  $W$  with period  $\rho$ .

$\rho_0 = \rho$ .) We show that for each  $1 \leq q \leq Q$  we have  $|W[i_q \dots j_q]| \leq 2\rho_{q-1}$ . This implies, in particular, that  $|W[i_q \dots a_{q-1}]| \leq 2\rho_{q-1}$  and therefore  $\rho_q \leq \rho_{q-1}/2$ . The lemma follows immediately.

Fix  $0 \leq q \leq Q$ . Assume by contradiction that  $|W[i_q \dots j_q]| > 2\rho_{q-1}$ . We then have that  $W[i_q \dots j_q]$  has periods  $\rho_{q-1}$  and  $\rho_q$  and  $\rho_{q-1} + \rho_q < |W[i_q \dots j_q]|$ . Hence,  $W[i_q \dots j_q]$  is periodic with period  $\pi = \gcd(\rho_{q-1}, \rho_q)$  by Fact 1.10. The substring  $W[i_q \dots j_q]$  contains a full copy of  $W[i_{q-1} \dots i_{q-1} + \rho_{q-1} - 1]$ . Therefore,  $W[i_{q-1} \dots i_{q-1} + \rho_{q-1} - 1]$  has a period  $\pi$ , which implies that it equals  $(W[i_{q-1} \dots i_{q-1} + \pi - 1])^{\rho_{q-1}/\pi}$ , i.e. the string period of  $W[i_{q-1} \dots j_{q-1}]$  is not primitive, a contradiction.  $\square$

**Definition 1.16.** Let  $W$  be a periodic string with period  $\rho$ . We say that a fragment  $F = W[i \dots j]$  is anchored by an anchor  $a \in \mathcal{A}^*(W)$  if  $i \leq a \leq j$  and for any strings  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  such that  $V \neq W[1 \dots \rho]$  there are at most eight occurrences of  $F$  in  $UV(W[1 \dots j])$  containing the anchor (i.e., containing  $|U| + |V| + a$ ).

**Lemma 1.17.** Let  $W$  be a periodic string with period  $\rho$ . Consider a fragment  $W[\ell \dots r]$  of length at least  $4\rho$  and a partitioning  $W[\ell \dots r] = W[\ell_1 \dots r_1]W[\ell_2 \dots r_2] \dots W[\ell_k \dots r_k]$ .

- (a) There exists  $1 \leq k' \leq k$  such that  $W[\ell_{k'} \dots r_{k'}]$  is anchored by an anchor  $a \in \mathcal{A}^*(W) \cap [r - 4\rho + 1, r]$ .
- (b) If, in addition,  $1 \leq \ell \leq 2\rho$ , there exists  $1 \leq k' \leq k$  such that  $W[\ell_{k'} \dots r_{k'}]$  is anchored by an anchor  $a \in \mathcal{A}^*(W) \cap [1, 4\rho]$ .

*Proof.* The high-level idea of the proof of (a) and (b) is as follows. Let  $\mathcal{A}(W) = \{a_0, a_1, \dots, a_Q\}$ . For every  $0 \leq q \leq Q$  and an integer  $\Delta > 0$  to be determined later, define  $a_q^\Delta := a_q + \Delta \cdot \rho$ . Let  $F_q = W[\ell_{k_q}, r_{k_q}]$  be the fragment that contains  $a_q^\Delta$ . We show that either  $F_q$  is anchored by  $a_q^\Delta$  or  $q < Q$ , which yields the lemma. We exploit two auxiliary claims:

**Claim 1.18.** Let  $\pi$  be the period of  $F_q$ ,  $0 \leq q \leq Q$ . If  $F_q$  is not anchored by  $a_q^\Delta$ , then there exist  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1 \dots \rho]$  such that there is a fragment  $S[p \dots t]$  of the string  $S = UV(W[1 \dots r_q])$  satisfying the following properties:

1.  $p \leq |U| + |V| + a_q^\Delta \leq t$  (the fragment contains the anchor);
2.  $S[p \dots t]$  is periodic with period  $\pi$ ;

3.  $p + 7 \cdot \pi \leq |U| + |V| + a_q^\Delta$  (there are at least seven repetitions of the string period of the fragment before the anchor);
4.  $|U| + |V| + a_q^\Delta \leq t - 6\pi - r$ , where  $r = t - p + 1 \pmod{\pi}$  (there are at least six repetitions of the string period of the fragment after the anchor).

*Proof.* Let  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1.. \rho]$  such that there are least eight occurrences of  $F_q$  in  $S = UV(W[1.. r_q])$  containing  $a := |U| + |V| + a_q^\Delta$ . Let the last eight occurrences be  $S[p_k.. t_k]$ ,  $1 \leq k \leq 8$ . As all occurrences contain  $a$ , the length of  $S[p_1.. t_8]$  is at most  $2|F_q|$ . By Observation 1.12, we obtain that  $S[p_1.. t_8]$  is periodic with period  $\pi$ ,  $p_k = p_1 + (k-1)\pi$ , and  $t_k = t_1 + (k-1)\pi$ . As  $p_8 = p_1 + 7\pi \leq a$ , we have  $S[p_1.. a] \geq 7\pi$ . On the other hand,  $t_2 - r + 1 \geq t_1 + 1 \geq a + 1$ . Therefore,  $|S[a+1.. t_8 - r]| \geq |S[t_2 - r + 1.. t_8 - r]| \geq 6\pi$ . By taking  $p = p_1$  and  $t = t_8$ , we obtain the claim. For an illustration, see Fig. 1.4.  $\square$

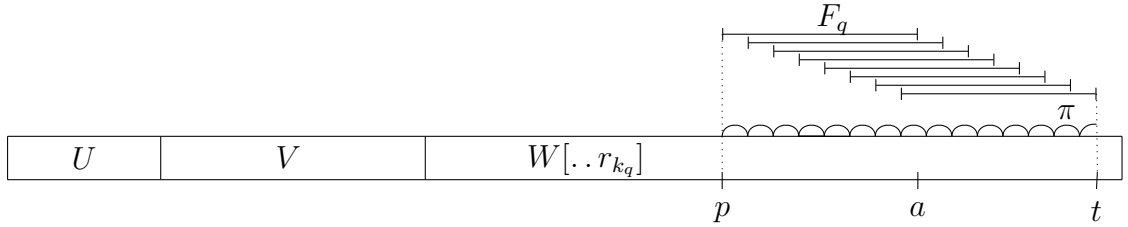


Figure 1.4: Illustration of Claim 1.18.

**Claim 1.19.** Assume that  $q \geq 0$  and that the period of  $F_q$  is  $\pi < \rho_q$ . If  $F_q$  is not anchored by  $a_q^\Delta$ , then  $q < Q$ .

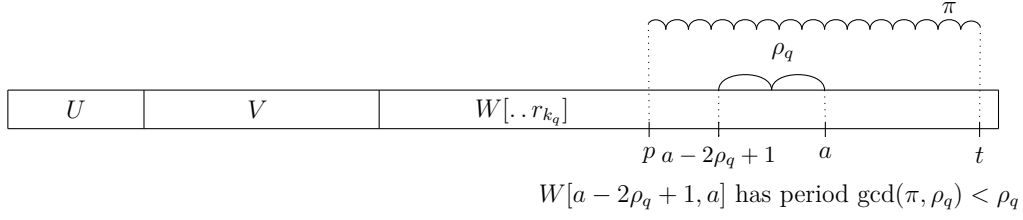
*Proof.* By Claim 1.18, there exist  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1.. \rho]$  such that there is a fragment  $S[p.. t]$  of the string  $S = UV(W[1.. r_q])$  periodic with period  $\pi$  that contains  $a := |U| + |V| + a_q^\Delta$  and such that there are at least six repetitions of the string period before and after  $a$ .

Let us show that  $a - 2\rho_q + 1 \leq p < t \leq a + 2\rho_q$ . Suppose by contradiction that  $p < a - 2\rho_q + 1$ . We have that  $S[a - 2\rho_q + 1.. a] = W[a_q^\Delta - 2\rho_q + 1.. a_q^\Delta]$  (note that by definition  $a_q^\Delta \geq 2\rho_q$  for any  $\Delta$ ). Furthermore,  $W[a_q^\Delta - 2\rho_q + 1.. a_q^\Delta]$  has periods  $\rho_q$  (by definition of  $i_q$  and  $a_q^\Delta$ ) and  $\pi$  (by the assumption). By Fact 1.10,  $W[a_q^\Delta - 2\rho_q + 1.. a_q^\Delta]$  has a period  $\gcd(\rho_q, \pi) < \rho_q$ . As  $W[a_q^\Delta - 2\rho_q + 1.. a_q^\Delta]$  contains a full copy of the string period of  $W[i_q.. j_q]$ , we obtain that it is not primitive, a contradiction. (See Fig. 1.5). To show that  $t \leq a + 2\rho_q$ , note that  $S[a+1.. a+2\rho_q] = W[a_q^\Delta + 1.. a_q^\Delta + 2\rho_q]$ ,  $a_0^\Delta + 2\rho \leq |W|$  and, for  $q \geq 1$ ,  $a_q^\Delta + 2\rho_q \leq a_{q-1}^\Delta$  by definition of  $i_q$  and  $a_q$ . Therefore, for all  $q \geq 0$ ,  $W[a_q^\Delta + 1.. a_q^\Delta + 2\rho_q]$  is periodic with period  $\rho_q$ . The rest of the argument is analogous.

From  $a - 2\rho_q + 1 \leq p < t \leq a + 2\rho_q$  and the fact that  $S[p.. t]$  contains at least six repetitions of its string period before and after  $a$ , we obtain that  $i_{q+1}, j_{q+1}$  and hence  $a_{q+1}, \rho_{q+1}$  are well-defined, which completes the proof of the claim.  $\square$

We are now ready to show (a) and (b).




 Figure 1.5: Illustration of the proof of Claim 1.19, case  $p < a - 2\rho_q + 1$ .

- (a) Let  $\Delta \leq \lfloor |W|/\rho \rfloor - 2$  be the smallest integer such that  $a_0 + \Delta \cdot \rho \geq r - 4\rho$ . Note that  $\Delta$  is well-defined. Consider an anchor  $a_0^\Delta = a_0 + \Delta \cdot \rho \in \mathcal{A}^*(W) \cap [r - 4\rho + 1, r]$ . Let  $F_0 = W[\ell_{k_0} \dots r_{k_0}]$  be the fragment that contains  $a_0^\Delta$  and  $\pi$  be its period. If  $F_0$  is anchored by  $a_0^\Delta$ , we are done. Otherwise, by Claim 1.18, there exist  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1 \dots \rho]$  such that there is a fragment  $S[p \dots t]$  of the string  $S = UV(W[..r_{k_0}])$  periodic with period  $\pi$  that contains  $a := |U| + |V| + a_0^\Delta$  and such that there are at least six repetitions of the string period of  $F_0$  before and after  $a$ . As we have  $r_{k_0} - a_0^\Delta + 1 \leq r - a_0^\Delta + 1 \leq 4\rho$ , there is  $\pi \leq 2\rho/3$ . It follows that  $i_1, j_1$  and hence  $a_1, \rho_1$  are well-defined, i.e.  $0 < Q$ .

We now show that for arbitrary  $q \geq 1$  either  $a_q^\Delta = a_q + \Delta \cdot \rho$  anchors  $F_q$  or the period  $\pi$  of  $F_q$  is smaller than  $\rho_q$ , which by Claim 1.19 implies that  $q < Q$ . By our choice of  $\Delta$ ,  $F_q$  is well-defined. If  $F_q$  is anchored by  $a_q^\Delta$ , we are done. Otherwise, by Claim 1.18, there exist  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1 \dots \rho]$  such that there is a fragment  $S[p \dots t]$  of the string  $S = UV(W[..r_{k_q}])$  periodic with period  $\pi$  that contains  $|U| + |V| + a_q^\Delta$  and such that there are at least six repetitions of the string period of  $S[p \dots t]$  before and after  $|U| + |V| + a_q^\Delta$ . Recall that  $a_q^\Delta = (i_q + 2\rho_q) + \Delta \cdot \rho$ .

First, we have that  $\pi \neq \rho_q$ , otherwise we could have extended  $W[i_q \dots j_q]$  to the left. Second, let us show that the case  $\pi > \rho_q$  is impossible. Suppose otherwise. If  $t - 4\pi + 1 \leq |U| + |V| + a_{q-1}^\Delta$ , then  $W[a_{q-1}^\Delta + 1 \dots a_{q-1}^\Delta]$  contains a copy of  $S[1 \dots 2\pi]$ , and therefore  $S[1 \dots 2\pi]$  has a period  $\rho_q$ . By Fact 1.10, the string period  $S[1 \dots \pi]$  of  $S$  is not primitive, a contradiction. (See Fig. 1.6a.) Otherwise,  $|U| + |V| + a_{q-1}^\Delta$  is contained in  $S[p \dots t]$ , which has period  $\pi > \rho_{q-1}$ . In addition, there are at least four repetitions of the string period of  $S[p \dots t]$  before and after  $|U| + |V| + a_{q-1}^\Delta$ , and  $p < |U| + |V| + a_q^\Delta - 2\rho_q \leq |U| + |V| + i_q$ , a contradiction with the choice of  $W[i_q \dots j_q]$ . (See Fig. 1.6b.) It finally follows that  $\pi < \rho_q$  and therefore by Claim 1.19,  $q < Q$ .

- (b) Let  $\Delta$  be the smallest integer such that  $a_0 + (\Delta - 2)\rho \geq \ell$  (note that  $\Delta = 1, 2$ ). Consider an anchor  $a_0^\Delta = a_0 + \Delta \cdot \rho \in \mathcal{A}^*(W) \cap [1, 4\rho]$ . We first show that either  $F_0$  is caught by  $a_0^\Delta$ , or  $0 < Q$ . If  $F_0$  is anchored by  $a_0^\Delta$ , we are done. Otherwise, let  $\pi$  be the period of  $F_0$ . We claim that  $\pi < \rho = \rho_0$ . As  $F_0$  is not anchored, by Claim 1.18 there exist  $U \in \Sigma^*$ ,  $V \in \Sigma^\rho$  with  $V \neq W[1 \dots \rho]$  such that there is a fragment  $S[p \dots t]$  of the string  $S = UV(W[..r_{k_0}])$  periodic with period  $\pi$  that contains  $a := |U| + |V| + a_0^\Delta$  and such that there are at least six repetitions of the string period of  $S$  before and after  $a$ . We can immediately rule out the case  $\pi = \rho$  as  $a \leq 4\rho$  and  $V \neq W[1 \dots \rho]$ . Consider now the case  $\pi > \rho_0$ . Consider the suffix  $S[a + 1 \dots] = W[a_0^\Delta + 1 \dots r_{k_0}]$ . It contains an occurrence of  $S[1 \dots 2\pi]$ . By Fact 1.10,  $S[1 \dots 2\pi]$  has a period  $\gcd(\pi, \rho)$ , and therefore the string period of  $S$  is

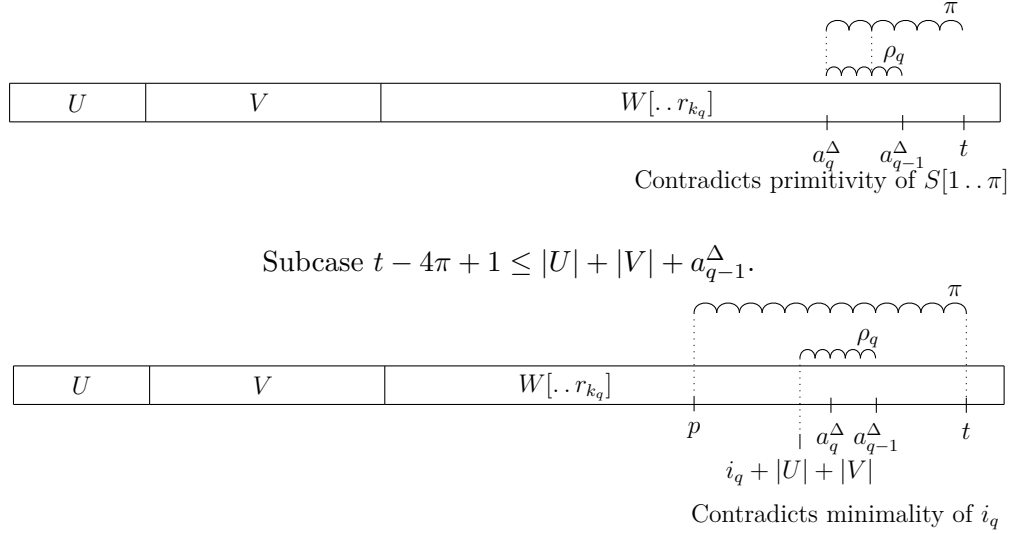


Figure 1.6: Illustration of the proof of Lemma 1.17(a), case  $\pi > \rho_q$ .

not primitive, a contradiction. For  $q \geq 1$ ,  $F_q$  is well-defined by our choice of  $\Delta$ . The rest of the argument repeats the argument in the proof of (a).

This concludes the proof of Lemma 1.17.  $\square$

## 4.2 Algorithms

In this section, we give a description of our new streaming algorithms for regular expression membership and pattern matching. The structure of the algorithms is very similar, the main difference is how we define a witness (see Definitions 1.7 and 1.8). For this reason, we describe the algorithms in parallel. Recall that  $T$  is a string of length  $n$  over an alphabet  $\Sigma = \{1, 2, \dots, \sigma\}$ , where  $\sigma = n^{\mathcal{O}(1)}$ , and  $A_1, A_2, \dots, A_d$  are the atomic strings for the regular expression  $R$ . Recall also that  $\Pi$  is the set of canonical prefixes of the atomic strings, defined as  $\Pi = \{A_i[1 \dots \min\{2^j, |A_i|\}] : 1 \leq i \leq d, 0 \leq j \leq \lceil \log |A_i| \rceil\}$ .

We make use of the following corollary of Fact 1.10:

**Corollary 1.20** (Of Fact 1.10). *For a primitive string  $X$  of length  $x$ , a string  $D = XX$  can contain only two occurrences of string  $X$ ,  $D[1 \dots x]$  and  $D[x + 1 \dots 2x]$ .*

**Preprocessing.** We start by deleting all transitions  $(u, v)$  from  $T_C(R)$  that are labelled by atomic strings of lengths larger than  $n$ .

Let  $\mathcal{F}$  contain all atomic strings  $A$  such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A$  to the final state of  $T_C(R)$ . In addition, for each atomic string  $A$ , compute the subset  $A_{i_1}, A_{i_2}, \dots, A_{i_j}$  of atomic strings such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A_{i_{j'}}$ ,  $1 \leq j' \leq j$ , to the starting point of the transition labelled by  $A$ .

For each periodic  $P \in \Pi$ , consider a string  $\Delta(P) = P[|P| - \rho + 1 \dots |P|]$ , where  $\rho$  is the period of  $P$ . During the preprocessing step, the algorithm computes the generator set of anchors  $\mathcal{A}$  (Definition 1.14) for the string  $W = (\Delta(P))^\infty$ .

Define the *overlap* of two strings  $X$  and  $Y$  as the maximal length of a suffix of  $X$  that equals a prefix of  $Y$ ,  $\Pi(P)$  to be the set of all canonical prefixes such that their overlap with  $W$  is at least  $2\rho$ , and  $\text{Overlap}(P) = \bigcup_{P' \in \Pi(P)} \{\ell : \ell \text{ is the overlap of } P' \text{ and } W\}$ . The algorithm computes  $\Pi(P)$  and  $\text{Overlap}(P)$  during the preprocessing step as well.

For regular expression pattern matching, it also computes the smallest integer  $\mu(P)$  such that  $p = \mu(P) \cdot \rho \in \text{occ}(P, W)$  and  $p$  is a witness for  $P$  in  $W$  (in the sense of Definition 1.8).

Finally, the algorithm creates a directed graph  $G(P) = (V, E)$  from the compact Thompson automaton  $T_C(R)$ . Consider again the string  $W = (\Delta(P))^\infty$ . For each canonical prefix  $P' \in \Pi(P)$ , the algorithm creates a node  $v \in V$  corresponding to a pair  $(P', r)$ , where  $r$  is the remainder of the overlap of  $P'$  and  $W$  modulo  $\rho$ . Additionally, for every fragment  $W[i..j] = P'' \in \Pi$  which is anchored by an anchor  $a \in \mathcal{A}^*(W)$ , it creates a node  $v \in V$  corresponding to a pair  $(P'', j \pmod{\rho})$  (for two identical prefix-remainder pairs, it creates just one node).

Consider two nodes  $v', v'' \in V$ . Suppose that  $v'$  corresponds to  $(P', r')$  and  $v''$  to  $(P'', r'')$ , where  $P', P''$  are canonical prefixes of atomic strings  $A', A''$ , respectively, and  $r', r''$  are remainders modulo  $\rho$ . For  $0 < \ell \leq 10\rho$ , the algorithm adds an edge  $(v', v'')$  of length  $\ell$  to  $E$  if there is a walk in  $T_C(R)$  from the ending state of the transition labelled by  $A'$  to the ending state of the transition labelled by  $A''$  such that the concatenation of the labels in this walk equals to a string  $L = \Delta(P)[r'..]\Delta(P)^\alpha\Delta(P)[..r'']$ , where the integer power  $\alpha$  is chosen so that  $|L| = \ell$  (in other words, the concatenation equals to a fragment of  $W$  with the offsets defined by  $v'$  and  $v''$  and of an appropriate length, if such a fragment does not exist, the algorithm does not create the edge).

It might seem that the resulting graph is infinite, but as we show below, this is not the case.

**Claim 1.21.** *Consider all occurrences  $W[\ell_i..r_i]$ ,  $i \in \mathbb{Z}_+$ , of a string  $X$  in  $W$ . The size of the set  $\{r_i \pmod{\rho} : W[\ell_i..r_i] \text{ is anchored by } a \in \mathcal{A}^*(W), i \in \mathbb{Z}_+\}$  is  $\mathcal{O}(\log \rho)$ .*

*Proof.* We consider two cases:  $|X| \geq 2\rho$  and  $|X| < 2\rho$ . In the first case, by Fact 1.10, if there is at least one occurrence of  $X$  in  $W$ , then  $X$  is periodic with period  $\rho$ . By Corollary 1.20, we have  $r_i = q \pmod{\rho}$  for some fixed  $q$  and all  $i \in \mathbb{Z}_+$ . The claim follows from Lemma 1.15.

In the second case, for all  $a \in \mathcal{A}^*(W)$  such that  $a \geq 2\rho$ , and for all strings  $U \in \Sigma^*, V \in \Sigma^\rho$  such that  $V \neq W[1.. \rho]$ , all occurrences of  $X$  that contain  $|U| + |V| + a$  are contained in  $(UVW)[|U| + |V| + a - 2\rho + 1..|U| + |V| + a - 2\rho - 1] = W[a - 2\rho + 1..a - 2\rho - 1]$ . It follows that for all  $a, a' \in \mathcal{A}^*(W)$  such that  $a, a' \geq 2\rho$  and  $a = a' \pmod{\rho}$ , the sets  $\{r_i \pmod{\rho} : W[\ell_i..r_i] \text{ is anchored by } a, i \in \mathbb{Z}_+\}$  and  $\{r_i \pmod{\rho} : W[\ell_i..r_i] \text{ is anchored by } a', i \in \mathbb{Z}_+\}$  are equal. Moreover, each of them contains only a constant number of elements. Therefore, the size of the set  $\{r_i \pmod{\rho} : W[\ell_i..r_i] \text{ is anchored by } a \in \mathcal{A}^*(W), a \geq 2\rho, i \in \mathbb{Z}_+\}$  is  $\mathcal{O}(\log \rho)$  by Lemma 1.15. It remains to estimate the size of the analogous sets for anchors smaller than  $2\rho$ . The number of such anchors is  $\mathcal{O}(\log \rho)$  by Lemma 1.15, and each of them can anchor only a constant number of occurrences of  $X$ . The claim follows.  $\square$

**Corollary 1.22.**  *$G(P)$  contains  $|V| = \mathcal{O}(d \log^2 n)$  nodes and  $|E| = \mathcal{O}(d^2 \log^4 n)$  edges, and can be constructed in  $\mathcal{O}(\rho \cdot d^3 \log^4 n)$  time.*

*Proof.* The size of  $\Pi$ , and consequently  $\Pi(P)$ , is  $\mathcal{O}(d \log n)$ . For each string  $P' \in \Pi(P)$ , the remainder  $r$  of the overlap of  $P'$  and  $W$  modulo  $\rho$  is defined in a unique way. By Claim 1.21, for each string  $P'' \in \Pi$  there are  $\mathcal{O}(\log \rho)$  different remainders  $r$  modulo  $\rho$  such that there is an anchored occurrence of  $P''$  ending at a position  $p = r \pmod{\rho}$ . Thus  $|V| = \mathcal{O}(d \log^2 n)$ .

Observe that the interval  $[0, 10\rho]$  can contain only a constant number of values  $\ell$  such that the power  $\alpha$  is an integer. Hence for each pair of nodes we have only a constant number of possible edges, so  $|E| = \mathcal{O}(d^2 \log^4 n)$ . For each edge, we can check whether it exists in time  $\mathcal{O}(\rho \cdot |T(R)|) = \mathcal{O}(|\rho| \cdot d)$ .  $\square$

**Corollary 1.23** (Of Theorem 1.13). *There exists an algorithm which, given the graph  $G(P)$ , its two nodes  $v_1$  and  $v_2$  and a number  $x \leq n$ , decides if there is a walk from  $v_1$  to  $v_2$  of total weight  $x$  in  $\mathcal{O}(xd^3 \text{polylog } n)$  time and  $\mathcal{O}(d^3 \text{polylog } n)$  space and succeeds with high probability.*

*Proof.* Recall that  $\rho \leq n$ . We substitute the bounds from Corollary 1.22 into Theorem 1.13. Then we repeat the algorithm of Theorem 1.13  $2c \lceil \log n \rceil$  times and output the majority answer to obtain a success probability of at least  $1 - 1/n^c$ .  $\square$

**Main phase.** During the main phase of the algorithm, we run the pattern matching algorithm (Theorem 1.9) for every  $P \in \Pi$ . For every non-periodic  $P \in \Pi$ , we store (at most) two latest witnesses. For every periodic  $P \in \Pi$ , we run the pattern matching algorithm for  $\Delta(P) = P[|P| - \rho + 1 \dots |P|]$  and  $T$ , where  $\rho$  is the period of  $P$ .

**Definition 1.24.** *We say that a fragment  $T[i \dots j]$  is a streak of a string  $X$  if  $T[i \dots j] = X^k$  for some integer  $k \geq 1$  and it is maximal, i.e. it cannot be extended neither to the left nor to the right.*

The pattern matching algorithm detects streaks of  $\Delta(P)$  in the arrived prefix of  $T$ . Every witness  $r$  for  $P$  belongs to  $\text{occ}(P, T)$  and by Observation 1.12 ends in such a streak. At any moment of the algorithm, we store (at most) two latest streaks and a compact representation of witnesses in  $\text{occ}(P, T)$  that end in it. For membership testing we assume that the witnesses are defined as in Definition 1.7, and for pattern matching as in Definition 1.8. Perhaps a bit counter-intuitively, the representation contains witnesses from  $\text{occ}(P, T)$  and witnesses for other canonical prefixes as well, the reason for it will become clear later. Formally, the representation of a streak  $S = T[i \dots j]$  consists of the following elements, where  $\rho = |\Delta(P)|$ :

1. For each  $P' \in \Pi(P)$  and its overlap  $\ell$  with  $W$ , the representation contains  $p = i + \ell - 1$  if  $p \in \text{occ}(P', T)$  and is a witness;
2. All witnesses in  $\text{occ}(P, T)$  that belong to the interval  $[i \dots i + 12\rho - 1]$ ;
3. For every  $\ell \in \text{Overlap}(P)$ , all witnesses in  $\text{occ}(P, T)$  that belong to the interval  $[i + \ell \dots (i + \ell - 1) + 8\rho]$ ;
4. For every  $P' \in \Pi$ , all witnesses  $r \in \text{occ}(P', T)$  such that  $T[r - |P'| + 1 \dots r]$  is a fragment of  $S$  and is anchored by an anchor in  $\mathcal{A}^*(F) \cap [i, i + 4\rho]$ .

By Observation 1.12 and Lemma 1.15, the compact representation of witnesses in a streak has size  $\mathcal{O}(d \log^2 n)$ . The algorithm uses the compact representations of witnesses to decide whether a newly detected occurrence  $r \in \text{occ}(P, T)$  for some  $P \in \Pi$  is a witness:

**Lemma 1.25.** *Let  $d < n$ . Assume that  $T[r]$  is the latest arrived character of the text and that  $r \in \text{occ}(P, T)$ . Assume that for each non-periodic  $P' \in \Pi$  we store two latest witnesses in  $\text{occ}(P', T)$ , and for each periodic  $P' \in \Pi$  we store the two latest streaks of  $\Delta(P')$  and compact representations of the witnesses in them. In addition, assume that we store the set of all atomic strings  $A$  such that  $(r - 1)$  is a witness for  $\text{occ}(A, T)$ . One can decide whether  $r$  is a witness for  $P$  in  $\mathcal{O}(nd^4 \text{polylog } n)$  time and  $\mathcal{O}(d^3 \text{polylog } n)$  (extra) space with high probability.*

*Proof.* Let  $P = A[1 \dots \min\{2^k, |A|\}]$ , where  $A$  is an atomic string. Consider two cases:  $k = 0$  and  $k \geq 1$ .

Case 1:  $k = 0$ . If  $k = 0$ , let  $A_{i_1}, A_{i_2}, \dots, A_{i_j}$  be the atomic strings such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A_{i_j}$ ,  $1 \leq j' \leq j$ , to the starting point of the transition labelled by  $A$ . The position  $r$  is a witness for  $P$  iff for some  $1 \leq j' \leq j$ , the position  $(r - 1)$  is a witness for  $A_{i_{j'}}$ . We can decide whether this holds in  $\mathcal{O}(d)$  time.

Case 2:  $k \geq 1$ . If  $k \geq 1$ , the position  $r$  is a witness for  $P$  iff  $r - 2^{k-1}$  is a witness for  $P[1 \dots 2^{k-1}]$ . For brevity, denote  $r' = r - 2^{k-1}$ ,  $P' = P[1 \dots 2^{k-1}]$ , and  $\rho = |\Delta(P')|$ . If  $P'$  is non-periodic and  $r' \in \text{occ}(P', T)$ , the algorithm stores it explicitly by Observation 1.11. Otherwise, by Observation 1.12,  $r'$  belongs to one of the two latest streaks of  $\Delta(P')$ , let it be a fragment  $S = T[i \dots i + \ell \cdot \rho - 1]$ . Suppose that  $r'$  is a witness for  $P'$ . Let us first explain the solution for the regular expression membership problem, and then we will show how to modify it for the regular expression pattern matching problem.

In the membership problem, if  $r'$  is a witness for  $P'$ , then  $T[1 \dots r']P[2^{k-1}+1 \dots]$  is a partial occurrence of  $R$  and there is a partition of  $T[1 \dots r'] = T[\ell_1 \dots r_1]T[\ell_2 \dots r_2] \dots T[\ell_m \dots r_m]$ , where each  $T[\ell_{m'} \dots r_{m'}]$ ,  $1 \leq m' < m$ , is an atomic string, and  $T[\ell_m \dots r_m] = P'$ . Let  $T[\ell_{m'} \dots r_{m'}]$  be the fragment containing  $i$ . We consider two subcases:  $r_{m'} - i + 1 > 2\rho$  and  $r_{m'} - i + 1 \leq 2\rho$ .

Case 2(a):  $r_{m'} - i + 1 > 2\rho$ . We claim that in this subcase  $r_{m'} - i + 1$  equals the overlap  $\ell$  of  $T[\ell_{m'} \dots r_{m'}]$  and  $W = (\Delta(P'))^\infty$ . By definition,  $r_{m'} - i + 1 \leq \ell$ . Suppose that  $r_{m'} - i + 1 < \ell$ . If  $\ell - (r_{m'} - i + 1)$  is a multiple of  $\rho$ , then we obtain that  $T[r_{m'} - \rho \dots r_{m'} - 1] = \Delta(P')$ , a contradiction with the definition of  $S$ . If  $\ell - (r_{m'} - i + 1)$  is not a multiple of  $\rho$ , then there is an occurrence of  $\Delta(P')$  in the prefix  $(\Delta(P'))^2$  of  $S$  that does not end at positions  $\rho$  or  $2\rho$ . By Corollary 1.20, we obtain a contradiction. Therefore,  $r_{m'} - i + 1 = \ell$  and the compact representation of witnesses in  $S$  stores  $r_{m'} \in \text{occ}(T[\ell_{m'} \dots r_{m'}], T)$ .

If  $m' = m$  or  $r_m - r_{m'} \leq 8\rho$ , then we are done: if  $r'$  is a witness, it must be stored explicitly, and we can check whether it is the case in  $\mathcal{O}(d \log^2 n)$  time. Otherwise, we use the following claim:

**Claim 1.26.** *There is a sequence  $m' = m_0 < m_1 < m_2 < \dots < m_q = m$  such that each  $T[\ell_{m_{q'}} \dots r_{m_{q'}}]$ ,  $1 \leq q' < q$ , is either anchored by an anchor  $a \in \mathcal{A}^*(S)$ , or has length at least  $2\rho$ , and for each  $1 \leq q' \leq q$ ,  $\ell_{m_{q'}} - r_{m_{q'-1}} \leq 10\rho$ .*

*Proof.* The sequence is built as follows. Let  $m_0 = m'$  and  $m_{q'}$  be the latest index added to the sequence. If there is an index  $m''$  such that  $T[\ell_{m''} \dots r_{m''}]$  has length at least  $2\rho$

or  $m'' = m$  and  $\ell_{m''} - r_{q'} \leq 10\rho$ , then set  $m_{q'+1} = m''$  and continue. Otherwise, let  $m''$  be the smallest index such that  $\ell_{m''} - r_{m_{q'}} \geq 8\rho$ . Note that we also have  $\ell_{m''} - r_{m_{q'}} \leq 10\rho$  (otherwise, the length of  $T[\ell_{m''-1} \dots r_{m''-1}]$  would have been larger than  $2\rho$ ). By Lemma 1.15, there is  $m_{q'} < \tilde{m} \leq m''$  such that  $T[\ell_{\tilde{m}} \dots r_{\tilde{m}}]$  is anchored by an anchor  $a \in \mathcal{A}^*(S) \cap [r_{m''} - 4\rho + 1, r_{m''}]$ . We set  $m_{q'+1} = \tilde{m}$  and continue.  $\square$

Let  $v'$  be the node in  $G(P)$  corresponding to  $(T[\ell_{m'} \dots r_{m'}], r_{m'} - i + 1 \pmod{\rho})$ , and  $v$  be the node corresponding to  $(P', r_m - i + 1 \pmod{\rho})$ . We have that  $j'$  is a witness iff there exists the sequence  $m' = m_0 < m_1 < m_2 < \dots < m_q = m$  as above iff there is a walk from  $v$  to  $v'$  of length  $|T[r_{m'} + 1 \dots r_m]| \leq n$ , which we can check in  $\mathcal{O}(nd^4 \text{polylog } n)$  time and  $\mathcal{O}(d^3 \text{polylog } n)$  extra space with high probability via Corollary 1.23 (we must check whether this condition is verified for each of the  $\mathcal{O}(d \log^2 n)$  witnesses stored in the compact representation of  $S$ ).

Case 2(b):  $r_{m'} - i + 1 \leq 2\rho$ . Consider now the second subcase. If  $r_m \leq 12\rho$ , then we are done: if  $r'$  is a witness, it must belong to the compact representation of witnesses in  $S$ , which can be verified in  $\mathcal{O}(d \log^2 n)$  time. Otherwise,  $r_m - r_{m'} \geq 8\rho$ . By Lemma 1.17(a) there is  $p$ ,  $m' < p \leq m$ , such that  $T[\ell_p \dots r_p]$  is anchored by an anchor  $\mathcal{A}^*(F) \cap [i, i + 4\rho - 1]$ , and therefore  $T[\ell_p \dots r_p]$  is stored in the compact representation of witnesses in  $S$ . Analogously to Case 2(a), we can show equivalence of the following conditions:  $r'$  is a witness; there is a walk in  $G(P)$  from the node corresponding to  $(T[\ell_p \dots r_p], r_p - i + 1 \pmod{\rho})$  to the node corresponding to  $(T[\ell_m \dots r_m], r_m - i + 1 \pmod{\rho})$  of length  $|T[r_p + 1 \dots r_{m-1}]|$ . We can therefore decide whether  $r'$  is a witness via Corollary 1.23 in  $\mathcal{O}(nd^4 \text{polylog } n)$  time and  $\mathcal{O}(d^3 \text{polylog } n)$  space with high probability.

We now explain how to modify the argument so that it can be used for regular expression pattern matching. Note that in pattern matching, if  $r'$  is a witness for  $P'$ , then there is some position  $\ell'$ ,  $1 \leq \ell' \leq r'$  such that  $r' \in \text{occ}(P', T)$  is a witness. The position  $\ell'$  can be inside the streak  $S$ , i.e.  $m'$  can be undefined, making it impossible to apply the argument above. However, we can easily check if this is the case using the integer  $\mu(P')$  we computed during the preprocessing step: if  $\mu(P') \cdot \rho \geq (r' - i + 1)$ , then  $r'$  is a witness and we are done, and otherwise  $\ell' \leq i$  (if  $r'$  is a witness),  $m'$  is defined, and we can apply the argument above.  $\square$

**Theorem 1.27.** *Given a streaming text  $T$  of length  $n$  and a regular expression  $R$  of size  $d$ . There is a randomised algorithm that solves the membership and the pattern matching problems for  $T$  and  $R$  in  $\mathcal{O}(d^3 \text{polylog } n)$  space and  $\mathcal{O}(nd^5 \text{polylog } n)$  time per character of the text. The algorithm succeeds with high probability.*

*Proof.* If  $d \geq n$ , we can use Claim 1.6. Below we assume that  $d < n$ . Recall that we do not account for the time used during the preprocessing step (but one can note that it is polynomial in  $d$  and the total length of the atomic strings of  $R$ ). The information computed during this step, including the graphs  $G(P)$  for each  $P \in \Pi$ , takes  $\mathcal{O}(d^3 \log^5 n)$  space.

During the main step, we use Lemma 1.25 to maintain the compact representations of the streaks of  $\Delta(P)$  for each  $P \in \Pi$ , and to decide, eventually, whether  $T$  matches the regular expression  $R$ . Whenever an instance of the pattern matching algorithm detects an occurrence of  $\Delta(P)$ , we decide in constant time whether this occurrence extends the latest streak of  $\Delta(P)$  or starts a new one. If the number of streaks becomes equal to three, we discard the oldest streak.

When an instance of the pattern matching algorithm detects  $r \in \text{occ}(P, T)$  for some  $P \in \Pi$ , we must decide whether  $r$  is a witness and whether we must store it in the compact representation of the streaks containing  $r$ . We apply Lemma 1.25 to decide whether  $r$  is a witness in  $\mathcal{O}(nd^4 \text{polylog } n)$  total time and  $\mathcal{O}(d^3 \text{polylog } n)$  space and then in  $\mathcal{O}(d \log n)$  time whether  $r$  must be added to the compact representations of the streaks containing  $r$ . Note that a position  $r$  can belong to  $\text{occ}(P, T)$  for  $\mathcal{O}(d \log n)$  canonical prefixes  $P \in \Pi$ , and therefore in the worst case we spend  $\mathcal{O}(nd^5 \text{polylog } n)$  to process  $r$ . The compact representations of the streaks take  $\mathcal{O}(d^2 \log^3 n)$  space.

Recall that  $\mathcal{F}$  contains all atomic strings  $A$  such that there is an  $\varepsilon$ -transitions path from the endpoint of the transition labelled by  $A$  to the final state of  $T_C(R)$ . In the regular expression pattern matching problem, we report all positions  $r$  such that  $r$  is a witness in  $\text{occ}(A, T)$  for some  $A \in \mathcal{F}$ . In the regular expression membership problem,  $T \in L(R)$  if  $n$  is a witness for  $\text{occ}(A, T)$  for some  $A \in \mathcal{F}$ .  $\square$

## 5 Proof of Theorem 1.13

An important tool in our proof is the framework that allows computing output of a circuit time- and space-efficiently. Before we describe the framework in detail, we provide some notation following [237]. A circuit is a directed acyclic graph with nodes of in-degree 0 or 2. Degree-0 nodes are called inputs and degree-2 nodes are gates. In our application, every node corresponds to a vector from  $\mathbb{Z}_p^t$  (i.e. a vector of length  $t$  with values in  $\mathbb{Z}_p$ ) indexed from 0 to  $t - 1$ , for some values of  $p$  and  $t$  that will be specified later. A vector in  $\mathbb{Z}_p^t$  is a singleton if it has at most one non-zero entry.

There are two types of gates:  $\boxplus$  and  $\boxtimes$  that denote respectively the pointwise addition and vector convolution binary gates, that is  $(a \boxplus b)[i] = a[i] + b[i]$  and  $(a \boxtimes b)[i] = \sum_{j=0}^i a[j] \cdot b[i - j]$ . Every gate corresponds to the result of its underlying operation applied to its incoming nodes. We say that a convolution gate with input  $a, b \in \mathbb{Z}_p^t$  does not overflow, if for all  $i \geq t$ ,  $(a \boxtimes b)[i] = 0$ . An element  $\omega$  is a  $t$ -th root of unity in  $\mathbb{Z}_p$  iff  $\omega^t \equiv 1 \pmod{p}$  but  $\omega^s \not\equiv 1 \pmod{p}$  for all  $0 < s < t$ .

With the definitions at hand, we are ready to state the technique introduced by Lokshantov and Nederlof [145] for complex numbers and its modular variant discussed by Bringmann [237]:

**Theorem 1.28** (cf. [237, Theorem 4.2]). *Let  $p$  be a prime,  $t \geq 1$ , and suppose that  $\mathbb{Z}_p$  contains a  $t$ -th root of the unity,  $\omega$ . Let  $C$  be a circuit over  $(\mathbb{Z}_p^t, \boxplus, \boxtimes)$  which takes as an input only singleton constants and outputs a vector  $\text{out}(C) \in \mathbb{Z}_p^t$ . Suppose that no convolution gate overflows. Then given  $p, t, \omega$ , and  $0 \leq x < t$  we can compute  $\text{out}(C)[x]$  in time  $\mathcal{O}(|C|t \text{polylog } p)$  and space  $\mathcal{O}(|C| \log p)$ .*

For Bringmann's framework to be efficient, one must provide a method to choose  $p$  and  $\omega$ . Bringmann [237] showed two different methods, one requiring the Extended Riemann Hypothesis and the other one resulting in an additional  $t^\varepsilon$  factor in both time and space [237, Lemma 4.4]. Below we show that one can achieve the bounds of the former method unconditionally.

## 5.1 Finding Primes

The goal of this section is to prove the following theorem:

**Theorem 1.29.** *There is a procedure that, given  $y$ , finds in  $\mathcal{O}(y \text{ polylog } y)$  arithmetic operations and  $\mathcal{O}(\log y)$  space a prime  $p$  and  $\omega$  such that for every  $N \leq 2^{\mathcal{O}(y \log y)}$ , with probability at least  $1/2$  the following holds:*

1.  $\omega$  is a  $t$ -th primitive root of unity in  $\mathbb{Z}_p$ , for some  $t$  satisfying  $y \leq t = \mathcal{O}(y \text{ polylog } y)$ ;
2.  $p = \mathcal{O}(y^2 \text{ polylog } y)$ ;
3.  $p \nmid N$ .

Let  $B$  be a constant to be determined later. To compute numbers  $p$  and  $\omega$  we run the following procedure:

1. Set  $x$  to be the smallest number such that  $\frac{1}{2}\sqrt{x} \log^{-B} x \geq y$  (using binary search);
2. Choose a random  $q \in [\frac{1}{2}, 1] \cdot \sqrt{x} \log^{-B} x$ ;
3. Find a prime  $p$  such that  $p \leq x$  and  $p \equiv 1 \pmod{q}$  (by guessing candidate  $p$  and checking all numbers up to  $\sqrt{p}$  if they divide  $p$  or not);
4. Find a generator  $g$  of  $\mathbb{Z}_p^*$  (by guessing candidate  $g$  and checking if  $g^{\frac{p-1}{p'}} \not\equiv 1 \pmod{p}$  for all prime divisors  $p'$  of  $p-1$ );
5. Set  $t = q$  and  $\omega = g^{\frac{p-1}{t}}$ .

Clearly,  $t \mid p-1$  and  $\omega = g^{\frac{p-1}{t}}$  is well-defined. As  $g$  is a generator, we have that  $\omega$  is a  $t$ -th primitive root of unity in  $\mathbb{Z}_p^*$ . By the choice of  $x$  and  $p$ , we have  $p \leq x = \mathcal{O}(y^2 \text{ polylog } y)$  and hence  $y \leq q = t = \mathcal{O}(y \text{ polylog } y)$ . In the following lemma we show that with probability at least  $7/8$ , there are many primes  $p$  satisfying  $p \leq x$  and  $p \equiv 1 \pmod{q}$  and hence we can efficiently find one. Finally, we show how to find the generator  $g$  efficiently. Let  $\pi(x; q, a) = |\{p \leq x : p \equiv a \pmod{q}\}|$  and  $\phi(n) = |\{1 \leq a \leq n : \gcd(a, n) = 1\}|$ .

**Lemma 1.30.** *Let  $q$  be chosen uniformly at random from  $[\frac{1}{2}, 1] \cdot \sqrt{x} \log^{-B} x$ . With probability at least  $7/8$  we have  $\pi(x; q, 1) = \Omega(\sqrt{x}(\log x)^{B-1})$ .*

Before we prove the lemma, we remind some number-theory notation and facts related to counting primes. The reader familiar with this area can skip this part. We first remind the definitions of von Mangoldt function  $\Lambda(n)$  and Chebyshev functions  $\psi$  and  $\vartheta$ :

$$\psi(x; q, a) = \sum_{\substack{n \leq x \\ n \equiv a \pmod{q}}} \Lambda(n), \quad \text{where } \Lambda(n) = \begin{cases} \log p & \text{if } n = p^k \\ & \text{for some prime } p \text{ and } k \in \mathbb{Z}_+, \\ 0 & \text{otherwise.} \end{cases}$$

$$\vartheta(x; q, a) = \sum_{\substack{p \leq x \\ p \equiv a \pmod{q}}} \log p, \quad \text{where the summation is over prime numbers } p.$$

By skipping the last two arguments we denote  $\vartheta(x) = \sum_{0 \leq a < q} \vartheta(x; q, a)$ , analogous for  $\psi(x)$ .



**Fact 1.31** (By definition).  $\vartheta(x; q, a) \leq \pi(x; q, a) \cdot \log x$ .

**Fact 1.32** (cf. [6, Theorem 13]).  $\psi(x; q, a) \leq \vartheta(x; q, a) + \mathcal{O}(\sqrt{x})$ .

*Proof.* Theorem 13 of [6] states that  $\psi(x) - \vartheta(x) = \mathcal{O}(\sqrt{x})$ , so for completeness we show an adaptation of this property to numbers forming an arithmetic progression.

$$\begin{aligned} \psi(x; q, a) - \vartheta(x; q, a) &= \sum_{\substack{n \leq x \\ n \equiv a \pmod q \\ n = p^k, k \geq 1}} \log p - \sum_{\substack{p \leq x \\ p \equiv a \pmod q}} \log p \\ &\leq \sum_{\substack{n \leq x \\ n = p^2}} \log p + \sum_{\substack{n \leq x \\ n = p^k, k \geq 3}} \log p \\ &\leq \vartheta(\sqrt{x}) + \mathcal{O}(\sqrt[3]{x} \log^2 x) = \mathcal{O}(\sqrt{x}) \end{aligned}$$

as  $\sum_{\substack{n \leq x \\ n = p^2}} \log p = \sum_{p \leq \sqrt{x}} \log p = \vartheta(\sqrt{x})$  and finally  $\vartheta(x) = x + o(x)$  by [6, (2.29)].  $\square$

**Theorem 1.33** (Bombieri–Vinogradov theorem [13]). *For every  $A > 0$  there exists  $B = B(A) > 0$  such that for every  $x$ :*

$$\sum_{q \leq \sqrt{x}(\log x)^{-B}} \max_{y \leq x} \max_{\gcd(a, q) = 1} \left| \psi(y; q, a) - \frac{y}{\phi(q)} \right| = \mathcal{O}(x \log^{-A} x).$$

With all the notation at hand we are ready to prove Lemma 1.30.

*Proof of Lemma 1.30.* Let  $\mathcal{R} = [\frac{1}{2}, 1] \cdot \sqrt{x} \log^{-B} x$  be the range from which we draw  $q$ . By choosing  $y = x$  and  $a = 1$  and summing only over  $q \in \mathcal{R}$  we lower bound the left-hand side of Bombieri–Vinogradov theorem obtaining that, for every  $A > 0$  there exists  $B = B(A) > 0$  such that

$$\sum_{q \in \mathcal{R}} \left| \psi(x; q, 1) - \frac{x}{\phi(q)} \right| = \mathcal{O}(x \log^{-A} x). \quad (1.1)$$

Similarly to Markov's inequality, for  $q$  chosen uniformly at random from  $\mathcal{R}$  we have with probability at least  $7/8$ :

$$\left| \psi(x; q, 1) - \frac{x}{\phi(q)} \right| = \mathcal{O}(\sqrt{x}(\log x)^{-A+B}) \quad (1.2)$$

Indeed, there can be at most  $|\mathcal{R}|/8$  numbers  $q$  in  $\mathcal{R}$  such that  $\psi(x; q, 1) \geq \frac{8 \cdot \Omega(x \log^{-A} x)}{|\mathcal{R}|} = \Omega(\sqrt{x}(\log x)^{-A+B})$  in order not to exceed the right-hand side of (1.1). Let  $B$  be the constant from Theorem 1.33 for  $A = 1$ . Without loss of generality, we assume  $B \geq 3 > A = 1$ . Now we rewrite (1.2) using properties of  $\phi(n)$ ,  $\vartheta(n)$  and  $\psi(n)$  and obtain:

$$\pi(x; q, 1) = \Omega(\sqrt{x}(\log x)^{B-1})$$

In more detail:

$$\begin{aligned}
 \frac{x}{\phi(q)} - \psi(x; q, 1) &= \mathcal{O}(\sqrt{x}(\log x)^{-A+B}) && \text{from (1.2)} \\
 \frac{x}{q} &\leq \vartheta(x; q, 1) + \mathcal{O}(\sqrt{x}) + \mathcal{O}(\sqrt{x}(\log x)^{-A+B}) && \text{by Fact 1.32 and } \phi(q) \leq q \\
 \frac{x}{q} &\leq \pi(x; q, 1) \cdot \log x + \mathcal{O}(\sqrt{x}(\log x)^{-A+B}) && \text{by Fact 1.31 and } B > A \\
 \pi(x; q, 1) &\geq \sqrt{x}(\log x)^{B-1} - \mathcal{O}(\sqrt{x}(\log x)^{-A+B-1}) && \text{as } q \in \mathcal{R} \\
 \pi(x; q, 1) &= \Omega(\sqrt{x}(\log x)^{B-1}) && \text{as } A = 1
 \end{aligned}$$

□

**Lemma 1.34.** *Suppose  $\pi(x; q, 1) = \Omega(\sqrt{x}(\log x)^{B-1})$ . With probability at least  $7/8$ , in  $\mathcal{O}(\sqrt{x} \log x)$  arithmetic operations and  $\mathcal{O}(\log x)$  space we can find such a prime  $p \leq x$  such that  $p \equiv 1 \pmod q$ .*

*Proof.* Clearly, we can check if a number  $n$  is prime by iterating through all numbers  $2, 3, \dots, \sqrt{n}$  and checking if they are a divisor of  $n$ . Let  $\mathcal{Q} = \{n \leq x : n \equiv 1 \pmod q\}$ . Observe that the probability that a number chosen uniformly at random from  $\mathcal{Q}$  is prime is at least:

$$\frac{\pi(x; q, 1)}{|\mathcal{Q}|} = \frac{\pi(x; q, 1)}{x/q} = \Omega\left(\frac{\sqrt{x}(\log x)^{B-1} \sqrt{x} \log^{-B} x}{x}\right) = \Omega\left(\frac{1}{\log x}\right)$$

Hence by checking  $\Theta(\log x)$  numbers from  $\mathcal{Q}$  we find a prime with probability at least  $7/8$ . □

**Lemma 1.35.** *With probability at least  $7/8$ , we can find a generator  $g$  of  $\mathbb{Z}_p^*$  in  $\mathcal{O}(\sqrt{p})$  arithmetic operations and  $\mathcal{O}(\log p)$  space.*

*Proof.* First, we generate the set of all divisors of  $p-1$  in  $\mathcal{O}(\sqrt{p})$  time by iterating through  $2, 3, \dots, \sqrt{p-1}$  and checking if they are a divisor of  $p-1$ . By using an auxiliary accumulator we can restrict only to prime divisors, we call this set  $\mathcal{D}$ . Now we can check if a number  $g$  is a generator of  $\mathbb{Z}_p^*$  by checking if for every  $p' \in \mathcal{D}$ , a prime divisor of  $p-1$ , we have  $g^{(p-1)/p'} \not\equiv 1 \pmod p$ . Using exponentiating by squaring, this runs in total  $\mathcal{O}(\text{polylog } p)$  time.

The probability of a random  $g \in \{0, \dots, p-2\}$  to be a generator is  $\phi(p-1)/(p-1) = \Omega(1/\log \log p)$ , as  $\phi(n) = \Omega(n/\log \log n)$  [6, Theorem 15]. Hence by checking  $\Theta(\log \log p)$  numbers from  $\mathbb{Z}_p^*$  we find a generator with probability at least  $7/8$ . □

Finally, as  $x \geq y^2$  and  $\pi(x; q, 1) = \Omega(\sqrt{x}(\log x)^{B-1})$  and  $B \geq 3$  we have  $\pi(x; q, 1) \geq y \log^2 y \geq 8 \log N$ , as  $N \leq 2^{\mathcal{O}(y \log y)}$ . Because  $N$  has at most  $\log N$  prime divisors, the probability that the chosen prime  $p$  is one of them is at most  $1/8$ . Summing up, there are four events due to which our algorithm can fail:

1. The number  $q$  does not satisfy  $\pi(x; q, 1) = \Omega(\sqrt{x}(\log x)^{B-1})$ ;
2. We did not find  $p$  in the planned number of iterations;

3. The chosen  $p$  divides  $N$ ;
4. We did not find  $g$  in the planned number of iterations.

We note that we do not have access to the value of  $N$  during the algorithm, so we cannot spot immediately that the chosen  $p$  is wrong. When we fail to find  $g$  or  $p$  in the planned number of iterations we terminate. However, if  $q$  is chosen wrongly, we cannot detect it immediately, but then the subsequent steps (choosing  $p$  or  $g$ ) will have a larger probability of failure. To conclude, the overall probability of a failure is at most  $1/2$  and the running time of the whole procedure is  $\mathcal{O}(\sqrt{x} \text{polylog}(x)) = \mathcal{O}(y \text{polylog } y)$ . This concludes the proof of Theorem 1.29.

## 5.2 Walks in a Weighted Graph

We can finally prove Theorem 1.13. We first describe an algorithm that uses significantly much more time and space than desired, and then improve it. We compute arrays  $C_k$  for  $k \in \{0, \dots, \lceil \log x \rceil\}$  indexed by nodes  $u, v \in V(G)$ , where  $C_k[u, v]$  is a bit-vector of length  $x + 1$  such that:

1.  $C_k[u, v][d] = 1$  implies that there exists a walk of weight  $d$  from  $u$  to  $v$ ;
2. For every  $d \leq 2^k$ , if there exists a walk of weight  $d$  from  $u$  to  $v$  in  $G$ , we have  $C_k[u, v][d] = 1$ .

In other words,  $C_k$  contains the information about all walks of weight at most  $2^k$  in  $G$  and possibly some other walks of weight at most  $x$ . We initialize the array  $C_0$  in the following way:  $\forall_{u \in V(G)} C_0[u, u][0] = 1$  and  $C_0[u, v][d] = 1$  if there is an edge from  $u$  to  $v$  of weight  $0 \leq d \leq x$ . If there are 0-weight edges in  $G$ , we first need to compute their transitive closure in  $G$  in  $\mathcal{O}(|V(G)|^3)$  time and mark in  $C_0$  all walks of total weight 0 or 1 in  $G$ . We define (OR, CONVOLUTION <sub>$x$</sub> )-product of matrices consisting of bit-vectors, truncated to the first  $x + 1$  positions:

$$\forall_{\substack{u, v \in V(G) \\ d \in \{0, \dots, x\}}} (A \odot B)[u, v][d] := \bigvee_{\substack{w \in V(G) \\ i \in \{0, \dots, d\}}} A[u, w][i] \wedge B[w, v][d - i]$$

Now, we compute the consecutive arrays  $C_k$  as follows by repeatedly applying the (OR, CONVOLUTION <sub>$x$</sub> )-product:

$$C_{k+1} := C_k \odot C_0 \odot C_k$$

Both invariants for the array  $C_k$  follow by inductive reasoning, as every walk of weight  $d$  can be split into three parts of weights  $d_1, d_2, d_3$  where  $d_1, d_3 \leq d/2$  and the middle part consists of a single edge (recall that for each edge  $(u, v)$  of weight  $0 \leq d \leq x$  we have  $C_0[u, v][d] = 1$ ). Then, for the given nodes  $v_1, v_2$  we can return the entry  $C_{\lceil \log x \rceil}[v_1, v_2][x]$ .

This approach runs in  $\mathcal{O}(|V(G)|^2 x)$  space and  $\mathcal{O}(|V(G)|^3 + |V(G)|^2 x \text{polylog } x)$  time when we use the fast Fourier transform at every step. Observe that this complexity matches the time and space bounds stated in Theorem 1.13 for the case when  $x = \mathcal{O}(|V(G)|)$ . Hence, we focus on the case when  $x = \Omega(|V(G)|)$ .

**Saving space with circuits.** To save both time and space, we will use circuits and the framework of Theorem 1.28. In order to use this framework, we need to modify our algorithm in various aspects. First, in  $\mathcal{O}(x \text{ polylog } x)$  time we find the appropriate values of  $p, t, \omega$  using Theorem 1.29 from Section 5.1 for  $y = \Theta(x)$  that will be defined precisely later. Then  $t = \mathcal{O}(x \text{ polylog } x)$ . Instead of bit-vectors as entries of the array  $C_k$ , we operate on vectors from  $\mathbb{Z}_p^t$  over  $(\mathbb{Z}_p, +, \cdot)$ . In other words we use  $\boxplus$  (addition in  $\mathbb{Z}_p^t$ ) instead of Boolean OR and  $\boxtimes$  (the standard  $(+, \cdot)$ -convolution modulo  $p$ ) instead of the Boolean  $(\vee, \wedge)$ -convolution. Then the  $\odot$  product between  $A \odot B$  becomes  $(A \odot B)[u, v] = \boxplus_{w \in V(G)} A[u, w] \boxtimes B[w, v]$ . With  $\odot$  defined this way,  $C_k[u, v][d]$  counts modulo  $p$  walks from  $u$  to  $v$  of weight  $d$ , possibly counting one walk more than once — we analyse these values in detail at the end of the proof.

Now we describe the construction of the circuit. To simplify the presentation, we work with multi-ary addition gates  $\boxplus^*$  which can be replaced with binary gates  $\boxplus$  at the expense of doubling the total size of the circuit.

1. For every  $k \in \{0, \dots, \lceil \log x \rceil\}$  and  $u, v \in V(G)$  we create a  $\boxplus^*$  gate  $C_k[u, v]$ ;
2. For every node  $v \in V(G)$  we create a singleton constant  $V_v$  with only the 0-th entry set to 1, connected to the  $\boxplus^*$  gate  $C_0[v, v]$ ;
3. For every edge  $(u, v, d) \in E(G)$  from node  $u$  to  $v$  of weight  $d$ , we create a singleton constant  $E_{u,v,d}$  with only the  $d$ -th entry set to 1, connected to the  $\boxplus^*$  gate  $C_0[u, v]$ ;
4. As  $(A \odot B)[u, v] = \boxplus_{w \in V(G)}^* A[u, w] \boxtimes B[w, v]$ , we can implement every product  $X = A \odot B$  with  $|V(G)|^3$  gates  $X^w[u, v] := A[u, w] \boxtimes B[w, v]$  and  $|V(G)|^2$  gates  $X[u, v] := \boxplus_{w \in V(G)}^* X^w[u, v]$ . For every  $k > 0$ , it holds  $C_k = C_{k-1} \odot C_0 \odot C_{k-1}$ , so we need an intermediate product  $C'_k := C_{k-1} \odot C_0$  and then  $C_k := C'_k \odot C_{k-1}$ .

The above construction gives a circuit on  $\mathcal{O}(|E(G)| + |V(G)|^3 \log x)$  gates with singleton constants, out of which we need to output if  $C_{\lceil \log x \rceil}[v_1, v_2][x] > 0$ . However, we still cannot use the framework from Theorem 1.28, as we cannot guarantee that there are no convolution gate overflows. Indeed, if there are edges of weight almost  $x$ , we would obtain walks of weight  $x^2$ . In the following paragraph we show a refined construction in which we have more control on the maximum weight of walks considered in the  $k$ -th step of the algorithm.

**Refined construction.** Let  $\varepsilon$  be a value to be determined precisely later. Instead of the arrays  $C_k$ , we will compute arrays  $D_k$  that, informally, describe all walks of total weight at most  $(1+\varepsilon)^k$ , some walks of weight  $d \leq (1+\varepsilon)^k \cdot (1+\varepsilon)^{2k \cdot \log(1+\varepsilon)}$  and no longer walks. As we operate on values modulo  $p$ , let  $D'_k[u, v][d]$  be the value of  $D_k[u, v][d]$  if computed exactly, without taking modulo  $p$  at every step. Formally, for every  $k = \{0, \dots, \lceil \log_{1+\varepsilon} x \rceil\}$  we have:

1.  $D'_k[u, v][d] > 0$  implies that there exists a walk of weight  $d$  from  $u$  to  $v$ ;
2. For each  $d \leq (1+\varepsilon)^k$ , if there exists a walk of weight  $d$  from  $u$  to  $v$  in  $G$ , we have  $D'_k[u, v][d] > 0$ ;
3.  $D_k[u, v][d] = 0$  for all  $d > (1+\varepsilon)^k \cdot (1+\varepsilon)^{2k \cdot \log(1+\varepsilon)}$ .

The array  $D_0$  stores all walks of total weight at most 1, that is  $D_0[u, v][d] = 1$  iff  $d \in \{0, 1\}$  and there is a walk from  $u$  to  $v$  of total weight  $d$ . It can be computed in  $\mathcal{O}(|V(G)|^3)$  time as  $C_0$ , by first computing all pairs of nodes connected by a walk of 0-weight edges. Now we show how to obtain the array  $D_k$ . Again, every walk of weight  $d$  can be cut into three parts of total weights  $d_1, d_2, d_3$  where  $d_1, d_3 \leq d/2$  and the middle part consist of a single edge. We need to control the total weight of the walk, so we will iterate over all possible base- $(1 + \varepsilon)$  logarithms of weights of the three parts  $k_1, k_2, k_3$ . For all possible values  $d_1, d_2, d_3$  such that  $d_1 + d_2 + d_3 \leq (1 + \varepsilon)^k$ , we process the triple  $k_1, k_2, k_3$  where  $\forall_{i \in \{1, 2, 3\}} (1 + \varepsilon)^{k_i - 1} < d_i \leq (1 + \varepsilon)^{k_i}$ . Then, from the definition of arrays  $D_k$ , every walk of weight  $d_i$  will be included in  $D_{k_i}$ . For single edges of particular weight, let  $B_k$  describe all pairs of nodes connected by an edge of weight at most  $(1 + \varepsilon)^k$ :  $B_k[u, v][d] = 1$  iff  $d \leq (1 + \varepsilon)^k$  and there is an edge of weight  $d$  from  $u$  to  $v$ . Note that  $B_k = B_{k-1} \boxplus F_k$  where  $F_k$  describes all edges of weight from  $((1 + \varepsilon)^{k-1}, (1 + \varepsilon)^k]$ . We restrict only to triples  $k_1, k_2, k_3$  satisfying both:

$$(a) \quad (1 + \varepsilon)^{k_1 - 1} + (1 + \varepsilon)^{k_2 - 1} + (1 + \varepsilon)^{k_3 - 1} \leq (1 + \varepsilon)^k \quad (1.3)$$

$$(b) \quad 2 \cdot (1 + \varepsilon)^{\max\{k_1, k_3\} - 1} \leq (1 + \varepsilon)^k \quad (1.4)$$

and call such triples  $k$ -good. Then we compute  $D_k$  in the following way:

$$D_k := \boxplus_{k\text{-good } k_1, k_2, k_3}^* D_{k_1} \odot B_{k_2} \odot D_{k_3} \quad (1.5)$$

Now we show that all the invariants about  $D_k$  are satisfied. Clearly there are no false-positive entries in the array. We never miss a walk of weight at most  $(1 + \varepsilon)^k$ , as the condition (a) filters out the triples  $k_i$  contributing only the walks of total weight larger than  $(1 + \varepsilon)^k$ . The condition (b) guarantees that the first and third part of the walk have weight at most  $\frac{1}{2}(1 + \varepsilon)^k$ . In the following lemma we show that we also never construct walks of too large weight.

**Lemma 1.36.** *For every  $k$  and every  $k$ -good triple  $k_1, k_2, k_3$ , the largest weight of a walk in  $D_{k_1} \odot B_{k_2} \odot D_{k_3}$  is at most  $(1 + \varepsilon)^k \cdot (1 + \varepsilon)^{2k \cdot \log(1 + \varepsilon)}$ .*

*Proof.* Induction on  $k$ . Without loss of generality assume  $k_1 \geq k_3$  and then the walks in  $D_{k_1} \odot B_{k_2} \odot D_{k_3}$  have total weight at most:

$$\begin{aligned} &\leq (1 + \varepsilon)^{k_1} \cdot (1 + \varepsilon)^{2k_1 \cdot \log(1 + \varepsilon)} + (1 + \varepsilon)^{k_2} + (1 + \varepsilon)^{k_3} \cdot (1 + \varepsilon)^{2k_3 \cdot \log(1 + \varepsilon)} \\ &\leq [(1 + \varepsilon)^{k_1} + (1 + \varepsilon)^{k_2} + (1 + \varepsilon)^{k_3}] \cdot (1 + \varepsilon)^{2k_1 \cdot \log(1 + \varepsilon)} \\ &\leq (1 + \varepsilon)^{k+1} \cdot (1 + \varepsilon)^{2k_1 \cdot \log(1 + \varepsilon)} \quad (\text{from the condition (a)}) \end{aligned} \quad (1.6)$$

Now we use the condition (b) of a good  $k$ -triple:

$$\begin{aligned} (1 + \varepsilon)^k &\geq 2 \cdot (1 + \varepsilon)^{k_1 - 1} && \text{apply } \log_{1+\varepsilon}(\cdot) \text{ and rearrange} \\ k - k_1 &\geq \log_{1+\varepsilon} 2 - 1 && \text{multiply both sides by } 2 \cdot \log_2(1 + \varepsilon) \\ 2 \cdot \log_2(1 + \varepsilon) \cdot (k - k_1) &\geq 2 \cdot (1 - \log_2(1 + \varepsilon)) > 1 && \text{as } 1 + \varepsilon < \sqrt{2} \\ 2k \cdot \log_2(1 + \varepsilon) &\geq 2k_1 \cdot \log_2(1 + \varepsilon) + 1 \end{aligned}$$

Applying the above inequality to (1.6) concludes the inductive step.  $\square$

Setting  $\varepsilon = 1/\log x$ , as  $\log(1 + \varepsilon) > \varepsilon$  we obtain that there are

$$r = \lceil \log_{1+\varepsilon} x \rceil \leq 1 + \frac{\log x}{\log(1 + \varepsilon)} = \mathcal{O}\left(\frac{\log x}{\varepsilon}\right) = \mathcal{O}(\log^2 x)$$

arrays  $D_k$  to compute. As  $\log(1 + \varepsilon) < 2\varepsilon$ , the largest possible weight is bounded from above by

$$\begin{aligned} (1 + \varepsilon)^r \cdot (1 + \varepsilon)^{2r \cdot \log(1+\varepsilon)} &\leq (1 + \varepsilon)^{r \cdot (4\varepsilon+1)} \leq (1 + \varepsilon)^{(\log_{1+\varepsilon} x + 1) \cdot (4\varepsilon+1)} \\ &\leq x \cdot x^{4\varepsilon} \cdot ((1 + \varepsilon)^\varepsilon)^4 \cdot 2 = \mathcal{O}(x \cdot 2^{4 \cdot \log x \cdot \frac{1}{\log x}}) = \mathcal{O}(x). \end{aligned}$$

Hence, the appropriate choice of  $y = \Theta(x)$  guarantees that no convolution gate overflows.

Now we estimate the size of the constructed circuit. We compute  $r = \mathcal{O}(\log^2 x)$  arrays  $D_k$ . For each of them we process  $\mathcal{O}(r^3)$   $k$ -good triples which perform two  $\odot$  products each. Every  $\odot$  product introduces  $\mathcal{O}(|V(G)|^3)$   $\boxplus$  and  $\boxtimes$  gates. Hence the total number of gates is  $\mathcal{O}(|E(G)| + |V(G)|^3 \cdot \text{polylog } x)$ .

Finally we discuss the properties of the values computed in  $D_r$  and all the intermediate gates. Recall that  $D'_k[u, v][d]$  is the value of  $D_k[u, v][d]$  if computed exactly, without taking modulo  $p$  at every step. For each  $d \leq (1 + \varepsilon)^k$ , every walk between  $u$  and  $v$  of weight  $d$  contributes at least 1 to  $D'_k[u, v][d]$ . Notice that such walk may contribute more than 1, as it can be cut into three parts in many ways, for different triples  $k_1, k_2, k_3$ . As no walks of weight different from  $d$  contribute to  $D'_k[u, v][d]$ , there is a walk from  $u$  and  $v$  of weight  $d$  iff  $D'_k[u, v][d] > 0$ . However, while computing the arrays  $D_k$  we operate in  $\mathbb{Z}_p$ , so we might have false negative error if  $p \mid D'_k[u, v][d]$ . In Theorem 1.29 we include such situations in the probability of failure (we fail if  $p \mid N$ , where  $N = D'_r[u, v][d]$ ), but we need to ensure that  $D'_r[u, v][d]$  never exceeds  $2^{\mathcal{O}(t \log t)}$ .

**Lemma 1.37.** *Suppose we execute the above algorithm up to the  $r$ -th matrix  $D_r$  in  $\mathbb{Z}$ , not applying modulo  $p$  in every gate. Then all the obtained values are bounded by  $2^{\mathcal{O}(x \log x)}$ .*

*Proof.* Recall that  $\varepsilon = \frac{1}{\log x}$ ,  $r = \lceil \log_{1+\varepsilon} x \rceil$ , we operate on vectors with  $t = \mathcal{O}(x \text{ polylog } x)$  entries, and the convolutions do not overflow as the result always fits in the first  $y = \mathcal{O}(x)$  elements of the vectors. Let  $f(k)$  be a monotonous function that upper bounds the values in  $D'_k$  and  $g = |V(G)|$ . Observe that a single product  $A \odot B$  of matrices with entries bounded by respectively  $a_{\max}$  and  $b_{\max}$  results in a matrix with entries bounded by  $g \cdot y \cdot a_{\max} \cdot b_{\max}$ . Hence, as values in  $B_{k_2}$  are 0 or 1, from Equation (1.5) we have the following bound:

$$f(k) \leq \sum_{k\text{-good } k_1, k_2, k_3} f(k_1) \cdot (y \cdot g)^2 \cdot f(k_3) \leq k^3 (y \cdot g)^2 \cdot f^2(k_{\max}) \leq W f^2(k_{\max})$$

where  $k_{\max}$  is the largest possible value of  $k_i$  that can be a part of a  $k$ -good triple and  $W = r^3 \cdot (y \cdot g)^2 = \mathcal{O}(x^5)$  as  $y = \mathcal{O}(x)$  and we consider the case when  $x = \Omega(g)$ . From Equation (1.4) we have:

$$\begin{aligned} 2 \cdot (1 + \varepsilon)^{k_{\max}-1} &< (1 + \varepsilon)^k \\ k_{\max} + \log_{1+\varepsilon} 2 - 1 &< k \end{aligned}$$

As arguments of  $f$  are integers, it would be more convenient to write  $k_{\max} \leq k - c$  where  $c = \lceil \log_{1+\varepsilon} 2 - 1 \rceil \geq \log_{1+\varepsilon} 2 - 1$ . As  $f$  is monotonous, we have:

$$f(k) \leq \begin{cases} W \cdot f^2(k - c), & k \geq c \\ W, & k < c \end{cases}$$

which solves by induction to  $f(k) < W^{2^{\lceil \frac{k}{c} \rceil + 1} - 1}$ . Then, as  $W = \mathcal{O}(x^5)$  we have:

$$f(r) < W^{2^{\lceil \frac{r}{c} \rceil + 1} - 1} < W^{\mathcal{O}(2^{r/c})} < 2^{\mathcal{O}(2^{r/c} \log x)}$$

Finally, we show that  $r/c \leq \log x + \mathcal{O}(1)$ .

$$\begin{aligned} \frac{r}{c} &\leq \frac{\log_{1+\varepsilon} x + 1}{\log_{1+\varepsilon} 2 - 1} = \frac{\frac{\log x}{\log(1+\varepsilon)} + 1}{\frac{\log 2}{\log(1+\varepsilon)} - 1} = \frac{\log x + \log(1+\varepsilon)}{1 - \log(1+\varepsilon)} \leq \frac{\log x + 2\varepsilon}{1 - 2\varepsilon} \quad (\text{as } \log(1+\varepsilon) < 2\varepsilon) \\ &= \frac{\log^2 x + 2}{\log x - 2} = \log x + \mathcal{O}(1) \end{aligned}$$

Combining that with the above bound on  $f(r)$  we obtain:

$$f(r) < 2^{\mathcal{O}(2^{r/c} \log x)} \leq 2^{\mathcal{O}(2^{\log x + \mathcal{O}(1)} \log x)} = 2^{\mathcal{O}(x \log x)}$$

which gives the desired bound on the obtained values.  $\square$

Hence we can apply Theorem 1.28 to the circuit computing  $D_r[u, v]$  for the values  $p, t, \omega$  from Theorem 1.29. The running time of the algorithm is  $\mathcal{O}(|C|t \text{ polylog } p) = \mathcal{O}((|E(G)| + |V(G)|^3)x \text{ polylog } x)$  as  $|C| = \mathcal{O}(|E(G)| + |V(G)|^3 \cdot \text{polylog } x)$ ,  $t = \mathcal{O}(x \text{ polylog } x)$ ,  $p = \mathcal{O}(y^2 \text{ polylog } y)$  and  $y = \Theta(x)$ . The space complexity is bounded by  $\mathcal{O}(|C| \log p) = \mathcal{O}((|E(G)| + |V(G)|^3) \text{ polylog } x)$ . This concludes the proof of Theorem 1.13.

## Chapter 2

# Compressed Indexing for Consecutive Occurrences

### Publication

This chapter corresponds to the following publication: Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner, “Compressed Indexing for Consecutive Occurrences”, in: *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, June 26-28, 2023, Marne-la-Vallée, France, ed. by Laurent Bulteau and Zsuzsanna Lipták, vol. 259, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 12:1–12:22, DOI: 10.4230/LIPIcs.CPM.2023.12.

The fundamental question considered in algorithms on strings is that of indexing, that is, preprocessing a given string for specific queries. By now we have a number of efficient solutions for this problem when the queries ask for an exact occurrence of a given pattern  $P$ . However, practical applications motivate the necessity of considering more complex queries, for example concerning near occurrences of two patterns. Recently, Bille et al. [CPM 2021] introduced a variant of such queries, called gapped consecutive occurrences, in which a query consists of two patterns  $P_1$  and  $P_2$  and a range  $[a, b]$ , and one must find all consecutive occurrences  $(q_1, q_2)$  of  $P_1$  and  $P_2$  such that  $q_2 - q_1 \in [a, b]$ . By their results, we cannot hope for a very efficient indexing structure for such queries, even if  $a = 0$  is fixed (although at the same time they provided a non-trivial upper bound). Motivated by this, we focus on a text given as a straight-line program (SLP) and design an index taking space polynomial in the size of the grammar that answers such queries in time optimal up to polylog factors.

## 1 Introduction

In the indexing problem, the goal is to preprocess a string for locating occurrences of a given pattern. For a string of length  $N$ , structures such as the suffix tree [12] or the suffix array [47], use space linear in  $N$  and allow for answering such queries in time linear in the length of the pattern  $m$ . By now, we have multiple space- and time-efficient solutions for this problem (both in theory and in practice). We refer the reader to the excellent survey by Lewenstein [172] that provides an overview of some of the approaches and some of its extensions, highlighting its connection to orthogonal range searching.

However, from the point of view of possible applications, it is desirable to allow for more general queries than just locating an exact match of a given pattern in the preprocessed text, while keeping the time sublinear in the length of the preprocessed string. A very general query is locating a substring matching a regular expression. Very recently, Gibney and Thankachan [336] showed that if the Online Matrix-Vector multiplication conjecture holds, even with a polynomial preprocessing time we cannot answer regular expression



query in sublinear time. A more reasonable and yet interesting query could concern occurrences of two given patterns that are closest to each other, or just close enough.

Preprocessing a string for queries concerning two patterns has been first studied in the context of document retrieval, where the goal is to preprocess a collection of strings. There, in *the two patterns document retrieval problem* the query consists of two patterns  $P_1$  and  $P_2$ , and we must report all documents containing both of them [86]. In *the forbidden pattern query problem* we must report all documents containing  $P_1$  but not  $P_2$  [160]. For both problems, the asymptotically fastest linear-space solutions need as much as  $\Omega(\sqrt{N})$  time to answer a query, where  $N$  is the total length of all strings [161, 144]. That is, the complexity heavily depends on the length of the strings. Larsen et al. [210] established a connection between Boolean matrix multiplication and the two problems, thus providing a conditional explanation for the high  $\Omega(\sqrt{N})$  query complexity. Later, Kopelowitz et al. [220] provided an even stronger argument using a connection to the 3SUM problem. Even more relevant to this paper is the question considered by Kopelowitz and Krauthgamer [219], who asked for preprocessing a string for computing, given two patterns  $P_1$  and  $P_2$ , their occurrences that are closest to each other. The main result of their paper is a structure constructible in  $O(N^{1.5} \log^\epsilon N)$  time that answers such queries in  $O(|P_1| + |P_2| + \sqrt{N} \log^\epsilon N)$ , for a string of length  $N$ , for any  $\epsilon > 0$ . They also established a connection between Boolean matrix multiplication and this problem, highlighting a difficulty in removing the  $O(\sqrt{N})$  from both the preprocessing and query time at the same time.

The focus of this paper is the recently introduced variant of the indexing problem, called *gapped indexing for consecutive occurrences*, in which a query consists of two patterns  $P_1$  and  $P_2$  and a range  $[a, b]$ , and one must find the pairs of consecutive occurrences of  $P_1, P_2$  separated by a distance in the range  $[a, b]$ . Navarro and Thankachan [228] showed that for  $P_1 = P_2$  there is a  $O(n \log n)$ -space index with optimal query time  $O(m + \text{output})$ , where  $m = |P_1| = |P_2|$  and output is the number of pairs to report, but in conclusion they noticed that extending their solution to the general case of two patterns might not be possible. Bille et al. [349] provided an evidence of hardness of the general case and established a (conditional) lower bound for gapped indexing for consecutive occurrences, by connecting its complexity to that of set intersection. This lower bound suggests that, at least for indexes of size  $\tilde{O}(N)$ , achieving query time better than  $\tilde{O}(|P_1| + |P_2| + \sqrt{N})$  would contradict the Set Disjointness conjecture, even if  $a = 0$  is fixed. In particular, obtaining query time depending mostly on the lengths of the patterns (perhaps with some additional logarithms), arguably the whole point of string indexing, is unlikely in this case.

Motivated by the (conditional) lower bound for gapped indexing for consecutive occurrences, we consider the compressed version of this problem for query intervals  $[0, b]$ . For exact pattern matching, there is a long line of research devoted to designing the so-called compressed indexes, that is, indexing structures with the size being a function of the length of the compressed representation of the text, see e.g. the entry in the Encyclopedia of Algorithms [223] or the Encyclopedia of Database Systems [261]. This suggests the following research direction: can we design an efficient compressed gapped index for consecutive occurrences?

The answer of course depends on the chosen compression method. With a goal to design an index that uses very little space, we focus on the most challenging setting

when the compression is capable of describing a string of exponential length (in the size of its representation). An elegant formalism for such a compression method is that of straight-line programs (SLP), which are context-free grammars describing exactly one string. SLPs are known to capture the popular Lempel–Ziv compression method up to a logarithmic factor [80, 91], and at the same time provide a more convenient interface, and in particular, allow for random access in  $O(\log N)$  time [195].

By now it is known that pattern matching admits efficient indexing in SLP-compressed space. Assuming a string  $S$  of length  $N$  described by an SLP with  $g$  productions, Claude and Navarro [157] designed an  $O(g)$ -space index for  $S$  that allows retrieving all occurrences of a pattern of length  $m$  in time  $O(m^2 \log \log N + \text{output} \log g)$ . Recently, several results have improved the query time bound while still using a comparable  $O(g \log N)$  amount of space: Claude, Navarro and Pacheco [330] showed an index with query time  $O((m^2 + \text{output}) \log g)$ ; Christiansen et al. [329] used strings attractors to further improve the time bound to  $O(m + \text{output} \log^\epsilon N)$ ; and Díaz-Domínguez et al. [333] achieved  $O((m \log m + \text{output}) \log g)$  query time.

However it is not always the case that a highly compressible string is easier to preprocess. On the negative side, Abboud et al. [233] showed that, for some problems on compressed strings, such as computing the LCS, one cannot completely avoid a high dependency on the length of the uncompressed string and that for other problems on compressed strings, such as context-free grammar parsing or RNA folding, one essentially cannot hope for anything better than just decompressing the string and working with the uncompressed representation! This is also the case for some problems related to linear algebra [302]. Hence, it was not clear to us if one can avoid a high dependency on the length of the uncompressed string in the gapped indexing for consecutive occurrences problem.

In this work, we address the lower bound of Bille et al. [349] and show that, despite the negative results by Abboud et al. [233], one can circumvent it assuming that the text is very compressible:

**Theorem 2.1.** *For an SLP of size  $g$  representing a string  $S$  of length  $N$ , there is an  $O(g^5 \log^5 N)$ -space data structure that maintains the following queries: given two patterns  $P_1, P_2$  both of length  $O(m)$ , and a range  $[0, b]$ , report all output consecutive occurrences of  $P_1$  and  $P_2$  separated by a distance  $d \in [0, b]$ . The query time is  $O(m \log N + (1 + \text{output}) \cdot \log^4 N \log \log N)$ .*

While achieving  $O(g)$  space and  $O(m + \text{output})$  query time would contradict the Set Disjointness conjecture by the reduction of Bille et al. [349], one might wonder if the space can be improved without increasing the query time and what is the true complexity of the problem when  $a$  is not fixed (recall that  $[a, b]$  is the range limiting the distance between co-occurrences to report). While we leave improvement on space and the general case as an interesting open question, we show that in the simpler case  $a = 0, b = N$  (i.e. when there is no bound on the distance between the starting positions of  $P_1$  and  $P_2$ ), our techniques do allow for  $O(g^2 \log^4 N)$  space complexity, see Corollary 2.16<sup>1</sup>.

Throughout the paper we assume a unit-cost RAM model of computation with word size  $\Theta(\log N)$ . All space complexities refer to the number of words used by a data structure.

<sup>1</sup>Note that the conditional lower bound of Bille et al. [349] does not hold for this simpler case.

## 2 Preliminaries

A *string*  $S$  of length  $|S| = N$  is a sequence  $S[0]S[1]\dots S[N-1]$  of characters from an alphabet  $\Sigma$ . We denote the *reverse*  $S[N-1]S[N-2]\dots S[0]$  of  $S$  by  $\text{rev}(S)$ . We define  $S[i\dots j]$  to be equal to  $S[i]\dots S[j]$  which we call a *substring* of  $S$  if  $i \leq j$  and to the empty string otherwise. We also use notations  $S[i\dots j)$  and  $S(i\dots j]$  which naturally stand for  $S[i]\dots S[j-1]$  and  $S[i+1]\dots S[j]$ , respectively. We call a substring  $S[0\dots i]$  a *prefix* of  $S$  and use a simplified notation  $S[\dots i]$ , and a substring  $S[i\dots N-1]$  a *suffix* of  $S$  denoted by  $S[i\dots]$ . We say that  $X$  is a *substring* of  $S$  if  $X = S[i\dots j]$  for some  $0 \leq i \leq j \leq N-1$ . The index  $i$  is called an *occurrence* of  $X$  in  $S$ .

An occurrence  $q_1$  of  $P_1$  and an occurrence  $q_2$  of  $P_2$  form a *consecutive occurrence* (co-occurrence) of strings  $P_1, P_2$  in a string  $S$  if there are no occurrences of  $P_1, P_2$  between  $q_1$  and  $q_2$ , formally, there should be no occurrences of  $P_1$  in  $(q_1, q_2]$  and no occurrences of  $P_2$  in  $[q_1, q_2)$ . For brevity, we say that a co-occurrence is *b-close* if  $q_2 - q_1 \leq b$ .

An integer  $\pi$  is a *period* of a string  $S$  of length  $N$ , if  $S[i] = S[i + \pi]$  for all  $i = 0, \dots, N-1-\pi$ . The smallest period of a string  $S$  is called *the period* of  $S$ . We say that  $S$  is *periodic* if the period of  $S$  is at most  $N/2$ . We exploit the well-known corollary of the Fine and Wilf's periodicity lemma [7]:

**Corollary 2.2.** *If there are at least three occurrences of a string  $Y$  in a string  $X$ , where  $|X| \leq 2|Y|$ , then the occurrences of  $Y$  in  $X$  form an arithmetic progression with a difference equal to the period of  $Y$ .*

### 2.1 Grammars

**Definition 2.3** (Straight-line program [69]). *A straight-line program (SLP)  $G$  is a context-free grammar (CFG) consisting of a set of non-terminals, a set of terminals, an initial symbol, and a set of productions, satisfying the following properties:*

- *A production consists of a left-hand side and a right-hand side, where the left-hand side is a non-terminal  $A$  and the right-hand side is either a sequence  $BC$ , where  $B, C$  are non-terminals, or a terminal;*
- *Every non-terminal is on the left-hand side of exactly one production;*
- *There exists a linear order  $<$  on the non-terminals such that  $A < B$  whenever  $B$  occurs on the right-hand side of the production associated with  $A$ .*

A *run-length straight-line program* (RLSLP) [229] additionally allows productions of form  $A \rightarrow B^k$  for positive integers  $k$ , which correspond to concatenating  $k$  copies of  $B$ . If  $A$  is associated with a production  $A \rightarrow a$ , where  $a$  is a terminal, we denote  $\text{head}(A) = a$ ,  $\text{tail}(A) = \varepsilon$  (the empty string); if  $A$  is associated with a production  $A \rightarrow BC$ , we denote  $\text{head}(A) = B$ ,  $\text{tail}(A) = C$ ; and finally if  $A$  is associated with a production  $A \rightarrow B^k$ , then  $\text{head}(A) = B$ ,  $\text{tail}(A) = B^{k-1}$ .

The *expansion*  $\bar{S}$  of a sequence of terminals and non-terminals  $S$  is the string that is obtained by iteratively replacing non-terminals by the right-hand sides in the respective productions, until only terminals remain. We say that  $G$  *represents* the expansion of its initial symbol.

**Definition 2.4** (Parse tree). *The parse tree of a SLP (RLSLP) is a rooted tree defined as follows:*

- *The root is labeled by the initial symbol;*
- *Each internal node is labeled by a non-terminal;*
- *If  $S$  is the expansion of the initial symbol, then the  $i$ th leaf of the parse tree is labeled by a terminal  $S[i]$ ;*
- *A node labeled with a non-terminal  $A$  that is associated with a production  $A \rightarrow BC$ , where  $B, C$  are non-terminals, has 2 children labeled by  $B$  and  $C$ , respectively. If  $A$  is associated with a production  $A \rightarrow a$ , where  $a$  is a terminal, then the node has one child labeled by  $a$ .*
- *(RLSLP only) A node labeled with non-terminal  $A$  that is associated with a production  $A \rightarrow B^k$ , where  $B$  is a non-terminal, has  $k$  children, each labeled by  $B$ .*

The *size* of a grammar is its number of productions. The *height* of a grammar is the height of the parse tree. We say that a non-terminal  $A$  is an *ancestor* of a non-terminal  $B$  if there are nodes  $u, v$  of the parse tree labeled with  $A, B$  respectively, and  $u$  is an ancestor of  $v$ . For a node  $u$  of the parse tree, denote by  $\text{off}(u)$  the number of leaves to the left of the subtree rooted at  $u$ .

**Definition 2.5** (Relevant occurrences). *Let  $A$  be a non-terminal associated with a production  $A \rightarrow \text{head}(A)\text{tail}(A)$ . We say that an occurrence  $q$  of a string  $P$  in  $\bar{A}$  is relevant with a split  $s$  if  $q = |\text{head}(A)| - s \leq |\text{head}(A)| \leq q + |P| - 1$ .*

For example, in Fig. 2.1 the occurrence  $q = 3$  of  $P = cab$  is a relevant occurrence in  $\bar{C}$  with a split  $s = 1$  but  $\bar{A}$  contains no relevant occurrences of  $P$ .

**Claim 2.6.** *Let  $q$  be an occurrence of a string  $P$  in a string  $S$ . Consider the parse tree of an RLSLP representing  $S$ , and let  $w$  be the lowest node containing leaves  $S[q], S[q+1], \dots, S[q+|P|-1]$  in its subtree, then either*

1. *The label  $A$  of  $w$  is associated with a production  $A \rightarrow BC$ , and  $q - \text{off}(w)$  is a relevant occurrence in  $\bar{A}$ ; or*
2. *The label  $A$  of  $w$  is associated with a production  $A \rightarrow B^r$  and  $q - \text{off}(w) = q' + r'|\bar{B}|$  for some  $0 \leq r' \leq r$ , where  $q'$  is a relevant occurrence of  $P$  in  $\bar{A}$ .*

*Proof.* Assume first that  $A$  is associated with a production  $A \rightarrow BC$ . We then have that the subtree rooted at the left child of  $w$  (that corresponds to  $\bar{B}$ ) does not contain  $S[q+|P|-1]$  and the subtree rooted at the right child of  $w$  (that corresponds to  $\bar{C}$ ) does not contain  $S[q]$ . As a consequence,  $q - \text{off}(w)$  is a relevant occurrence in  $\bar{A}$ .

Consider now the case where  $A$  is associated with a production  $A \rightarrow B^r$ . The leaves labeled by  $S[q]$  and  $S[q+|P|-1]$  belong to the subtrees rooted at different children of  $A$ . If  $S[q]$  belongs to the subtree rooted at the  $(r'+1)$ -th child of  $A$ , then  $q' = q - \text{off}(w) - |\bar{B}| \cdot r'$  is a relevant occurrence of  $P$  in  $\bar{A}$ .  $\square$

**Definition 2.7** (Splits). Consider a non-terminal  $A$  of an RLSLP  $G$ . If it is associated with a production  $A \rightarrow BC$ , define

$$\text{Splits}(A, P) = \text{Splits}_{\text{rev}}(A, P) = \{s : q \text{ is a relevant occurrence of } P \text{ in } \bar{A} \text{ with a split } s\}.$$

If  $A$  is associated with a rule  $A \rightarrow B^k$ , define

$$\text{Splits}(A, P) = \{s : q \text{ is a relevant occurrence of } P \text{ in } \bar{A} \text{ with a split } s\};$$

$$\text{Splits}_{\text{rev}}(A, P) = \{|P| - s : q \text{ is a relevant occurrence of } \text{rev}(P) \text{ in } \text{rev}(\bar{A}) \text{ with split } s\}.$$

Define  $\text{Splits}(G, P)$  ( $\text{Splits}_{\text{rev}}(G, P)$ ) to be the union of  $\text{Splits}(A, P)$  ( $\text{Splits}_{\text{rev}}(A, P)$ ) over all non-terminals  $A$  in  $G$ , and  $\text{Splits}'(G, P) = \text{Splits}(G, P) \cup \text{Splits}_{\text{rev}}(G, P)$ .

We need the following lemma, which can be derived from Gawrychowski et al. [269]:

**Lemma 2.8.** Let  $G$  be an SLP of size  $g$  representing a string  $S$  of length  $N$ , where  $g \leq N$ . There exists a Las Vegas algorithm that builds a RLSLP  $G'$  of size  $g' = O(g \log N)$  of height  $h = O(\log N)$  representing  $S$  in time  $O(g \log N)$  with high probability. This RLSLP has the following additional property: For a pattern  $P$  of length  $m$ , we can in  $O(m \log N)$  time provide a certificate that  $P$  does not occur in  $S$ , or compute the set  $\text{Splits}'(G', P)$ . In the latter case,  $|\text{Splits}'(G', P)| = O(\log N)$ .

## 2.2 Compact Tries

We assume the reader to be familiar with the definition of a compact trie (see e.g. [57]). Informally, a trie is a tree that represents a lexicographically ordered set of strings. The edges of a trie are labeled with strings. We define the label  $\lambda(u)$  of a node  $u$  to be the concatenation of labels on the path from the root to  $u$  and an interval  $I(u)$  to be the interval of the set of strings starting with  $\lambda(u)$ . From the implementation point of view, we assume that a node  $u$  is specified by the interval  $I(u)$ . The *locus* of a string  $P$  is the minimum depth node  $u$  such that  $P$  is a prefix of  $\lambda(u)$ .

The standard tree-based implementation of a trie for a generic set of strings  $\mathcal{S} = \{S_1, \dots, S_k\}$  takes  $\Theta\left(\sum_{i=1}^k |S_i|\right)$  space. Given a pattern  $P$  of length  $m$  and  $\tau > 0$  suffixes  $Q_1, \dots, Q_\tau$  of  $P$ , the trie allows retrieving the ranges of strings in (the lexicographically-sorted)  $\mathcal{S}$  prefixed by  $Q_1, \dots, Q_\tau$  in  $O(m^2)$  time. However, in this work, we build the tries for very special sets of strings only, which allows for a much more efficient implementation based on the techniques of Christiansen et al. [329]:

**Lemma 2.9.** Given an RLSLP  $G$  of size  $g$  and height  $h$ . Assume that every string in a set  $\mathcal{S}$  is either a prefix or a suffix of the expansion of a non-terminal of  $G$  or its reverse. The trie for  $\mathcal{S}$  can be implemented in space  $O(|\mathcal{S}|)$  to maintain the following queries in  $O(m + \tau \cdot (h + \log m))$  time: Given a pattern  $P$  of length  $m$  and suffixes  $Q_i$  of  $P$ ,  $1 \leq i \leq \tau$ , find, for each  $i$ , the interval of strings in the (lexicographically sorted)  $\mathcal{S}$  prefixed by  $Q_i$ .

*Proof.* Let us first recall the definition of the Karp–Rabin fingerprint.

**Definition 2.10** (Karp–Rabin fingerprint). For a prime  $p$  and an  $r \in \mathbb{F}_p^*$ , the Karp–Rabin fingerprint [35] of a string  $X$  is defined as a tuple  $(r^{|X|-1} \bmod p, r^{-|X|+1} \bmod p, \varphi_{p,r}(X))$ , where  $\varphi_{p,r}(X) = \sum_{k=0}^{|X|-1} S[k]r^k \bmod p$ .

We use the result of Christiansen et al. [329], which builds on Belazzougui et al. [139] and Gagie et al. [179, 263].

**Fact 2.11** ([329, Lemma 6.5]). *Let  $\mathcal{S}$  be a set of strings and assume we have a data structure supporting extraction of any length- $l$  prefix of strings in  $\mathcal{S}$  in time  $f_e(l)$  and computing the Karp–Rabin fingerprint  $\varphi$  of any length- $l$  prefix of a string in  $\mathcal{S}$  in time  $f_h(l)$ . We can then build a data structure that uses  $O(|\mathcal{S}|)$  space and supports the following queries in  $O(m + f_e(m) + \tau(f_h(m) + \log m))$  time: Given a pattern  $P$  of length  $m$  and  $\tau > 0$  suffixes  $Q_1, \dots, Q_\tau$  of  $P$ , find the intervals of strings in (the lexicographically-sorted)  $\mathcal{S}$  prefixed by  $Q_1, \dots, Q_\tau$ .*

It should be noted that despite using a hash function, the query algorithm is deterministic: the proof shows that  $p$  and  $r$  can be chosen during the construction time to ensure that there are no collisions on the substrings of the strings in  $\mathcal{S}$ .

To bound  $f_e$ , we use [329, Lemma 6.6] which builds on Gąsieniec et al. [100] and Claude and Navarro [157].

**Fact 2.12** ([329, Lemma 6.6]). *Given an RLSP of size  $O(g)$ , there exists a data structure of size  $O(g)$  such that any length- $l$  prefix or suffix of  $\bar{A}$  can be obtained from any non-terminal  $A$  in time  $f_e(l) = O(l)$ .*

To bound  $f_h(l)$ , we introduce a simple construction based on the following well-known fact:

**Fact 2.13.** *Consider strings  $X, Y, Z$  where  $XY = Z$ . Given the Karp–Rabin fingerprints of two of the three strings, one can compute the fingerprint of the third string in constant time.*

**Claim 2.14.** *Given a RLSP  $G$  of size  $g$  and height  $h$ , there exists a data structure of size  $O(g)$  that given a non-terminal  $A$  and an integer  $l$  allows to retrieve the Karp–Rabin fingerprints of the length- $l$  prefix and suffix of  $\bar{A}^r$  and  $\text{rev}(\bar{A}^r)$  in time  $f_h(l) = O(h + \log l)$ .*

*Proof.* The claim for  $\text{rev}(\bar{A}^r)$  follows for the claim for  $\bar{A}^r$  by considering the grammar  $G_{\text{rev}}$ , where the order of the non-terminals in each production is reversed. Below we focus on extracting the fingerprints for  $\bar{A}^r$ , and we further restrict our attention to prefixes of  $\bar{A}^r$ , the algorithm for suffixes being analogous.

The data structure consists of two sets. The first set contains the lengths of the expansions of all non-terminals in the grammar, and the second one their fingerprints.

By Fact 2.13 and doubling, it suffices to show an algorithm for computing the fingerprint of the length- $l$  prefix of  $\bar{A}$ . Assume that  $A$  associated with a rule  $A \rightarrow BC$ . If the length of  $\bar{A}$  is smaller than  $l$ , we return error. Otherwise, to compute the fingerprint of the length- $l$  prefix of  $\bar{A}$ , we consider two cases. If  $l \leq |\bar{B}|$ , we recurse on  $B$  to retrieve the fingerprint of the  $l$ -length prefix of  $\bar{B}$ . Otherwise, we recurse on  $C$  to retrieve the fingerprint of  $\bar{C}[\dots l - |\bar{B}|]$  and then compute the fingerprint of the  $l$ -length prefix of  $\bar{A}$  from the fingerprints of  $\bar{B}$  and  $\bar{C}[\dots l - |\bar{B}|]$  in constant time by Fact 2.13.

For a non-terminal  $A$  associated with a rule  $A \rightarrow B^r$ , we compute the fingerprint analogously. If the length of  $\bar{A}$  is smaller than  $l$ , we return error. Otherwise, let  $q$  be such that  $q \cdot |\bar{B}| \leq l < (q + 1) \cdot |\bar{B}|$ . We compute the fingerprint of  $\bar{B}^q$  from the fingerprint of  $\bar{B}$  by applying Fact 2.13  $O(1 + \log q)$  times, and the fingerprint of  $\bar{B}[\dots l - q \cdot |\bar{B}|]$  recursively. We can then apply Fact 2.13 to compute the fingerprint of the length- $l$  prefix

of  $\bar{A}$  in constant time. Note that in this case, the length of the prefix decreases by a factor at least  $q$ .

If we are in a terminal  $A$ , the calculation takes  $O(1)$  time (the prefix must be equal to  $A$  itself).

In total, we spend  $O(h + \log l)$  time as we recurse  $O(h)$  times, and whenever we spend more than constant time in a symbol, we charge it on the decrease in the length. The fingerprints of length- $l$  suffixes are computed analogously.  $\square$

By substituting the bounds for  $f_e(l)$  (Fact 2.13) and  $f_h(l)$  (Claim 2.12) into Fact 2.11, we obtain the claim of the lemma.  $\square$

### 3 Relevant, Extremal, and Predecessor Occurrences in a Non-terminal

In this section, we present a data structure that allows various efficient queries, which we will need to prove Theorem 2.1. We also show how it can be leveraged for an index in the simpler case of consecutive occurrences ( $a = 0, b = N$ ). Recall that the text  $S$  is a string of length  $N$  represented by an SLP  $G$  of size  $g$ . By applying Lemma 2.8, we transform  $G$  into an RLSP  $G'$  of size  $g' = O(g \log N)$  and depth  $h = O(\log N)$  representing  $S$ , which we fix from now on. We start by showing that  $G'$  can be processed in small space to allow multiple efficient queries:

**Theorem 2.15.** *There is a  $O(g^2 \log^4 N)$ -space data structure for  $G'$  that given a pattern  $P$  of length  $m$  can preprocess it in  $O(m \log N + \log^2 N)$  time to support the following queries for a given non-terminal  $A$  of  $G'$ :*

1. *Report the sorted set of relevant occurrences of  $P$  in  $\bar{A}$  in  $O(\log N)$  time;*
2. *Decide whether there is an occurrence of  $P$  in  $\bar{A}$  in  $O(\log N \log \log N)$  time;*
3. *Report the leftmost and the rightmost occurrences of  $P$  in  $\bar{A}$ ,  $\overline{\text{head}(A)}$ , and  $\overline{\text{tail}(A)}$  in  $O(\log^2 N \log \log N)$  time;*
4. *Given a position  $p$ , find the rightmost (leftmost) occurrence  $q \leq p$  ( $q \geq p$ ) of  $P$  in  $\bar{A}$  in  $O(\log^3 N \log \log N)$  time (predecessor/successor).*

Before we proceed to the proof, let us derive a data structure to report all consecutive occurrences (co-occurrences) of a given pair of patterns.

**Corollary 2.16.** *For an SLP of size  $g$  representing a string  $S$  of length  $N$ , there is an  $O(g^2 \log^4 N)$ -space data structure that supports the following queries: given two patterns  $P_1, P_2$  both of length  $O(m)$ , report all output co-occurrences of  $P_1$  and  $P_2$  in  $S$ . The query time is  $O(m \log N + (1 + \text{output}) \cdot \log^3 N \log \log N)$ .*

*Proof.* We exploit the data structure of Theorem 2.15 for  $G'$ . To report all co-occurrences of  $P_1, P_2$  in  $S$ , we preprocess  $P_1, P_2$  in  $O(m \log N + \log^2 N)$  time and then proceed as follows. Suppose that we want to find the leftmost co-occurrence of  $P_1$  and  $P_2$  in the string  $S[i \dots]$ , where at the beginning  $i = 0$ . We find the leftmost occurrence  $q'_1$  of  $P_1$

with  $q'_1 \geq i$  (if it exists) by a successor query on the initial symbol of  $G'$  (the expansion of which is the entire string  $S$ ). Then we find the leftmost occurrence  $q_2$  of  $P_2$  with  $q_2 \geq q'_1$  (if it exists) by a successor query and the rightmost occurrence  $q_1$  of  $P_1$  with  $q_1 \leq q_2$  by a predecessor query. If either  $q'_1$  or  $q_2$  do not exist, then there are no more co-occurrences in  $S[i \dots]$ . Otherwise, clearly,  $(q_1, q_2)$  is a co-occurrence, and there can be no other co-occurrences starting in  $S[i \dots q_2]$ . In this case, we return  $(q_1, q_2)$  and set  $i = q_2 + 1$ . The running time of the retrieval phase is  $O(\log^3 N \log \log N \cdot (\text{output} + 1))$ , since we use at most three successor/predecessor queries to either output a new co-occurrence or decide that there are no more co-occurrences.  $\square$

### 3.1 Proof of Theorem 2.15

The data structure consists of two compact tries  $T_{pre}$  and  $T_{suf}$  defined as follows. For each non-terminal  $A$ , we store  $\text{rev}(\text{head}(A))$  in  $T_{pre}$  and  $\text{tail}(A)$  in  $T_{suf}$ . We augment  $T_{pre}$  and  $T_{suf}$  by computing their heavy path decomposition:

**Definition 2.17.** *The heavy path of a trie  $T$  is the path that starts at the root of  $T$  and at each node  $v$  on the path branches to the child with the largest number of leaves in its subtree (heavy child), with ties broken arbitrarily. The heavy path decomposition is a set of disjoint paths defined recursively, namely it is defined to be a union of the singleton set containing the heavy path of  $T$  and the heavy path decompositions of the subtrees of  $T$  that hang off the heavy path.*

For each non-terminal  $A$  of  $G'$ , a heavy path  $h_{pre}$  in  $T_{pre}$ , and a heavy path  $h_{suf}$  in  $T_{suf}$ , we construct a multiset of points  $\mathcal{P}(A, h_{pre}, h_{suf})$ . For every non-terminal  $A'$  and nodes  $u \in h_{pre}$ ,  $v \in h_{suf}$  the multiset contains a point  $(|\lambda(u)|, |\lambda(v)|)$  iff  $A'$ ,  $u$ ,  $v$  satisfy the following properties:

1.  $A$  is an ancestor of  $A'$ ;
2.  $I(u)$  contains  $\text{rev}(\text{head}(A'))$  and  $I(v)$  contains  $\text{tail}(A')$ .
3.  $u, v$  are the lowest nodes in  $h_{pre}, h_{suf}$ , respectively, satisfying Property 2.

(See Fig. 2.1.) The set  $\mathcal{P}(A, h_{pre}, h_{suf})$  is stored in a two-sided 2D orthogonal range emptiness data structure [172, 169] which occupies  $O(|\mathcal{P}(A, h_{pre}, h_{suf})|)$  space. Given a 2D range of the form  $[\alpha, \infty] \times [\beta, \infty]$ , it allows to decide whether the range contains a point in  $\mathcal{P}(A, h_{pre}, h_{suf})$  in  $O(\log \log N)$  time.

**Claim 2.18.** *The data structure occupies  $O(g^2 \log^4 N)$  space.*

*Proof.* Each non-terminal  $A'$  has at most  $g'$  distinct ancestors and each root-to-leaf path in  $T_{pre}$  or  $T_{suf}$  crosses  $O(\log g')$  heavy paths (as each time we switch heavy paths, the number of leaves in the subtree of the current node decreases by at least a factor of two). As a corollary, each non-terminal creates  $O(g' \log^2 g') = O(g \log^3 N)$  points across all orthogonal range emptiness data structures.  $\square$

When we receive a pattern  $P$ , we compute  $\text{Splits}'(G', P)$  via Lemma 2.8 in  $O(m \log N)$  time or provide a certificate that  $P$  does not occur in  $S$ , in which case there are no occurrences of  $P$  in the expansions of the non-terminals of  $G'$ . Recall that  $|\text{Splits}'(G', P)| \in$



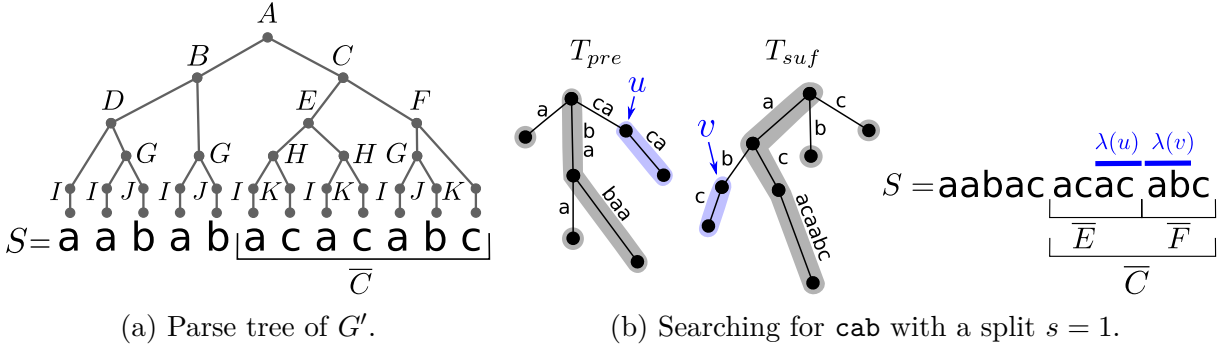


Figure 2.1: A string  $S = \text{aababacacabc}$  is generated by an SLP  $G'$ . Nodes  $u$  and  $v$  are the loci of  $\text{c}$  and  $\text{ab}$  in  $T_{pre}$  and  $T_{suf}$  respectively. The heavy paths  $h_{pre}$  in  $T_{pre}$  and  $h_{suf}$  in  $T_{suf}$  are shown in blue. We have  $(2, 2) \in \mathcal{P}(A, h_{pre}, h_{suf})$  corresponding to  $C, u, v$ .

$O(\log N)$ . We then sort  $\text{Splits}'(G', P)$  in  $O(\log^2 N)$  time (a technicality which will allow us reporting relevant occurrences sorted without time overhead). Finally, we compute, for each  $s \in \text{Splits}'(G', P)$ , the interval of strings in  $T_{pre}$  prefixed by  $\text{rev}(P[\dots s])$  (which is the interval  $I(u)$  for the locus  $u$  of  $\text{rev}(P[\dots s])$  in  $T_{pre}$ ) and the interval of strings in  $T_{suf}$  prefixed by  $P(s \dots)$  (which is the interval  $I(u)$  for the locus  $u$  of  $P(s \dots)$  in  $T_{suf}$ ). By Lemma 2.9, with  $\tau = |\text{Splits}'(G', P)| = O(\log N)$  and  $h = O(\log N)$ , this step takes  $O(m + \log^2 N)$  time.

Reporting relevant occurrences is easy: by definition, each relevant occurrence  $q$  of  $P$  in  $\bar{A}$  is equal to  $|\text{head}(\bar{A})| - s$  for some  $s \in \text{Splits}'(G', P)$  such that  $\text{rev}(P[\dots s])$  is a prefix of  $\text{rev}(\text{head}(\bar{A}))$  and  $P(s \dots)$  is a prefix of  $\text{tail}(\bar{A})$ . As we already know the intervals of the strings in  $T_{suf}$  and  $T_{pre}$  starting with  $\text{rev}(P[\dots s])$  and  $P(s \dots)$ , respectively, both conditions can be checked in constant time per split, or in  $O(|\text{Splits}'(G', P)|) = O(\log N)$  time overall. Note that since  $\text{Splits}'(G', P)$  are sorted, the relevant occurrences are reported sorted as well.

We now explain how to answer emptiness queries on a non-terminal:

**Claim 2.19.** *Let  $A$  be a non-terminal labeling a node in the parse tree of  $G'$ . We can decide whether  $\bar{A}$  contains an occurrence of  $P$  in  $O(\log N \log \log N)$  time.*

*Proof.* Below we show that  $P$  occurs in  $\bar{A}$  iff there exists a split  $s \in \text{Splits}'(G', P)$  such that for  $u$  being the locus of  $\text{rev}(P[\dots s])$  in  $T_{pre}$  and  $v$  the locus of  $P(s \dots)$  in  $T_{suf}$ , for  $h_{pre}$  the heavy path containing  $u$  in  $T_{pre}$  and  $h_{suf}$  the heavy path containing  $v$  in  $T_{suf}$ , the rectangle  $[|\lambda(u)|, +\infty] \times [|\lambda(v)|, +\infty]$  contains a point from  $\mathcal{P}(A, h_{pre}, h_{suf})$ . Before we proceed to the proof, observe that by the bound on  $|\text{Splits}'(G', P)|$  this allows us to decide whether  $P$  occurs in  $\bar{A}$  in  $O(\log N)$  range emptiness queries, which results in  $O(\log N \log \log N)$  query time.

Assume that  $[|\lambda(u)|, +\infty] \times [|\lambda(v)|, +\infty]$  contains a point  $(x, y) \in \mathcal{P}(A, h_{pre}, h_{suf})$  corresponding to a non-terminal  $A'$ . By construction,  $A$  is an ancestor of  $A'$ , the subtree of  $u$  contains a leaf corresponding to  $\text{rev}(\text{head}(A'))$  and the subtree of  $v$  contains a leaf corresponding to  $\text{tail}(A')$ . Consequently,  $\bar{A}'$  contains an occurrence of  $P$ , which implies that  $\bar{A}$  contains an occurrence of  $P$ . To show the reverse direction, let  $\ell = \text{off}(u) + 1$  and  $r = \text{off}(u) + |\bar{A}|$ , i.e.  $S[\ell \dots r] = \bar{A}$ . The string  $\bar{A}$  contains an occurrence  $\bar{A}[q \dots q + |P|]$  of  $P$  iff  $S[\ell + q \dots \ell + q + |P|]$  is an occurrence of  $P$  in  $S$ . From Claim 2.6 it follows that if  $w$  is

the lowest node in the parse tree of  $G'$  that contains leaves  $S[\ell + q], \dots, S[\ell + q + |P| - 1]$  in its subtree and  $A'$  is its label, then there exists a split  $s \in \text{Splits}'(G', P)$  such that  $\text{rev}(P[\dots s])$  is a prefix of  $\text{rev}(\text{head}(A'))$  and  $P(s \dots)$  of  $\text{tail}(A')$ . By definition of  $u$  and  $v$ , the leaf of  $T_{pre}$  labeled with  $\text{rev}(\text{head}(A'))$  belongs to  $I(u)$  and the leaf of  $T_{suf}$  labeled with  $\text{tail}(A')$  belongs to  $I(v)$ . Let  $h_{pre}$  ( $h_{suf}$ ) be the heavy path in  $T_{pre}$  ( $T_{suf}$ ) containing  $u$  ( $v$ ) and  $(x, y)$  be the point in  $\mathcal{P}(A, h_{pre}, h_{suf})$  created for  $A'$ . As  $|\lambda(u)| \leq x$  and  $|\lambda(v)| \leq y$ , the rectangle  $[|\lambda(u)|, +\infty] \times [|\lambda(v)|, +\infty]$  is not empty.  $\square$

It remains to explain how to retrieve the leftmost/rightmost occurrences in a non-terminal, as well as to answer predecessor/successor queries. The main idea for all four types of queries is to start at any node of the parse tree of  $G'$  labeled by  $A$  and recurse down via emptiness queries and case inspection. Since the length of the expansion decreases each time we recurse from a non-terminal to its child and the height of  $G'$  is  $h = O(\log N)$ , this allows to achieve the desired query time.

**Claim 2.20.** *Given a non-terminal  $A$  of  $G'$ , we can find the leftmost and the rightmost occurrences of  $P$  in  $\overline{A}$  and as a corollary in  $\text{head}(A)$  and  $\text{tail}(A)$  in  $O(\log^2 N \log \log N)$  time.*

*Proof.* We explain how to find the leftmost occurrence of  $P$  in  $\overline{A}$ , the rightmost one can be found analogously. We first check whether  $\overline{A}$  contains an occurrence of  $P$  via Claim 2.19 in  $O(\log N \log \log N)$  time. If it does not, we can stop immediately. Below we assume that there is an occurrence of  $P$  in  $\overline{A}$ . Next, we check whether  $\text{head}(A)$  contains an occurrence of  $P$  via Claim 2.19 in  $O(\log N \log \log N)$  time. If it does, the leftmost occurrence of  $P$  in  $\overline{A}$  is the leftmost occurrence of  $P$  in  $\text{head}(A)$  and we can find it by recursing on  $\text{head}(A)$ . If  $\text{head}(A)$  does not contain an occurrence of  $P$ , but  $\overline{A}$  contains relevant occurrences of  $P$ , then the leftmost occurrence of  $P$  in  $\overline{A}$  is the leftmost relevant occurrence of  $P$  in  $\overline{A}$  and we can find it in  $O(|\text{Splits}'(G', P)|) = O(\log N)$  time. Finally, if  $P$  neither occurs in  $\text{head}(A)$  nor has relevant occurrences in  $\overline{A}$ , then the leftmost occurrence of  $P$  in  $\overline{A}$  is the leftmost occurrence of  $P$  in  $\text{tail}(A)$ . If  $\text{tail}(A)$  is a non-terminal  $C$ , we recurse on  $C$  to find it. If  $\text{tail}(A) = B^{r-1}$  for a non-terminal  $B$ ,  $\text{tail}(A)$  cannot contain an occurrence of  $P$  because  $\overline{B}$  does not contain  $P$  and there are no relevant occurrences in  $A$ . We recurse down at most  $h = O(\log N)$  levels, and spend  $O(\log N \log \log N)$  time per level. The claim follows.  $\square$

**Lemma 2.21.** *Let  $A$  be a non-terminal of  $G'$ . For any position  $p$ , we can find the rightmost occurrence  $q \leq p$  of  $P$  in  $\overline{A}$  and the leftmost occurrence  $q' \geq p$  of  $P$  in  $\overline{A}$  in  $O(\log^3 N \log \log N)$  time.*

*Proof.* First we describe how to locate  $q$ . Consider a node  $u$  of the parse tree of  $G'$  labeled by  $A$ . The algorithm starts at  $u$  and recurses down. Let  $A'$  be the label of the current node. It computes the leftmost and rightmost occurrences in  $\overline{A'}$ ,  $\text{head}(A')$  and  $\text{tail}(A')$  as well as all relevant occurrences via Claim 2.20. If the leftmost occurrence of  $P$  in  $\overline{A'}$  is larger than  $p$ , the search result is empty. Otherwise, consider two cases.

1.  $A'$  is associated with a rule  $A' \rightarrow B'C'$ , i.e.  $\text{head}(A') = B'$ ,  $\text{tail}(A') = C'$ .
  - (a) If  $p \leq |\overline{B'}|$ , recurse on  $B'$ .

- (b) Assume now that  $p > |\overline{B'}|$ . If the leftmost occurrence of  $P$  in  $\overline{C'}$  is smaller than  $p$ , recurse on  $C'$ . Otherwise, return the rightmost relevant occurrence of  $P$  in  $\overline{A'}$  if it exists else the rightmost occurrence of  $P$  in  $\overline{B'}$ .
2.  $A'$  is associated with a rule  $A \rightarrow (B')^r$ , i.e.  $\text{head}(A') = B'$ ,  $\text{tail}(A') = (B')^{r-1}$ . Let an integer  $k$  be such that  $(k-1) \cdot |\overline{B'}| + 1 \leq p \leq k \cdot |\overline{B'}|$ . The desired occurrence of  $P$  is the rightmost one of the following ones:
- (a) The rightmost occurrence  $q \leq p$  of  $P$  which crosses the border between two copies of  $\overline{B'}$ . To compute  $q$ , we compute all relevant occurrences of  $P$  in  $\overline{A'}$  and then shift each of them by the maximal possible shift  $r' \cdot |\overline{B'}|$ , where  $r'$  is an integer, which guarantees that it starts before  $p$  and ends before  $|\overline{A'}|$  and take the rightmost of the computed occurrences to obtain  $q$ .
  - (b) The rightmost occurrence  $q$  of  $P$  such that for some integer  $k'$ , we have  $(k'-1) \cdot |\overline{B'}| \leq q \leq q + |P| - 1 \leq k' \cdot |\overline{B'}|$  (i.e. the occurrence fully belongs to some copy of  $\overline{B'}$ ). In this case,  $q$  is either the rightmost occurrence of  $P$  in the  $(k-1)$ -th copy of  $\overline{B'}$ , or the rightmost occurrence of  $P$  in the  $k$ -th copy of  $\overline{B'}$  that is smaller than  $p$ . In the second case, we compute  $q$  by recursing on  $B'$ .

We recurse down at most  $h$  levels. On each level we spend  $O(\log^2 N \log \log N)$  time to compute the leftmost, the rightmost, and relevant occurrences and respective shifts for a constant number of non-terminals via Claim 2.20. Therefore, in total we spend  $O(h \cdot \log^2 N \log \log N) = O(\log^3 N \log \log N)$  time.

Locating  $q'$  is very similar and differs only in small technicalities. The algorithm starts at the node  $u$  and recurses down. Let  $A'$  be the label of the current node. We compute the leftmost and rightmost occurrences in  $\overline{A'}$ ,  $\overline{\text{head}(A')}$  and  $\overline{\text{tail}(A')}$  as well as all relevant occurrences via Claim 2.20. If the rightmost occurrence of  $P$  in  $\overline{A'}$  is smaller than  $p$ , the search result is empty. Otherwise, consider two cases.

- 1.  $A'$  is associated with a rule  $A' \rightarrow B'C'$ , i.e.  $\text{head}(A') = B'$ ,  $\text{tail}(A') = C'$ .
  - (a) If  $p > |\overline{B'}|$ , recurse on  $C'$ .
  - (b) Assume now that  $p \leq |\overline{B'}|$ . If the rightmost occurrence of  $P$  in  $\overline{B'}$  is larger than  $p$ , recurse on  $B'$ . Otherwise, return the leftmost relevant occurrence  $q$  satisfying  $q \geq p$ , if it exists, and otherwise the leftmost occurrence of  $P$  in  $\overline{C'}$ .
- 2.  $A'$  is associated with a rule  $A \rightarrow (B')^r$ , i.e.  $\text{head}(A') = B'$ ,  $\text{tail}(A') = (B')^{r-1}$ . Let an integer  $k$  be such that  $(k-1) \cdot |\overline{B'}| + 1 \leq p \leq k \cdot |\overline{B'}|$ . The desired occurrence of  $P$  is the leftmost one of the following ones:
  - (a) The leftmost occurrence  $q' \geq p$  of  $P$  which crosses the border between two copies of  $\overline{B'}$ . To compute  $q'$ , we compute all relevant occurrences of  $P$  in  $\overline{A'}$  and then shift each of them by the minimal possible shift  $r' \cdot |\overline{B'}|$ , where  $r'$  is an integer, which guarantees that it starts after  $p$  and ends before  $|\overline{A'}|$  (if it exists) and take the leftmost of the computed occurrences to obtain  $q$ .
  - (b) The leftmost occurrence  $q'$  of  $P$  such that for some integer  $k'$ , we have  $(k'-1) \cdot |\overline{B'}| \leq q' \leq q' + |P| - 1 \leq k' \cdot |\overline{B'}|$  (i.e. the occurrence fully belongs to

some copy of  $\overline{B'}$ ). In this case,  $q'$  is either the leftmost occurrence of  $P$  in the  $(k + 1)$ -st copy of  $\overline{B'}$ , or the leftmost occurrence of  $P$  in the  $k$ -th copy of  $\overline{B'}$  that is larger than  $p$ . In the second case, we compute  $q'$  by recursing on  $B'$ .

The time complexities are the same as for computing  $q$ .  $\square$

## 4 Compressed Indexing for Close Co-occurrences

In this section, we show our main result, Theorem 2.1. Recall that  $S$  is a string of length  $N$  represented by an SLP  $G$  of size  $g$ . We start by applying Lemma 2.8 to transform  $G$  into an RLSLP  $G'$  of size  $g' = O(g \log N)$  and height  $h = O(\log N)$  representing  $S$ .

The query algorithm uses the following strategy: first, it identifies all non-terminals of  $G'$  such that their expansion contains a  $b$ -close relevant co-occurrence, where a relevant co-occurrence is defined similarly to a relevant occurrence:

**Definition 2.22** (Relevant co-occurrence). *Let  $A$  be a non-terminal of  $G'$ . We say that a co-occurrence  $(q_1, q_2)$  of  $P_1, P_2$  in  $\overline{A}$  is relevant if  $q_1 \leq |\overline{\text{head}(A)}| \leq q_2 + |P_2| - 1$ .*

Second, it retrieves all  $b$ -close relevant co-occurrences in each of those non-terminals, and finally, reports all  $b$ -close co-occurrences by traversing the (pruned) parse tree of  $G'$ , which is possible due to the following claim:

**Claim 2.23.** *Assume that  $P_2$  is not a substring of  $P_1$ , and let  $(q_1, q_2)$  be a co-occurrence of  $P_1, P_2$  in a string  $S$ . In the parse tree of  $G'$ , there exists a unique node  $u$  such that either*

1. *Its label  $A$  is associated with a production  $A \rightarrow BC$ , and  $(q_1 - \text{off}(u), q_2 - \text{off}(u))$  is a relevant co-occurrence of  $P_1, P_2$  in  $\overline{A}$ ;*
2. *Its label  $A$  is associated with a production  $A \rightarrow B^k$ ,  $q_1 - \text{off}(u) = q'_1 + k'|\overline{B}|$ ,  $q_2 - \text{off}(u) = q'_2 + k'|\overline{B}|$  for some  $0 \leq k' \leq k$ , where  $(q'_1, q'_2)$  is a relevant co-occurrence of  $P_1, P_2$  in  $\overline{A}$ .*

*Proof.* Let  $A$  be the label of the lowest node  $u$  in the parse tree that contains leaves  $S[q_1], S[q_1 + 1], \dots, S[q_2 + |P_2| - 1]$  in its subtree. Because  $P_2$  is not a substring of  $P_1$ ,  $A$  cannot be associated with a production  $A \rightarrow a$ . By definition,  $S[\text{off}(u) + 1]$  is the leftmost leaf in the subtree of this node.

Assume first that  $A$  is associated with a production  $A \rightarrow BC$ . We then have that the subtree rooted at the left child of  $u$  (labelled by  $B$ ) does not contain  $S[q_2 + |P_2| - 1]$  and the subtree rooted at the right child of  $u$  (labelled by  $C$ ) does not contain  $S[q_1]$ . As a consequence,  $(q_1 - \text{off}(u), q_2 - \text{off}(u))$  is a relevant co-occurrence of  $P_1, P_2$  in  $\overline{A}$ .

Consider now the case where  $A$  is associated with a production  $A \rightarrow B^k$ . The leaves labelled by  $S[q_1]$  and  $S[q_2 + |P_2| - 1]$  belong to the subtrees rooted at different children of  $A$ . If  $S[q_1]$  belongs to the subtree rooted at the  $k'$ -th child of  $A$ , then  $(q_1 - \text{off}(u) - |\overline{B}| \cdot (k' - 1), q_2 - \text{off}(u) - |\overline{B}| \cdot (k' - 1))$  is a relevant co-occurrence of  $P_1, P_2$  in  $\overline{A}$ .  $\square$

## 4.1 Combinatorial Observations

Informally, we define a set of  $O(g^2)$  strings and show that for any patterns  $P_1, P_2$  there are two strings  $S_1, S_2$  in the set with the following property: whenever the expansion of a non-terminal  $A$  in  $G'$  contains a pair of occurrences  $P_1, P_2$  forming a relevant co-occurrence, there are occurrences of  $S_1, S_2$  in the proximity. This will allow us to preprocess the non-terminals of  $G'$  for occurrences of the strings in the set and use them to detect  $b$ -close relevant co-occurrences of  $P_1, P_2$ .

Consider two tries,  $T_{pre}$  and  $T_{suf}$ : For each production of  $G'$  of the form  $A \rightarrow BC$ , we store  $\overline{C}$  in  $T_{suf}$  and  $\text{rev}(\overline{B})$  in  $T_{pre}$ . For each production of the form  $A \rightarrow B^k$ , we store  $\overline{B}$ ,  $\overline{B^2}$ ,  $\overline{B^{k-2}}$ , and  $\overline{B^{k-1}}$  in  $T_{suf}$  and the reverses of those strings in  $T_{pre}$ . For  $j \in \{1, 2\}$  and  $s \in \text{Splits}'(G', P_j)$  define  $S_j(s) = \text{rev}(U)V$ , where  $U$  is the label of the locus of  $\text{rev}(P_j[\dots s])$  in  $T_{pre}$  and  $V$  is the label of the locus of  $P_j(s\dots)$  in  $T_{suf}$ . Let  $l_j(s) = |\text{rev}(U)|$  and  $\Delta_j(s) = l_j(s) - s$ .

Consider a non-terminal  $A$  such that its expansion  $\overline{A}$  contains a relevant co-occurrence  $(q_1, q_2)$  of  $P_1, P_2$ .

**Claim 2.24.** *There exists  $s \in \text{Splits}'(G', P_2)$  such that  $p_2 = q_2 - \Delta_2(s)$  is an occurrence of  $S_2(s)$  in  $\overline{A}$  and  $[p_2, p_2 + |S_2(s)|] \supseteq [q_2, q_2 + |P_2|]$ .*

*Proof.* Below we show that there exists a descendant  $A'$  of  $A$  and a split  $s \in \text{Splits}'(G', P_2)$  such that either  $\text{rev}(P_2[\dots s])$  is a prefix of  $\text{rev}(\text{head}(A'))$  and  $P_2(s\dots)$  is a prefix of  $\text{tail}(A')$ , or  $A'$  is associated with a rule  $A' \rightarrow (B')^k$ ,  $\text{rev}(P_2[\dots s])$  is a prefix of  $\text{rev}((B')^2)$  and  $P_2(s\dots)$  is a prefix of  $\overline{(B')^{k-2}}$ . The claim follows by the definition of  $T_{pre}$ ,  $T_{suf}$ , and  $S_2(s)$ .

If  $q_2$  is relevant in  $\overline{A}$ , there exists a split  $s \in \text{Splits}'(G', P_2)$  such that  $\text{rev}(P_2[\dots s])$  is a prefix of  $\text{rev}(\text{head}(A))$  and  $P_2(s\dots)$  is a prefix of  $\text{tail}(A)$  by definition. If  $q_2$  is not relevant, then  $q_2 \geq |\text{head}(A)|$  by the definition of a co-occurrence. By Claim 2.6, there is a descendant  $A'$  of  $A$  corresponding to a substring  $\overline{A}[\ell\dots r]$  for which either  $(q_2 - \ell)$  is relevant (and then we can repeat the argument above), or  $A'$  is associated with a rule  $A' \rightarrow (B')^k$  and  $(q_2 - \ell) - k' \cdot |\overline{B'}|$  is relevant, for some  $0 \leq k' \leq k$ . Consider the latter case. If  $A' = A$ , then  $k' = 1$ , as otherwise  $q_1 < q'_2 = q_2 - |\overline{B'}| < q_2$  is an occurrence of  $P_2$  in  $\overline{A}$  contradicting the definition of a co-occurrence (recall that  $(q_1, q_2)$  is a relevant co-occurrence and hence by definition  $q_1 < |\text{head}(A)|$ ), and therefore  $s = |(\overline{B'})^2| - q_2 + \ell \in \text{Splits}'(G', P_2)$ ,  $\text{rev}(P_2[\dots s])$  is a prefix of  $\text{rev}((B')^2)$  and  $P_2(s\dots)$  is a prefix of  $\overline{(B')^{k-2}}$ . If  $A' \neq A$ , then we can analogously conclude that  $k' = 0$ , which implies  $s = |\overline{B'}| - q_2 + \ell \in \text{Splits}'(G', P_2)$ ,  $\text{rev}(P_2[\dots s])$  is a prefix of  $\text{rev}(B')$  and  $P_2(s\dots)$  is a prefix of  $\overline{(B')^{k-1}}$ .  $\square$

As the definition of a co-occurrence is not symmetric,  $q_1$  does not enjoy the same property. However, a similar claim can be shown:

**Lemma 2.25.** *There exists  $s \in \text{Splits}'(G', P_1)$  and an occurrence  $p_1$  of  $S_1(s)$  in  $\overline{A}$  such that  $[p_1, p_1 + |S_1(s)|] \supseteq [q_1, q_1 + |P_1|]$  and at least one of the following holds:*

1.  $q_1 - \Delta_1(s)$  is an occurrence of  $S_1(s)$ ;
2.  $q_2$  is a relevant occurrence of  $P_2$  in  $\overline{A}$ , the period of  $S_1(s)$  equals the period  $\pi_1$  of  $P_1$ , and there exists an integer  $k$  such that  $p_1 = q_1 - \Delta_1(s) - \pi_1 \cdot k$  and  $q_2 + \pi_1 - 1 \leq p_1 + |S_1(s)| - 1 \leq q_2 + |P_2| - 1$ .

*Proof.* If  $q_1$  is a relevant occurrence of  $P_1$  in  $A$  with a split  $s \in \text{Splits}'(G', P_1)$ , then  $\text{rev}(P_1[\dots s])$  is a prefix of  $\text{rev}(\text{head}(A))$  and  $P_1(s \dots)$  is a prefix of  $\text{tail}(A)$  and therefore the first case holds by the definition of  $T_{pre}$  and  $T_{suf}$ .

Otherwise, by Claim 2.6, there is a descendant  $A'$  of  $\text{head}(A)$  corresponding to a substring  $\overline{A}[\ell \dots r]$  for which either  $(q_1 - \ell)$  is relevant (and then we can repeat the argument above), or  $A'$  is associated with a rule  $A' \rightarrow (B')^k$  and  $(q_1 - \ell) - k' \cdot |\overline{B'}|$ , for some  $0 \leq k' \leq k$ , is a relevant occurrence of  $P_1$  in  $\overline{A'}$  with a split  $s \in \text{Splits}'(G', P_1)$ . Consider the latter case. We must have (1)  $q_1 + |P_1| - 1 + |\overline{B'}| \geq r$  or (2)  $q_1 + |\overline{B'}| - 1 \geq q_2$ , because if both inequalities do not hold, then  $q_1 < q_1 + |\overline{B'}| \leq q_2$  is an occurrence of  $P_1$  in  $\overline{A}$ , which contradicts the definition of a co-occurrence. Additionally, if (1) holds, then by definition there exists a split  $s' \in \text{Splits}'(G', P_1)$  (which might be different from the split  $s$  above) such that  $\text{rev}(P_1[\dots s'])$  is a prefix of  $\text{rev}((B')^{r-1})$  and  $P_1(s' \dots)$  is a prefix of  $\overline{B'}$  and we fall into the first case of the lemma.

From now on, assume that (2) holds and (1) does not. Since  $q_1 + |\overline{B'}| \leq r \leq |\text{head}(A)|$  and  $(q_1, q_2)$  is a relevant co-occurrence,  $q_2$  must be a relevant occurrence of  $P_2$  in  $\overline{A}$ . If  $|P_1| - s \leq |(\overline{B'})^2|$ , then  $\text{rev}(P_1[\dots s])$  is a prefix of  $\text{rev}(\overline{B'})$  and  $P_1(s \dots)$  is a prefix of  $(\overline{B'})^2$  and therefore  $q_1 - \Delta_1(s)$  is an occurrence of  $S_1(s)$ . Otherwise, by Fine and Wilf's periodicity lemma [7], the periods of  $\overline{A'}$ ,  $P_1$ , and  $S_1(s)$  are equal, since  $P_1$  and hence  $S_1(s)$  span at least two periods of  $\overline{A'}$ . By periodicity,  $S_1(s)$  occurs at positions  $q_1 - \Delta_1(s) - |\overline{B'}| \cdot k$  of  $\overline{A}$ . Let  $p_1$  be the leftmost of these positions which satisfies  $p_1 + |S_1(s)| - 1 \geq q_1 + |P_1| - 1$ . This position is well-defined as (1) does not hold, and furthermore  $[q_1, q_1 + |P_1|) \subseteq [p_1, p_1 + |S_1(s)|)$  as  $s \leq l_1(s)$  and  $|S_1(s)| - l_1(s) \geq |P_1| - s$ . We have  $p_1 = q_1 - \Delta_1(s) - \pi_1 \cdot k''$  for some integer  $k''$  (as  $|\overline{B'}|$  is a multiple of  $\pi_1$ ), and  $q_2 + \pi_1 - 1 \leq q_1 + 2|\overline{B'}| - 1 \leq q_1 + |P_1| - 1 \leq p_1 + |S_1(s)| - 1 \leq r < q_2 + |P_2| - 1$ , where the last inequality holds as  $q_2$  is a relevant occurrence in  $\overline{A}$ . The claim of the lemma follows.  $\square$

We summarize Claim 2.24 and Lemma 2.25:

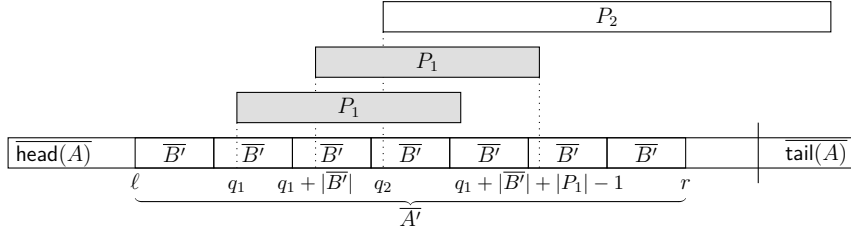
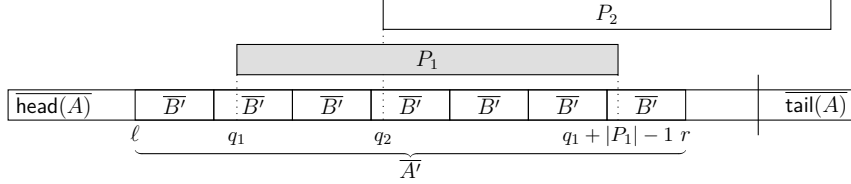
**Corollary 2.26.** *Let  $(q_1, q_2)$  be a co-occurrence of  $P_1, P_2$  in the expansion of a non-terminal  $A$ . There exist splits  $s_1 \in \text{Splits}'(G', P_1)$ ,  $s_2 \in \text{Splits}'(G', P_2)$  and occurrences  $p_1$  of  $S_1(s_1)$  and  $p_2$  of  $S_2(s_2)$ , where  $[p_1, p_1 + |S_1(s_1)|) \supseteq [q_1, q_1 + |P_1|)$  and  $[p_2, p_2 + |S_2(s_2)|) \supseteq [q_2, q_2 + |P_2|)$ , such that at least one of the following holds:*

1. *The occurrence  $p_1$  is either relevant or  $p_1 + |S_1(s_1)| - 1 \leq |\text{head}(A)|$ . The occurrence  $p_2$  is either relevant or  $p_2 > |\text{head}(A)|$ . Additionally,  $p_1 = q_1 - \Delta_1(s_1)$  and  $p_2 = q_2 - \Delta_2(s_2)$ .*
2. *The occurrence  $p_2$  is relevant and  $p_1 \leq |\text{head}(A)|$ . Additionally,  $p_2 = q_2 - \Delta_2(s_2)$ , the period of  $S_1(s)$  equals the period  $\pi_1$  of  $P_1$ , and there exists an integer  $k$  such that  $p_1 = q_1 - \Delta_1(s_1) - \pi_1 \cdot k$  and  $p_2 + \pi_1 - 1 \leq p_1 + |S_1(s_1)| - 1 \leq p_2 + |S_2(s_2)| - 1$ .*

The reverse observation holds as well:

**Observation 2.27.** *If  $p_j$  is an occurrence of  $S_j(s)$  in  $\overline{A}$ ,  $j = 1, 2$ , then  $q_j = p_j + \Delta_j(s)$  is an occurrence of  $P_j$ . Furthermore, if  $S_1(s)$  is periodic with period  $\pi_1$ , then  $q_1 + \pi_1 \cdot k$ ,  $0 \leq k \leq \lfloor (|S_1(s)| - q_1 - |P_1|) / \pi_1 \rfloor$ , are occurrences of  $P_1$  in  $\overline{A}$ .*

Finally, the following trivial observation will be important for upper bounding the time complexity of our query algorithm:


 (a) If neither (1) nor (2), then  $(q_1, q_2)$  is not consecutive.


(b) (1) holds and (2) does not.

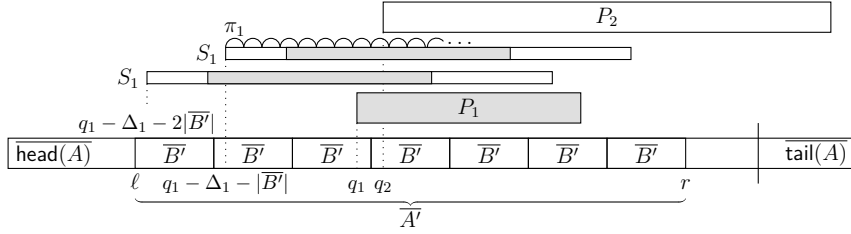

 (c) (2) holds, (1) does not, and  $|P_1| - s \geq \overline{(B')^2}$ .

Figure 2.2: Subcases of Lemma 2.25.

**Observation 2.28.** *If a string contains a pair of occurrences  $(q_1, q_2)$  of  $P_1$  and  $P_2$  such that  $0 \leq q_2 - q_1 \leq b$ , then it contains a  $b$ -close co-occurrence of  $P_1$  and  $P_2$ .*

## 4.2 Index

The first part of the index is the data structure of Theorem 2.15 and the index of Christiansen et al. [329]:

**Fact 2.29** ([329, Introduction and Theorem 6.12]). *There is a  $O(g \log^2 N)$ -space data structure that can find the output occurrences of any pattern  $P[1 \dots m]$  in  $S$  in time  $O(m + \text{output})$ .*

The second part of the index are the tries  $T_{pre}$  and  $T_{suf}$ , augmented as explained below. Consider a quadruple  $(u_1, u_2, v_1, v_2)$ , where  $u_1$  and  $u_2$  are nodes of  $T_{pre}$  and  $v_1$  and  $v_2$  are nodes of  $T_{suf}$ . Let  $U_1, U_2, V_1, V_2$  be the labels of  $u_1, u_2, v_1, v_2$ , respectively. Define  $S_1 = \text{rev}(U_1)V_1$  and  $S_2 = \text{rev}(U_2)V_2$ , and let  $l_1 = |\text{rev}(U_1)|$  and  $l_2 = |\text{rev}(U_2)|$ .

First, we store a binary search tree  $\mathcal{T}_1(u_1, u_2, v_1, v_2)$  that for each non-terminal  $A$  contains at most six integers  $d = p_2 - p_1$ , where  $p_1, p_2$  are occurrences of  $S_1, S_2$  in  $\overline{A}$ , satisfying at least one of the below:

1.  $p_1$  is the rightmost occurrence of  $S_1$  such that  $p_1 + |S_1| - 1 < |\text{head}(A)|$  and  $p_2$  is the leftmost occurrence of  $S_2$  such that  $p_2 \geq |\text{head}(A)|$ ;

2.  $p_1$  is a relevant occurrence of  $S_1$  with a split  $l_1$  and  $p_2$  is the leftmost occurrence of  $S_2$  such that  $p_2 \geq |\text{head}(A)|$ ;
3.  $p_1$  is a relevant occurrence of  $S_1$  with a split  $l_1$ ,  $p_2$  is a relevant occurrence of  $S_2$  with a split  $l_2$ ;
4.  $p_2$  is a relevant occurrence of  $S_2$  with a split  $l_2$  and  $p_1$  is the rightmost occurrence of  $S_1$  such that  $p_1 + |S_1| - 1 < p_2$ ;
5.  $p_2$  is a relevant occurrence of  $S_2$  with a split  $l_2$  and  $p_1$  is the leftmost or second leftmost occurrence of  $S_1$  in  $\text{head}(A)$  such that  $p_1 < p_2 \leq p_1 + |S_1| - 1 < p_2 + |S_2| - 1$ .

Second, we store a list of non-terminals  $\mathcal{L}(u_2, v_2)$  such that their expansion contains a relevant occurrence of  $S_2$  with a split  $l_2$ . Additionally, for every  $k \in [0, \log N]$ , we store, if defined:

1. The rightmost occurrence  $p_1$  of  $S_1$  in  $S_2$  such that  $p_1 + (|S_1| - 1) \leq l_2 - 2^k$ ;
2. The leftmost occurrence  $p'_1$  of  $S_1$  in  $S_2$  such that  $p'_1 \leq l_2 - 2^k \leq p'_1 + |S_1| - 1$ ;
3. The rightmost occurrence  $p''_1$  of  $S_1$  in  $S_2$  such that  $p''_1 \leq l_2 - 2^k \leq p''_1 + |S_1| - 1$ .

Finally, we compute and memorize the period  $\pi_1$  of  $S_1$ . If the period is well-defined (i.e.,  $S_1$  is periodic), we build a binary search tree  $\mathcal{T}_2(u_1, u_2, v_1, v_2)$ . Consider a non-terminal  $A$  containing a relevant occurrence  $p_2$  of  $S_2$  with a split  $l_2$ . Let  $p_1$  be the leftmost occurrence of  $S_1$  such that  $p_1 \leq p_2 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$  and  $p'_1$  the rightmost. If  $p_1$  and  $p'_1$  exist ( $p_1$  might be equal to  $p'_1$ ) and  $p'_1 + |S_1| - 1 \geq p_2 + \pi_1 - 1$ , we add an integer  $(p'_1 - p_1)/\pi_1$  to the tree and associate it with  $A$ . We also memorize a number  $\text{ov}(S_1, S_2) = p_2 - p'_1$ , which does not depend on  $A$  by Corollary 2.2 and therefore is well-defined (it corresponds to the longest prefix of  $S_2$  periodic with period  $\pi_1$ ).

**Claim 2.30.** *The data structure occupies  $O(g^5 \log^5 N)$  space.*

*Proof.* The data structure of Theorem 2.15 occupies  $O(g^2 \log^4 N)$  space. The index of Christiansen et al. occupies  $O(g \log^2 N)$  space. The tries, by Lemma 2.9, use  $O(g') = O(g \log N)$  space. There are  $O((g')^4)$  quadruples  $(u_1, u_2, v_1, v_2)$  and for each of them the trees take  $O(g')$  space. The arrays of occurrences of  $S_1$  in  $S_2$  use  $O(\log N)$  space. Therefore, overall the data structure uses  $O(g^5 \log^5 N)$  space.  $\square$

### 4.3 Query

Recall that a query consists of two strings  $P_1, P_2$  of length at most  $m$  each and an integer  $b$ , and we must find all  $b$ -close co-occurrences of  $P_1, P_2$  in  $S$ , let output be their number.

We start by checking whether  $P_2$  occurs in  $P_1$  using a linear-time and constant-space pattern matching algorithm such as [36]. If it is, let  $q_2$  be the position of the first occurrence. If  $q_2 > b$ , then there are no  $b$ -close co-occurrences of  $P_1, P_2$  in  $S$ . Otherwise, to find all  $b$ -close co-occurrences of  $P_1, P_2$  in  $S$  (that *always* consist of an occurrence of  $P_1$  in  $S$  and the first occurrence of  $P_2$  in  $P_1$ ), it suffices to find all occurrences of  $P_1$  in  $S$ , which we do using the index of Christiansen et al. [329] in time  $O(|P_1| + \text{output}) = O(m + \text{output})$ .

From now on, assume that  $P_2$  is not a substring of  $P_1$ . Let  $\mathcal{N}$  be the set of all non-terminals in  $G'$  such that their expansion contains a relevant  $b$ -close co-occurrence of  $P_1, P_2$ . By Claim 2.23,  $|\mathcal{N}| \leq \text{output}$ .



**Lemma 2.31.** *Assume that  $P_2$  is not a substring of  $P_1$ . One can retrieve in  $O(m + (1 + \text{output}) \log^3 N)$  time a set  $\mathcal{N}' \supset \mathcal{N}$ ,  $|\mathcal{N}'| = O(\text{output} \log N)$ .*

*Proof.* We start by computing  $\text{Splits}'(G', P_1)$  and  $\text{Splits}'(G', P_2)$  via Lemma 2.8 in time  $O((|P_1| + |P_2|) \log N) = O(m \log N)$  (or providing a certificate that either  $P_1$  or  $P_2$  does not occur in  $S$ , in which case there are no co-occurrences of  $P_1, P_2$  in  $S$  and we are done). Recall that  $|\text{Splits}'(G', P_1)|, |\text{Splits}'(G', P_2)| \in O(\log N)$ . For each fixed pair of splits  $s_1 \in \text{Splits}'(G', P_1)$ ,  $s_2 \in \text{Splits}'(G', P_2)$  and  $j \in \{1, 2\}$ , we compute the interval of strings in  $T_{pre}$  prefixed by  $\text{rev}(P_j[\dots s_j])$ , which corresponds to the locus  $u_j$  of  $\text{rev}(P_j[\dots s_j])$  in  $T_{pre}$  and the interval of strings in  $T_{suf}$  prefixed by  $P_j(s_j \dots)$ , which corresponds to the locus  $v_j$  of  $P_j(s_j \dots)$  in  $T_{suf}$ . Computing the intervals takes  $O(m + \log^2 N)$  time for all the splits by Lemma 2.9. Consider the strings  $S_1 = \text{rev}(U_1)V_1$  and  $S_2 = \text{rev}(U_2)V_2$ , where  $U_1, U_2, V_1, V_2$  are the labels of  $u_1, v_1, u_2, v_2$ , respectively. Let  $l_1 = |\text{rev}(U_1)|$ ,  $\Delta_1 = l_1 - s_1$ ,  $l_2 = |\text{rev}(U_2)|$ ,  $\Delta_2 = l_2 - s_2$ , and  $\Delta = \Delta_1 - \Delta_2$ .

Consider a relevant co-occurrence  $(q_1, q_2)$  of  $P_1, P_2$  in the expansion of a non-terminal  $A$ . By Corollary 2.26, the relevant co-occurrence  $q_1, q_2$  imply the existence of occurrences  $p_1, p_2$  of  $S_1, S_2$  such that  $[p_1, p_1 + |S_1|] \supseteq [q_1, q_1 + |P_1|]$  and  $[p_2, p_2 + |S_2|] \supseteq [q_2, q_2 + |P_2|]$ . Our index must treat both cases of Corollary 2.26. We consider eight subcases defined in Fig. 2.3, which describe all possible locations of  $p_1$  and  $p_2$ .

Subcases (1.1)-(1.4). To retrieve the non-terminals, we query  $\mathcal{T}_1(u_1, u_2, v_1, v_2)$  to find all integers that belong to the range  $[\Delta, \Delta + b]$  (and the corresponding non-terminals). Recall that, for each non-terminal  $A$ , the tree stores an integer  $d = p_2 - p_1$ , where  $p_1$  is the starting position of an occurrence of  $S_1$  in  $\bar{A}$  and  $p_2$  of  $S_2$ . By Observation 2.27,  $p_1 + \Delta_1$  is an occurrence of  $P_1$  and  $p_2 + \Delta_2$  is an occurrence of  $P_2$ . The distance between them is in  $[0, b]$  iff  $d \in [\Delta, \Delta + b]$ . By Observation 2.28, each retrieved non-terminal contains a close co-occurrence of  $(q_1, q_2)$ . On the other hand, if  $\bar{A}$  contains a co-occurrence  $(q_1, q_2)$  corresponding to one Subcases (1.1)-(1.4), then by Corollary 2.26,  $p_1 = q_1 - \Delta_1$  is an occurrence of  $S_1$  and  $p_2 = q_2 - \Delta_2$  is an occurrence of  $S_2$  and by construction  $\mathcal{T}_1(u_1, u_2, v_1, v_2)$  stores an integer  $d = p_2 - p_1$ . Therefore, the query retrieves all non-terminals corresponding to Subcases (1.1)-(1.4).

Subcases (1.5) and (2.1). We must decide whether an occurrence of  $P_1$  in  $S_2$  forms a  $b$ -close co-occurrence with the occurrence  $\Delta_2$  of  $P_2$  in  $S_2$ , and if so, report all non-terminals such that their expansion contains a relevant co-occurrence of  $S_2$  with a split  $l_2$ , which are exactly the non-terminals stored in the list  $\mathcal{L}(u_2, v_2)$ . Let  $k = \lceil \log(s_2) \rceil$ . Recall that the index stores the following information for  $k$ :

1.  $p_1$ , the rightmost occurrence of  $S_1$  in  $S_2$  such that  $p_1 + (|S_1| - 1) \leq l_2 - 2^k$ ;
2.  $p'_1$ , the leftmost occurrence of  $S_1$  in  $S_2$  such that  $p'_1 \leq l_2 - 2^k \leq p_1 + (|S_1| - 1)$ ;
3.  $p''_1$ , the rightmost occurrence of  $S_1$  in  $S_2$  such that  $p''_1 \leq l_2 - 2^k \leq p'_1 + (|S_1| - 1)$ .

(See Fig. 2.4). By Observation 2.27, the occurrence  $p_1$  of  $S_1$  induces an occurrence  $q_1 = p_1 + \Delta_1$  of  $P_1$ . Furthermore, if  $S_1$  is periodic with period  $\pi_1$ , then  $q_1 + \pi_1 \cdot k$ ,  $0 \leq k \leq \lfloor (|S_1| - q_1 - |P_1|)/\pi_1 \rfloor$ , are also occurrences of  $P_1$ . One can decide whether the distance from any of these occurrences to  $q_2$  is in  $[0, b]$  in constant time, and if yes, then there  $S_2$  contains a  $b$ -close co-occurrence of  $P_1, P_2$  by Observation 2.28. Second, by Corollary 2.2, if  $S_1$  is not periodic, then there are no occurrences of  $S_1$  between  $p'_1$  and  $p''_1$  and  $p'_1, p''_1$

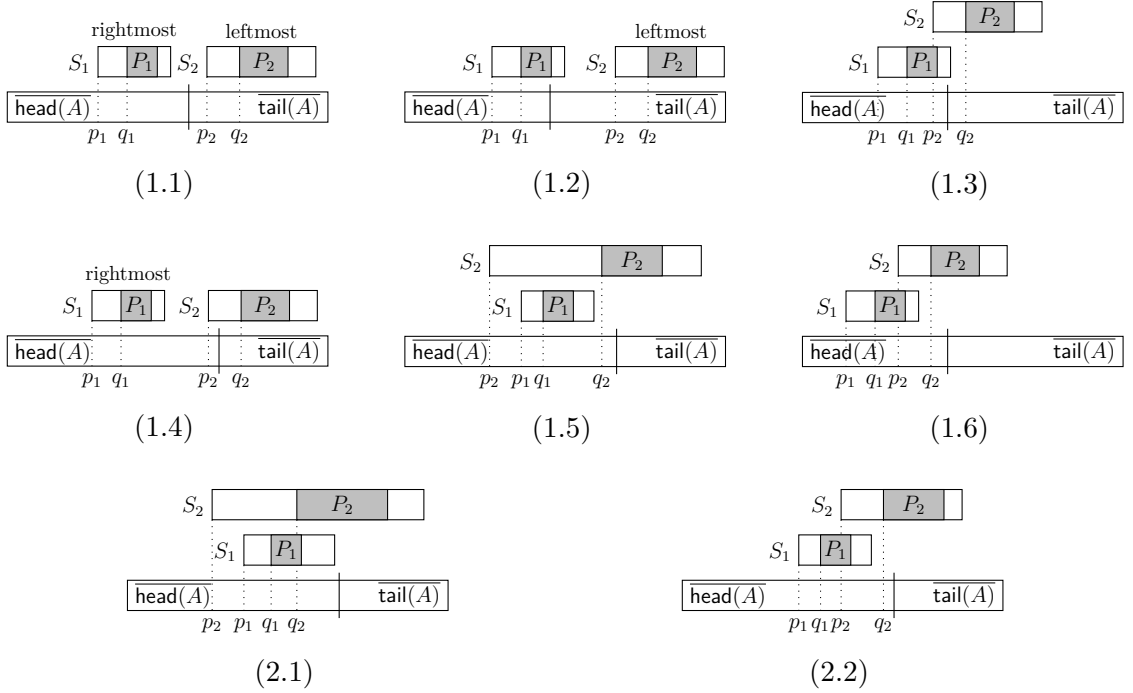


Figure 2.3: Assume that  $S_1$  does not contain  $S_2$ . The figure shows all possible locations of occurrences  $p_1, p_2$  of  $S_1, S_2$  in  $\bar{A}$ . **In Case 1 of Corollary 2.26**, there are six subcases: (1.1)  $p_1 + |S_1| - 1 \leq |\text{head}(A)|$ ,  $p_2 > |\text{head}(A)|$ ; (1.2)  $p_1$  is a relevant occurrence of  $S_1$ ,  $p_2 > |\text{head}(A)|$ ; (1.3)  $p_1, p_2$  are relevant; (1.4)  $p_2$  is relevant,  $p_1 + |S_1| - 1 \leq p_2$ ; (1.5)  $p_2$  is relevant,  $p_2 < p_1 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$ ; (1.6)  $p_2$  is relevant,  $p_1 < p_2 < p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$ . By the definition of a co-occurrence and by Observation 2.27, in Subcases (1.1) and (1.4)  $p_1$  must be as far to the right as possible, and in Subcases (1.1) and (1.2)  $p_2$  must be as far to the left as possible. **In Case 2**, there are two subcases: (2.1)  $p_2$  is relevant and  $p_2 \leq p_1 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$ ; (2.2)  $p_2$  is relevant and  $p_1 < p_2 < p_2 + \pi_1 - 1 \leq p_1 + |S_1| - 1$ , where  $\pi_1$  is the period of  $S_1$ . In all subcases,  $q_2 = p_2 + \Delta_2$ . In Subcases (1.1)-(1.6)  $q_1 = p_1 + \Delta_1$  and in Subcases (2.1) and (2.2)  $q_1 = p_1 + \Delta_1 + k \cdot \pi_1$  for some integer  $k$ .

by Observation 2.27 induce occurrences  $p'_1 + \Delta_1, p''_1 + \Delta_1$  of  $P_1$ . Otherwise, there are occurrences of  $P_1$  in every position  $p'_1 + \Delta_1 + k \cdot \pi_1$ ,  $0 \leq k \leq \lfloor (|S_1| + p'_1 - |P_1| - p'_1) / \pi_1 \rfloor$ . Similarly, we can decide whether the distance from any of them to the occurrence  $\Delta_2$  of  $P_2$  in  $S_2$  is in  $[0, b]$  in constant time. Finally, let  $q_1$  be the rightmost occurrence of  $P_1$  in  $S_2$  in the interval  $[l_2 - 2^k + 1, \Delta_2]$ . We extract  $S_2(l_2 - 2^k, \Delta_2 + |P_2|)$  via Fact 2.12 and search for  $q_1$  using a linear-time pattern matching algorithm for  $P_1$ , which takes  $O(|P_1| + |P_2|) = O(m)$  time. If  $0 \leq \Delta_2 - q_1 \leq b$ , then there is a  $b$ -close co-occurrence of  $P_1, P_2$  in  $S_2$ . Correctness follows from Corollary 2.26, Observation 2.27 and Observation 2.28.

Subcase (2.2). Let  $\pi_1$  be the period of  $S_1$ . We retrieve the non-terminals associated with the integers  $q \in \mathcal{T}_2(u_1, u_2, v_1, v_2)$  such that the intersection of an interval  $I = [a, b]$  and  $[\ell, q]$  is non-empty, where

$$a = \lceil (\Delta - \text{ov}(S_1, S_2)) / \pi_1 \rceil, b = \lfloor (\Delta - \text{ov}(S_1, S_2) + b) / \pi_1 \rfloor \text{ and } \ell = -\lfloor (|S_1| - |P_1| - \Delta_1) / \pi_1 \rfloor$$

(See the description of the index for the definition of  $\text{ov}(S_1, S_2)$ ). As  $\ell$  is fixed, we can implement the query via at most one binary tree search: If  $b \leq \ell$ , the output is empty, if

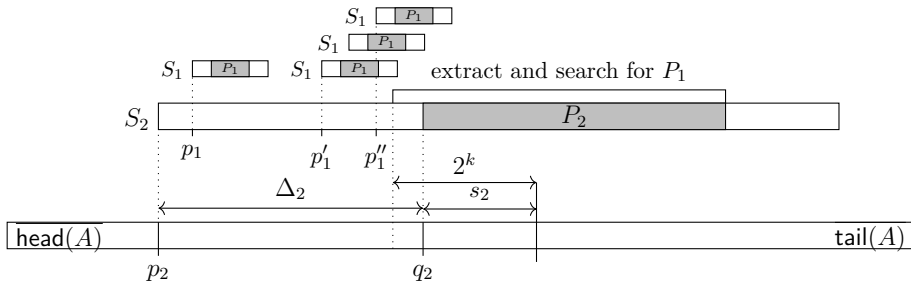


Figure 2.4: Query algorithm for Subcases (1.5) and (2.1).

$a \leq \ell \leq b$ , we must output all integers, and if  $\ell \leq a$ , we must output all  $q \geq b$ . Let us now explain why the algorithm is correct. Consider a non-terminal  $A$  for which  $\mathcal{T}_2(u_1, u_2, v_1, v_2)$  stores an integer  $q$ . By construction,  $\overline{A}$  contains a relevant occurrence of  $S_2$  with a split  $l_2$ . A position  $p_1 = |\text{head}(A)| - l_2 - \text{ov}(S_1, S_2) - q \cdot \pi_1$  is the leftmost occurrence of  $S_1$  in  $\overline{A}$  such that  $p_1 \leq p_2 \leq p_1 + |S_1| - 1$  and  $p_2 = \lfloor |\text{head}(A)| - l_2 - \text{ov}(S_1, S_2) \rfloor$  the rightmost. Consequently, there is an occurrence  $q_1 = \lfloor |\text{head}(A)| - l_2 - \text{ov}(S_1, S_2) - q' \cdot \pi_1 + \Delta_1 \rfloor$  of  $P_1$  for each  $-\lfloor (|S_1| - |P_1| - \Delta_1)/\pi_1 \rfloor \leq q' \leq q$ . The occurrence of  $S_2$  implies that  $q_2 = \lfloor |\text{head}(A)| - s_2 \rfloor$  is an occurrence of  $P_2$ . We have  $0 \leq q_2 - q_1 = q' \cdot \pi_1 + \text{ov}(S_1, S_2) - \Delta \leq b$  iff  $\Delta - \text{ov}(S_1, S_2) \leq q' \cdot \pi_1 \leq \Delta - \text{ov}(S_1, S_2) + b$ , which is equivalent to  $[\ell, q] \cap I \neq \emptyset$ . It follows that we retrieve every non-terminal corresponding to Subcase (2.2). On the other hand, by Observation 2.28, the expansion of each retrieved non-terminal contains a  $b$ -close co-occurrence of  $P_1, P_2$ .

Subcase (1.6). We argue that we have already reported all non-terminals corresponding to this subcase and there is nothing left to do. Consider a non-terminal  $A$  such that its expansion contains a relevant occurrence  $p_2$  of  $S_2$ . If there are at most two occurrences  $p_1$  of  $S_1$  such that  $p_1 \leq p_2 \leq p_1 + |S_1| - 1 \leq p_2 + |S_2| - 1$ , we will treat them when we query  $\mathcal{T}_1(u_1, u_2, v_1, v_2)$  (Subcases (1.1)-(1.4)). Otherwise, by Corollary 2.2,  $S_1$  is periodic and there is an occurrence  $p'_1$  of  $S_1$  such that  $p'_1 \leq p_2 < p_2 + \pi_1 \leq p_1 + |S_1| - 1 < p_2 + |S_2| - 1$ . The non-terminals corresponding to this case are reported when we query  $\mathcal{T}_2(u_1, u_2, v_1, v_2)$  (Subcase (2.2)).

Time complexity. As shown above, the algorithm reports a set  $\mathcal{N}' \supset \mathcal{N}$  of non-terminals and each non-terminal in  $\mathcal{N}'$  contains a  $b$ -close co-occurrence. By Claim 2.23 and since the height of  $G'$  is  $h = O(\log N)$ , we have  $|\mathcal{N}'| = O(\text{output} \log N)$ . Furthermore, for a fixed pair of splits of  $P_1, P_2$ , each non-terminal in  $\mathcal{N}'$  can be reported a constant number of times. Since  $|\text{Splits}'(G', P_1)| \cdot |\text{Splits}'(G', P_2)| = O(\log^2 N)$ , the total size of the output is  $|\mathcal{N}'| \cdot O(\log^2 N) = O(\text{output} \cdot \log^3 N)$ . We therefore obtain that the running time of the algorithm is  $O(m + \log^3 N + \text{output} \log^3 N) = O(m + (1 + \text{output}) \log^3 N)$  as desired.  $\square$

Once we have retrieved the set  $\mathcal{N}'$ , we find all  $b$ -close relevant co-occurrences for each of the non-terminals in  $\mathcal{N}'$  using Theorem 2.15. In fact, our algorithm acts naively and computes *all* relevant co-occurrences for a non-terminal in  $\mathcal{N}'$ , and then selects those that are  $b$ -close. By case inspection, one can show that a relevant co-occurrence for a non-terminal  $A$  always consists of an occurrence of  $P_2$  that is either relevant or the leftmost in  $\overline{\text{tail}(A)}$ , and a preceding occurrence of  $P_1$ . Intuitively, this allows to compute all relevant co-occurrences efficiently and guarantees that their number is small. Formally, we show

the following claim:

**Lemma 2.32.** *Assume that  $P_2$  is not a substring of  $P_1$ . After  $O(m \log N + \log^2 N)$ -time preprocessing, the data structure of Theorem 2.15 allows to compute all  $b$ -close relevant co-occurrences of  $P_1, P_2$  in the expansion of a given non-terminal  $A$  in time  $O(\log^3 N \log \log N)$ .*

*Proof.* We preprocess  $P_1, P_2$  in  $O(m \log N + \log^2 N)$  time as explained in Theorem 2.15. Upon receiving a non-terminal  $A$ , we compute the leftmost and the rightmost occurrences of  $P_1, P_2$  in  $\overline{\text{head}(A)}$  and  $\overline{\text{tail}(A)}$ , as well as a set  $\Pi_1$  of all relevant occurrences of  $P_1$  in  $\overline{A}$  and a set  $\Pi_2$  of all relevant occurrences of  $P_2$  in  $\overline{A}$  via Claim 2.20. We will compute all relevant co-occurrences in  $\overline{A}$ , selecting those of them that are  $b$ -close is then trivial. As  $q_1 \leq q_2$  by definition, each relevant co-occurrence  $(q_1, q_2)$  of  $P_1, P_2$  in  $\overline{A}$  falls under one of the following categories:

1.  $q_1$  is a relevant occurrence of  $P_1$  in  $\overline{A}$  and  $q_2$  is a relevant occurrence of  $P_2$  in  $\overline{A}$  (i.e.  $q_1 \in \Pi_1, q_2 \in \Pi_2$ ). To check whether a pair  $q_1 \in \Pi_1, q_2 \in \Pi_2$  forms a co-occurrence of  $P_1, P_2$  in  $\overline{A}$ , we must check whether there is an occurrence  $q$  of either  $P_1$  or  $P_2$  between  $q_1$  and  $q_2$ . The occurrence  $q$  can only be the rightmost occurrence  $r_q$  of  $P_2$  in  $\overline{\text{head}(A)}$ , the leftmost occurrence  $l_q$  of  $P_1$  in  $\overline{\text{tail}(A)}$ , or an occurrence in  $\Pi_1 \cup \Pi_2$ . Consequently, we can find all co-occurrences in this category by merging two (sorted) sets:  $\Pi_1 \cup \{l_q\}$  and  $\{r_q\} \cup \Pi_2$ , which can be done in  $O(2 + |\Pi_1 \cup \Pi_2|)$  time.
2.  $1 \leq q_1 \leq q_1 + |P_1| - 1 \leq |\overline{\text{head}(A)}|$  and  $|\overline{\text{head}(A)}| < q_2 \leq q_2 + |P_2| - 1$ . In this case,  $q_1$  must be the rightmost occurrence of  $P_1$  in  $\overline{\text{head}(A)}$  and  $q_2$  the leftmost occurrence in  $\overline{\text{tail}(A)}$ ,  $q_1 \leq q_2$ , and there must be no occurrence  $q \in \Pi_1 \cup \Pi_2$  such that  $q_1 \leq q \leq q_2$ . Therefore, if there is a co-occurrence in this category, we can retrieve it in  $O(|\Pi_1 \cup \Pi_2|)$  time.
3.  $q_1$  is a relevant occurrence of  $P_1$  in  $\overline{A}$  (i.e.  $q_1 \in \Pi_1$ ) and  $|\overline{\text{head}(A)}| < q_2 \leq q_2 + |P_2| - 1$ . In this case,  $q_1$  must be the rightmost occurrence in  $\Pi_1$  and  $q_2$  the leftmost occurrence of  $P_2$  in  $\overline{\text{tail}(A)}$ , and there should be no occurrence from  $\Pi_2$  between  $q_1$  and  $q_2$ . Therefore, if there is a co-occurrence in this category, we can find it in  $O(|\Pi_1 \cup \Pi_2|)$  time.
4.  $q_1 \leq q_1 + |P_1| - 1 \leq |\overline{\text{head}(A)}|$  and  $q_2$  is a relevant occurrence of  $P_2$  in  $\overline{A}$  (i.e.  $q_2 \in \Pi_2$ ). First, consider the leftmost occurrence in  $q_2 \in \Pi_2$ . We find the rightmost occurrence  $q_1 \leq q_2$  of  $P_1$  in  $\overline{A}$  via a predecessor query. The pair  $(q_1, q_2)$  is a co-occurrence iff the rightmost occurrence of  $P_2$  in  $\overline{\text{head}(A)}$  is smaller than  $q_1$ , which can be checked in constant time. Second, we consider the remaining occurrences in  $\Pi_2$ . Let  $q'_2$  be the leftmost one. We begin by computing the preceding occurrence  $q'_1$  of  $P_1$  via a predecessor query and if  $q_2 \leq q'_1$ , output the resulting co-occurrence. If  $\Pi_2 = \{q_2, q'_2\}$ , we are done. Otherwise, by Corollary 2.2, the occurrences in  $\Pi_2 \setminus \{q_2\}$  form an arithmetic progression with difference equal to the period of  $P_2$  (as all of them contain the position  $|\overline{\text{head}(A)}|$ ). Furthermore, as  $P_1$  does not contain  $P_2$ , the occurrence of  $P_1$  preceding  $q'_2$  belongs to the periodic region formed by the relevant occurrences of  $P_2$ . Therefore, all the remaining co-occurrences can be obtained from

the co-occurrence for  $q'_2$  by shifting them by the period. In total, this step takes  $O(|\Pi_2| + \log^3 N \log \log N)$  time.

□

A part of the index of Christiansen et al. [329] is a pruned copy of the parse tree of  $G'$ . They showed how to traverse the tree to report all occurrences of a pattern, given its relevant occurrences in the non-terminals. By using essentially the same algorithm, we can report all  $b$ -close co-occurrences in amortized constant time per co-occurrence, which concludes the proof of Theorem 2.1.

**Lemma 2.33.** *Assume that  $P_2$  is not a substring of  $P_1$ . One can compute all  $b$ -close co-occurrences of  $P_1, P_2$  in  $S$  in time  $O(m + (1 + \text{output}) \cdot \log^4 N \log \log N)$ .*

*Proof.* During the preprocessing, we prune the parse tree: First, for each non-terminal  $B$ , all but the first node labeled by  $B$  in the preorder is converted into a leaf and its subtree is pruned. For each node  $v$  labeled by a non-terminal  $B$ , we store  $\text{anc}(v)$ , the nearest ancestor  $u$  of  $v$  labeled by  $A$  such that  $u$  is the root or  $A$  labels more than one node in the pruned tree. Second, for every node labeled by a non-terminal  $A$  associated with a rule  $A \rightarrow B^k$ , we replace its  $k - 1$  rightmost children with a leaf labeled by  $B^{k-1}$ . We call the resulting tree *the pruned parse tree* and for each node  $v$  labeled by a non-terminal  $B$  store  $\text{next}(v)$ , the next node labeled by  $B$  in preorder, if there is one. As every non-terminal labels at most one internal node of the pruned parse tree and every node has at most two children, it occupies  $O(g')$  space.

When the algorithm of Lemma 2.31 outputs  $A \in \mathcal{N}'$ , we compute all relevant co-occurrences  $(q_1, q_2)$  in  $\bar{A}$  in time  $O(\log^3 N \log \log N)$  using Lemma 2.32 and select those which satisfy  $q_2 - q_1 \leq b$ .

Fix a  $b$ -close relevant co-occurrence  $(q_1, q_2)$  in  $\bar{A}$ . If  $A$  is associated with a rule  $A \rightarrow BC$ , construct a set  $\text{output}(A) := \{(q_1, q_2)\}$ , and otherwise if  $A$  is associated with a rule  $A \rightarrow B^k$ ,

$$\text{output}(A) := \{(q_1 + i \cdot |\bar{B}|, q_2 + i \cdot |\bar{B}|) : 0 \leq i \leq \lfloor (|\bar{A}| - q_2 - |P_2| + 1) / |\bar{B}| \rfloor\}$$

Suppose that  $A$  labels nodes  $v_1, v_2, \dots, v_k$  of the unpruned parse tree of  $G'$  (by construction  $v_1$  is not pruned and we assimilate it to the corresponding node in the pruned parse tree). If  $W$  is a set of co-occurrences, denote for brevity  $W + \delta = \{(q_1 + \delta, q_2 + \delta) : (q_1, q_2) \in W\}$ . Below we show an algorithm that generates a set  $\mathcal{S} = \cup_i \text{output}(A) + \text{off}(v_i)$  that contains all secondary  $b$ -close co-occurrences due to  $(q_1, q_2)$ .

We traverse the pruned parse tree, while maintaining a priority queue. The queue is initialized to contain the first node in the preorder labeled by  $A$  together with  $\text{output}(A)$ . Until the priority queue is empty, pop a node  $v$  and a set  $W$  of co-occurrences of  $P_1, P_2$  in the expansion of its label, and perform the following steps:

- Reporting step: If  $v$  is the root, report  $W$ ;
- Next node step: If  $\text{next}(v)$  is defined, push  $(\text{next}(v), W + \text{off}(\text{next}(v)) - \text{off}(v))$ ;
- Sibling step: If  $v$  is labeled by a non-terminal  $B$  and its sibling by  $B^k$ , for some integer  $k$ , then  $W := \cup_{0 \leq i \leq k} W + i \cdot |\bar{B}|$

- Ancestor step: Push to the queue  $(\text{anc}(v), W + \text{off}(\text{anc}(v)) - \text{off}(v))$ .

By construction and as every node is connected with the root by a path of anc links, the algorithm generates all co-occurrences in  $\mathcal{S}$ .

Let us show that it reports every co-occurrence in  $\mathcal{S}$  at most once. Assume that a co-occurrence was reported twice by two different sequences of nodes. Let  $u, w$  be the first nodes in the sequences where the same co-occurrence  $(q_1, q_2) \in \mathcal{S}$  was created. Note that ancestor steps do not create new occurrences, but only update already created ones. Therefore, we have only three possible cases for  $u, w$ : either we generated  $(q_1, q_2)$  because the node equals to  $v_1$ , or we applied the next node step, or we applied the sibling step. We cannot have  $u = w = v_1$ , as  $v_1$  is considered by the algorithm only once due to the fact that the preorder number of the current node increases both after the next node step and the ancestor step. Suppose now that  $u = v_1$  and that we applied the next node or the sibling step to  $w$ . Neither  $\text{next}(w)$  nor the siblings of  $w$  can be in the subtree of  $u$ . On the other hand,  $u$  cannot be neither in the subtree of  $\text{next}(w)$  nor in the subtrees of the siblings of  $w$  (they are pruned). Therefore, in this case we could not generate  $(q_1, q_2)$  both for  $u$  and  $w$ . If we generated  $(q_1, q_2)$  by applying the next node step to  $u$  and the next node step to  $w$ , then by the choice of  $u, w$ , we have  $\text{next}(u) \neq \text{next}(w)$ . Furthermore, neither  $\text{next}(u)$  can be an ancestor of  $\text{next}(w)$ , nor vice versa, as the subtrees of  $\text{next}(u)$  and  $\text{next}(w)$  are pruned, and therefore we could not create  $(q_1, q_2)$  both for  $u$  and  $w$ . If  $(q_1, q_2)$  was generated by applying the next node step to  $u$  and the sibling step to  $w$ , then  $u \neq w$  by definition and as the subtree of  $\text{next}(u)$  is pruned, neither  $w$  nor its siblings can be descendants of  $u$ . On the other hand, the subtrees and the siblings of  $w$  are pruned, and cannot be ancestors of  $\text{next}(w)$ . A contradiction. Finally, consider the case when we apply the sibling step both to  $u$  and  $w$ . In this case, the subtrees of the siblings of  $u$  and  $w$  are pruned by construction, and neither  $u$  can belong to a subtree of a sibling of  $w$ , nor vice versa, and therefore we cannot generate  $(q_1, q_2)$  both for  $u$  and  $w$ . All remaining cases are symmetrical.

The time complexity follows: Lemma 2.31 takes time  $O(m + (1 + \text{output}) \cdot \log^3 N)$ ; applying Lemma 2.32 to every non-terminal in  $\mathcal{N}'$  takes time  $O(\text{output} \cdot \log^4 N \log \log N)$ ; and maintaining the queue and reporting the co-occurrences takes  $O(\text{output})$  time as at every step we can charge the time needed to update the queue on newly created co-occurrences.  $\square$



## Joint work

This chapter is a joint work with Pawel Gawrychowski, Tatiana Starikovskaya, and Teresa Anna Steiner, unpublished.

Originating from the work of Navarro and Thankanchan [TCS 2016], the problem of consecutive pattern matching is a variant of the fundamental pattern matching problem, where one is given a text and a pair of patterns  $p_1, p_2$ , and must compute consecutive occurrences of  $p_1, p_2$  in the text. Assuming that the text is given as a straight-line program of size  $g$ , we develop an algorithm that computes all consecutive occurrences of  $p_1, p_2$  in optimal  $O(g + |p_1| + |p_2| + \text{output})$  time. As a corollary, we also derive an algorithm that reports all co-occurrences separated by a distance  $d \in [a, b]$  in  $O(g + |p_1| + |p_2| + \text{output})$  time and an algorithm that reports the top- $k$  closest co-occurrences in  $O(g + |p_1| + |p_2| + k)$  time.

## 1 Introduction

In the classical pattern matching problem, one is given a pattern and a text, and must find all substrings of the text equal to the pattern. However, considering potential applications, it is advantageous to enable queries beyond simply identifying exact matches of a given pattern in the preprocessed text.

Recently, Navarro and Thankanchan [228] suggested a generalisation of the pattern matching problem, where in addition to the pattern and the text one is given two integers  $a, b$ , and must find all pairs of consecutive occurrences of the pattern in the text separated by a distance  $d \in [a, b]$ . They showed that there is a  $O(n \log n)$ -space index for this problem with optimal query time  $O(m + \text{output})$ , where  $n$  is the length of the text and  $m$  of the pattern.

Following their work, indexing for consecutive occurrences is receiving growing attention in the literature [348, 349, 364].

Bille et al. [349] considered an even more general case of the problem, where a query consists of two different patterns  $p_1, p_2$  of total length  $m$  and two integers  $a, b$ , and one must find all pairs of consecutive occurrences of  $p_1, p_2$  in the text separated by a distance  $d \in [a, b]$ . For reporting the occurrences, they showed a linear-space data structure with  $\tilde{O}(m + n^{2/3} \text{output})$  query time. On the other hand, by reduction from the set intersection problem, they showed a lower bound suggesting that achieving query time better than  $\tilde{O}(m + \sqrt{n})$  for indexes of size  $\tilde{O}(n)$  is impossible, even if  $a = 0$  is fixed.

Gawrychowski et al. [364] showed that one can circumvent this lower bound in the case  $a = 0$  assuming that the text is very compressible: assuming that the text is represented by a straight-line program (SLP) of size  $g$ , they showed a  $\tilde{O}(g^5)$ -space data structure with  $\tilde{O}(m + \text{output})$  query time, where  $m$  is the total length of the patterns.

Unfortunately, the above upper bounds, despite their theoretical interest, are still far from providing an efficient solution. Motivated by this, we study the dual variant of the problem, where one must process the text and the patterns simultaneously. Note that if the text is uncompressed and has length  $n$ , the problem can be solved by classical online pattern matching



in  $O(n + m + \text{output})$  time by keeping the most recent occurrences of  $p_1$  and  $p_2$ . In this work, we show that a similar complexity can be achieved for very compressible texts by extending the ideas of [353]:

**Theorem 3.1.** *Given a text of length  $n$  represented by an SLP  $G$  of size  $g$  and patterns  $p_1, p_2$  of total length  $m$ , all consecutive occurrences of  $p_1, p_2$  in the text can be found in  $O(g + m + \text{output})$  time assuming the word-RAM model with a machine word of size  $w = \Omega(\log n)$ .*

Using similar techniques, we derive an algorithm to output all consecutive occurrences with a bounded distance between them:

**Corollary 3.2.** *Given a text of length  $n$  represented by an SLP  $G$  of size  $g$  and patterns  $p_1, p_2$  of total length  $m$ , all consecutive occurrences of  $p_1, p_2$  in the text separated by a distance  $d \in [a, b]$  can be found in  $O(g + m + \text{output})$  time assuming the word-RAM model with a machine word of size  $w = \Omega(\log n)$ .*

Finally, our techniques allow to derive an efficient solution for the variant of the problem suggested by Bille et al. [348], where one must report the top- $k$  consecutive occurrences of  $p$  with smallest distances between them.

**Corollary 3.3.** *Given an integer  $k$ , a text of length  $n$  represented by an SLP  $G$  of size  $g$  and patterns  $p_1, p_2$  of total length  $m$ , the  $k$  closest consecutive occurrences of  $p_1, p_2$  in the text can be found in  $O(g + m + k)$  time assuming the word-RAM model with a machine word of size  $w = \Omega(\log n)$ .*

## 2 Preliminaries

A *string*  $s$  of length  $|s| = n$  is a sequence  $s[0]s[1] \dots s[n-1]$  of characters from an integer alphabet  $\Sigma$  that can be sorted in  $O(m + g)$  time. A *substring* of a string  $s$  is a pair  $(i, j)$  where  $0 \leq i \leq j < |s|$  and is identified with the string  $s[i \dots j] = s[i] \dots s[j]$ . We also use the notation  $s[i \dots j)$  and  $s(i \dots j]$  which stand for the substring  $s[i \dots j-1]$  and  $s[i-1 \dots j]$  respectively. We say that a substring  $s[i \dots j]$  is fully contained in another substring  $s[i' \dots j']$  if  $i' \leq i \leq j \leq j'$ . We call a substring  $s[0 \dots i]$  a *prefix* of  $s$  and use a simplified notation  $s[\dots i]$ , and a substring  $s[i \dots n-1]$  a *suffix* of  $s$  denoted by  $s[i \dots ]$ . We say that  $x$  occurs in  $s$  at position  $i$  if  $x = s[i \dots i + |x|)$ , alternatively we say  $i$  is an occurrence of  $x$  in  $s$ . Additionally, an occurrence  $i$  is fully included in a substring  $f$  of  $s$  if  $s[i \dots i + |x|)$  is fully included in  $f$ .

An occurrence  $k_1$  of  $p_1$  together with an occurrence  $k_2$  of  $p_2$  form a *consecutive occurrence* (co-occurrence)  $(k_1, k_2)$  of the strings  $p_1, p_2$  in a string  $s$  if there are no occurrences of  $p_1$  in  $[k_1, k_2]$  and no occurrences of  $p_2$  in  $[k_1, k_2]$ . The distance  $k_2 - k_1$  is sometimes referred to as a *gap*.

An integer  $\pi$  is a *period* of a string  $s$  of length  $n$  if  $s[i] = s[i + \pi]$  for all  $i = 0, \dots, n-1-\pi$ . We say that  $s$  is *periodic* if its smallest period, referred to as *the period* of  $s$ , is at most  $|s|/2$ . We also exploit the well-known corollary of the Fine and Wilf's periodicity lemma [7]:

**Corollary 3.4.** *Let  $x, y$  be strings such that  $|x| \leq 2|y|$ . If there are at least three occurrences of  $y$  in  $x$ , then all occurrences of  $y$  in  $x$  form an arithmetic progression with difference equal to the period of  $y$ .*

**Proposition 3.5.** *One can preprocess a string  $p$  of length  $m$  in  $O(m)$  time and space to maintain the following queries in constant time: Given a substring  $(i, j)$ , find the leftmost and the rightmost occurrences of  $p[i \dots j]$  in  $p$ , as well as the total number of occurrences. Given two substrings  $(i, j)$  and  $(k, l)$ , compute the longest common prefix and longest common suffix between  $p[i \dots j]$  and  $p[k \dots l]$ .*

*Proof.* We assume the reader to be familiar with suffix trees. We build the suffix tree of  $p$  in  $O(m)$  time and space. Belazzougui et al. [328] showed that the suffix tree can be preprocessed in linear time so that, given a substring  $(i, j)$ , one can find the node  $u$  of the suffix tree labeled by  $p[i...j]$  in constant time. The leaves of the subtree of  $u$  correspond to the occurrences of  $p[i...j]$  in  $p$ . For each node, we can precompute in linear time, the size of its subtree, the leftmost and rightmost occurrences of its label. This is done by simply traversing the tree from the bottom to the top and propagating the information. We also preprocess the suffix trees in linear time so that they can find the lowest common ancestor between two nodes in constant time [66]. Given two substrings  $(i, j)$  and  $(k, l)$  the length of the label of their common ancestor is their longest common prefix. Analogously, by having the same suffix tree built for the reversed string, we can compute the longest common suffix.  $\square$

**Corollary 3.6** (of Corollary 3.4 and Proposition 3.5). *One can preprocess a string  $p$  of length  $m$  in  $O(m)$  time and space to maintain the following queries in constant time: Given a substring  $(i, j)$ , such that  $j - i \geq m/2$ , one can output the arithmetic progression of the occurrences of  $p[i...j]$  in  $p$  in constant time.*

## 2.1 Grammars

**Definition 3.7** (Straight-line program [69]). *A straight-line program (SLP) is a context-free grammar (CFG) consisting of a set of non-terminals  $X_1, \dots, X_q$ , a set of terminals, an initial symbol  $X_q$ , and a set of productions, satisfying the following properties:*

- *A production consists of a left-hand side and a right-hand side, where the left-hand side is a non-terminal  $X_i$  and the right-hand side is a sequence  $X_j X_k$ , where  $j, k < i$ , or a terminal;*
- *Every non-terminal is on the left-hand side of exactly one production.*

The *expansion*  $\bar{S}$  of a sequence of terminals and non-terminals  $S$  is the string that is obtained by iteratively replacing non-terminals by the right-hand sides in the respective productions, until only terminals remain. We say that  $G$  *represents* the expansion  $\bar{G}$  of its initial symbol.

**Definition 3.8** (Parse tree). *The parse tree of a SLP is defined as follows:*

- *The root is labeled by the initial symbol;*
- *Each internal node is labeled by a non-terminal;*
- *If  $S$  is the expansion of the initial symbol, then the  $i^{\text{th}}$  leaf of the parse tree is labeled by a terminal  $S[i]$ ;*
- *A node labeled with a non-terminal  $A$  that is associated with a production  $A \rightarrow BC$  has two children labeled by  $B$  and  $C$ , respectively.*

The *size* of a grammar is the total size of all right-hand sides of all productions. The *height* of a grammar is the height of the parse tree.

## 3 Boundary Information

For the duration of this section, fix a pattern  $p$  of length  $m$ . For a string  $s$ , let  $\text{prefix}_p(s)$  be the longest prefix of  $s$  which is a suffix of  $p$  and  $\text{suffix}_p(s)$  the longest suffix of  $s$  which is a prefix

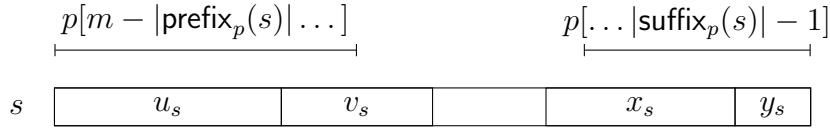


Figure 3.1: A  $p$ -boundary information for a string  $s$  that is not a substring of  $p$ .

of  $p$ . If  $s$  occurs in  $p$  at position  $i$ , meaning  $s = p[i \dots i + |p| - 1]$ , then we define  $p$ -substring information for  $s$  as  $(i, i + |p| - 1)$ , and otherwise  $p$ -substring information for  $s$  is undefined. For a string  $s$ , a  $p$ -boundary information is defined as follows:

1. If  $s$  occurs in  $p$ , then it is simply  $p$ -substring information for  $s$ ;
2. Otherwise, it is two substrings of  $p$ ,  $u_s$  and  $v_s$  such that  $\text{prefix}_p(s)$  is a prefix of  $u_s v_s$  which in turn is a prefix of  $s$  ( $p$ -prefix information), and two substrings of  $p$ ,  $x_s, y_s$  such that  $\text{suffix}_p(s)$  is a suffix of  $x_s y_s$  which in turn is a suffix of  $s$  ( $p$ -suffix information). (See Fig. 3.1.)

For a string  $s$ , multiple  $p$ -boundary information can be constructed. One way to construct one is recursively: for two strings  $s, t$ , assume to be given  $p$ -boundary information for  $s, t$ , Algorithm 3.1 (first described in [353]) correctly constructs a  $p$ -boundary information for  $c = st$ .

---

**Algorithm 3.1** A boundary information of  $c = st$

---

1. If  $s$  is a substring of  $p$  and  $t$  is not, then the  $p$ -suffix information of  $c$  is the  $p$ -suffix information of  $t$  and we define the  $p$ -prefix information for  $c$  as follows:
    - (a) If  $su_t$  is a substring of  $p$ , then  $u_c = su_t$  and  $v_c = v_t$ ;
    - (b) Otherwise,  $u_c = s$  and  $v_c = u_t$ .
  2. If  $t$  is a substring of  $p$  and  $s$  is not, then the  $p$ -prefix information of  $c$  is the  $p$ -prefix information of  $s$  and we define the suffix information for  $c$  as follows:
    - (a) If  $y_s t$  is a substring of  $p$  then  $y_c = y_s t$  and  $x_c = x_s$ ;
    - (b) Otherwise,  $x_c = y_s$  and  $y_c = t$ ;
  3. If  $s$  and  $t$  are both substrings of  $p$ , and  $c$  is a substring  $p[i \dots j]$  of  $p$ , then the  $p$ -boundary information is  $p$ -substring information for  $c$ , and we define it equal to  $(i, j)$ . Otherwise, we put  $u_c = x_c = s$  and  $v_c = y_c = t$ .
  4. If neither  $s$  nor  $t$  is a substring of  $p$ , then one can take  $p$ -prefix information for  $c$  equal to  $p$ -prefix information for  $s$  and  $p$ -suffix information to  $p$ -suffix information for  $t$ .
- 

**Definition 3.9** (Crossing occurrences). *Let  $s, t$  be two strings. We say that a position  $i$  is a crossing occurrence of  $p$  in a string  $c = st$  if  $i$  is an occurrence of  $p$  in  $c$  and  $i \leq |s| \leq i + |p| - 1$ . By extension,  $i$  is a crossing occurrence of  $p$  in the expansion  $\bar{A}$  of a non-terminal  $A$  of a grammar  $G$ , or simply a crossing occurrence for  $A$ , if  $A$  is associated with a production  $A \rightarrow BC$  and  $i \leq |B| \leq i + |p| - 1$ .*

Ganardi and Gawrychowski [353] showed that given a  $p$ -boundary information of two strings  $s, t$  one can efficiently decide whether there is a crossing occurrence of  $p$  in  $c = st$ . By slightly modifying their solution, we can report all such occurrences:

**Lemma 3.10.** *Assume to be given a  $p$ -boundary information of strings  $s_k, t_k$  for  $k \in [1, q]$ . One can compute all crossing occurrences of  $p$  in all strings  $s_k t_k$ , for  $k \in [1, q]$ , in  $O(q + m)$  time. For each  $k$ , the output is represented as an arithmetic progression.*

We defer the proof of the lemma to Section 6. The proof and our algorithm exploit the following fact:

**Fact 3.11** (see [353, Lemma 2.2, Theorem 1.3]). *Let  $w$  be the size of the machine word. A string  $p$  of length  $m$  can be preprocessed in  $O(m)$  time so that:*

- $q$  substring concatenation queries on  $p$  can be answered in  $O(q + m/w)$  time. A substring concatenation query on a string  $p$  asks: Given two substrings  $(i, j)$  and  $(k, \ell)$ , let  $u = p[i \dots j]$  and  $v = p[k \dots \ell]$  of  $p$ , check whether  $uv$  occurs in  $p$  and, if so, return an occurrence;
- Given  $q$  substrings  $u_1, \dots, u_q$  of  $p$  one can compute  $\text{prefix}_p(u_i)$  and  $\text{suffix}_p(u_i)$  in  $O(q + m/w)$  time.

## 4 Compressed Consecutive Pattern Matching

We are now ready to prove Theorem 3.1. Recall that the text has length  $n$  and is represented by an SLP  $G$  of size  $g \ll n$ , and the patterns  $p_1, p_2$  have total length  $m$ .

### 4.1 Computing Boundary Information and Crossing Occurrences

We first use a linear-time pattern matching algorithm (e.g. the Knuth-Morris-Pratt algorithm [16]) to check whether  $p_2$  is a substring of  $p_1$ . If it is, then every co-occurrence of  $p_1, p_2$  in the text is a pair  $(i, i + \ell)$ , where  $i$  is an occurrence of  $p_1$  in the text and  $\ell$  is the leftmost occurrence of  $p_2$  in  $p_1$ . (By definition, there is no occurrence of  $p_2$  in  $[i, i + \ell)$ . Note also that there cannot exist an occurrence  $i < i' \leq i + \ell$  of  $p_1$ , because then  $i + \ell - i' < \ell$  would be an occurrence of  $p_2$  in  $p_1$ , a contradiction with the choice of  $\ell$ .) In other words, to find all co-occurrences of  $p_1, p_2$  in the text it suffices to find all occurrences of  $p_1$  in the text, which can be done in  $O(g + m + \text{output})$  time [353].

Below we assume that  $p_2$  is not a substring of  $p_1$ . Define an array  $P$  such that for every  $j \in [0, |p_2| - 1]$ ,  $P[j]$  is the rightmost occurrence of  $p_1$  in  $p_2$  to the left of  $j$  if there is one, else  $-1$ . We call  $P$  the *predecessor array*.  $P$  can be computed in  $O(m)$  time by a linear-time pattern matching algorithm.

By [287], we can restructure the SLP  $G$  representing the text in  $O(g)$  time to ensure that its height is  $O(\log n)$ , while its size increases by only a constant factor.

For every symbol  $A$  of  $G$  (a non-terminal or a terminal) associated with a production  $A \rightarrow BC$ , we compute:

1. a  $p_1$ -boundary information and a  $p_2$ -boundary information for  $\overline{A}$ ;
2. All crossing occurrences of  $p_1$  and  $p_2$  for  $A$ ;
3. The rightmost and the leftmost occurrences of  $p_1$  and  $p_2$  in  $\overline{A}$ ;

4. Furthermore, if  $p_2$ -suffix information for  $\bar{A}$  is  $(x_A, y_A)$ , then we compute:

- (a) a  $p_1$ -boundary information of  $x_A$  and  $y_A$ , which we refer to as *secondary boundary information*;
- (b) All crossing occurrences of  $p_1$  for  $x_A, y_A$ ;
- (c) The rightmost occurrence of  $p_1$  in  $\bar{A}$  starting before  $x_A$ .

**Proposition 3.12.** *There is a  $O(g)$ -time algorithm that computes boundary and secondary boundary information for all symbols of  $G$ .*

*Proof.* We first compute boundary information. For all terminals, it suffices to check if the characters occur in  $p$ , if they do occur, let  $i$  be one of their occurrences, we can define the  $p$ -substring information to be  $(i, i)$ , else we can define  $p$ -prefix information and the  $p$ -suffix information to be empty strings. Let the  $k$ -th level  $L_k$  of  $G$  be the set of its symbols whose parse tree has height  $k$ . We apply Algorithm 3.1 to compute boundary information for the symbols of each level in turn, starting from level 0. Processing  $L_k$  takes  $|L_k|$  substring concatenation queries and  $O(|L_k|)$  extra time. Since the height of  $G$  is  $O(\log n)$ , by Fact 3.11 we obtain  $O(\sum_k(|L_k| + m/w)) = O(g + m)$  total time.

The secondary boundary information is computed by applying Algorithm 3.1 on the substrings constituting the boundary information. In more detail, note that for any non-terminal  $A$  of  $G$  associated with a production  $A \rightarrow BC$  Algorithm 3.1 computes the boundary information of  $\bar{A}$  by either copying the boundary information of  $\bar{B}$  or  $\bar{C}$  or by concatenating the substrings constituting the boundary information of  $\bar{B}$  and  $\bar{C}$  a constant number of times. It follows that in total there are  $O(|L_k|)$  copying and concatenation operations at level  $k$ . For each concatenation operation, we apply Algorithm 3.1 to update the boundary information. Processing  $L_k$  hence takes  $O(|L_k|)$  substring concatenation queries and  $O(|L_k|)$  extra time. As above, this leads to  $O(\sum_k(|L_k| + m/w)) = O(g + m)$  total time. □

We now apply Lemma 3.10 to compute all crossing occurrences for the non-terminals of  $G$  in  $O(g + m)$  time. By a second application of Lemma 3.10, we compute, for every non-terminal  $A$  of  $G$ , all crossing occurrences for pairs  $x_A, y_A$ , which constitute  $p_2$ -suffix information for  $A$ , using the same amount of time.

**Proposition 3.13.** *Given the boundary information and the crossing occurrences for all symbols of  $G$ , one can compute the rightmost and leftmost occurrences of  $p_1$  and  $p_2$  in the expansion of every symbol of  $G$  in  $O(g)$  time.*

*Proof.* We explain how to compute the rightmost occurrences of  $p_1$ , the rest can be computed analogously. The algorithm processes the symbols of  $G$  bottom-up. Consider a symbol  $A$  of  $G$ . If it is a terminal, then we can find the rightmost occurrence of  $p_1$  in its expansion (if it exists) in  $O(1)$  time. Otherwise, assume that  $A$  is associated with a production  $A \rightarrow BC$ . If  $\bar{C}$  contains an occurrence of  $p_1$ , then the rightmost occurrence of  $p_1$  in  $\bar{A}$  is the rightmost occurrence of  $p_1$  in  $\bar{C}$  (and we have already computed it). Otherwise, if there are crossing occurrences of  $p_1$  for  $\bar{B}, \bar{C}$ , it is the rightmost such occurrence. And finally, if  $\bar{C}$  does not contain an occurrence of  $p_1$  and there are no crossing occurrences, then we copy the rightmost occurrence of  $p_1$  in  $\bar{B}$ . □

We will need the following auxiliary claim:

**Observation 3.14.** *Given an arithmetic progression by its starting position, difference, and length, we can find the predecessor of a number  $z$  in that progression in constant time.*

**Proposition 3.15.** *There is a  $O(g + m)$ -time algorithm that computes, for each symbol  $A$  of  $G$ , the rightmost occurrence of  $p_1$  in  $\bar{A}$  before  $x_A$ , where  $(x_A, y_A)$  is  $p_2$ -suffix information for  $\bar{A}$ .*

*Proof.* Let  $(x_B, y_B)$  (resp.,  $(x_C, y_C)$ ) be the  $p_2$ -suffix information for  $\bar{B}$  (resp.,  $\bar{C}$ ) that we computed using Algorithm 3.1. We review the cases of Algorithm 3.1:

In Cases 1 and 4, the  $p_2$ -suffix information of  $\bar{A}$  is the  $p_2$ -suffix information of  $\bar{C}$ . Therefore, the rightmost occurrence of  $p_1$  before  $x_A$  in  $\bar{A}$  is either the last occurrence before  $x_C = x_A$  in  $\bar{C}$ , or the rightmost crossing occurrence of  $p_1$  for  $A$ , or the rightmost occurrence of  $p_1$  in  $\bar{B}$ , and we can compute it in constant time.

In Case 3, either  $(x_A, y_A)$  is undefined or  $x_A = \bar{B}$ , and therefore the rightmost occurrence of  $p_1$  is undefined.

In Case 2a,  $x_A = x_B$ , and therefore the rightmost occurrence of  $p_1$  in  $\bar{A}$  before  $x_A$  is either the rightmost occurrence of  $p_1$  in  $\bar{B}$  before  $x_B$  or the rightmost crossing occurrence of  $p_1$  for  $\bar{A}$  that precedes  $x_B$ , which can be found in constant time by Observation 3.14.

In Case 2b, we have  $x_A = y_B$ . Thus, the rightmost occurrence before  $x_A$  equals to one of the following:

1. The rightmost crossing occurrence of  $p_1$  for  $x_B$  and  $y_B$ ;
2. The rightmost occurrence fully contained in  $x_B$ ;
3. The rightmost occurrence of  $p_1$  preceding  $x_B$ ;
4. The rightmost crossing occurrence of  $p_1$  for  $\bar{A}$  preceding  $x_B$ .

We can find the rightmost occurrence fully contained in  $x_B = p_2[i \dots j]$  by querying the predecessor array for  $j - |p_1| + 1$ , and the two other occurrences have been already computed.  $\square$

## 4.2 Reporting Co-occurrences

We now show how to quickly report the co-occurrences using the precomputed information.

**Proposition 3.16.** *Consider a symbol  $A$  of  $G$  associated with a production rule  $A \rightarrow BC$ . Let  $j < |\bar{B}| \leq j + p_2 - 1$  be an occurrence of  $p_2$  in  $\bar{A}$ . One can find the rightmost occurrence  $i \leq j$  of  $p_1$  such that  $i + |p_1| - 1 < |\bar{B}|$  in  $O(1)$  time.*

*Proof.* Let  $(x_B, y_B)$  be  $p_2$ -suffix information for  $\bar{B}$ . By definition,  $j$  belongs to  $x_B y_B$ . If  $j$  belongs to  $y_B$ , then  $i$  is the rightmost existing one of the following candidates:

1. The rightmost occurrence  $i' \leq j$  of  $p_1$  such that  $\bar{A}[i \dots i + |p_1|)$  is fully contained in  $y_B$  (which we can find in  $O(1)$  time using the predecessor array  $P$ );
2. The rightmost crossing occurrence  $i' \leq j$  of  $p_1$  for  $x_B, y_B$  (which we can find in  $O(1)$  time by Observation 3.14);
3. The rightmost occurrence of  $p_1$  that is fully in  $x_B$  (which we can find in  $O(1)$  time using the predecessor array  $P$ );
4. The rightmost occurrence of  $p_1$  in  $\bar{B}$  starting before  $x_B$  (which we have precomputed).

If  $j$  is in  $x_B$ , then  $i$  is the rightmost existing one of the following candidates:

- The rightmost occurrence  $i' \leq j$  of  $p_1$  such that  $\bar{A}$  is fully contained in  $x_B$  (which we can find in  $O(1)$  time using the predecessor array  $P$ );

- The rightmost crossing occurrence  $i' \leq j$  of  $p_1$  for  $x_B, y_B$  (which we can find in  $O(1)$  time by Observation 3.14);
- The rightmost occurrence of  $p_1$  in  $\overline{B}$  starting before  $x_B$ .

It follows that  $i$  can be computed in constant time. □

**Definition 3.17** (Primary co-occurrence). *Let  $A$  be a non-terminal of  $G$  associated with a production  $A \rightarrow BC$ . We say that a co-occurrence  $(i, j)$  of  $p_1, p_2$  in  $\overline{A}$  is primary if  $i \leq |\overline{B}| \leq j + |p_2| - 1$ .*

For a node  $u$  of the parse tree of  $G$ , denote by  $\text{off}(u)$  the number of leaves to the left of the subtree rooted at  $u$ .

**Observation 3.18.** *Assume that  $p_2$  is not a substring of  $p_1$ , and let  $(i, j)$  be a co-occurrence of  $p_1, p_2$  in the text. In the parse tree of  $G$ , there exists a unique node  $u$  such its label  $A$  is associated with a production  $A \rightarrow BC$ , and  $(i - \text{off}(u), j - \text{off}(u))$  is a primary co-occurrence of  $p_1, p_2$  in  $\overline{A}$ .*

**Lemma 3.19.** *Assume that  $p_2$  is not a substring of  $p_1$  and that we are given the information computed in Section 4.1. There is a  $O(g + m)$ -time algorithm that reports all primary co-occurrences of  $p_1$  and  $p_2$  in the expansions of the non-terminals of  $G$ . If there is more than one primary co-occurrence in the expansion of a non-terminal, they are output as a single arithmetic progression.*

*Proof.* Let  $A$  be a non-terminal associated with a production  $A \rightarrow BC$ . We consider three types of co-occurrences of  $p_1, p_2$  in  $\overline{A}$ :

1. The occurrence of  $p_1$  is fully contained in  $\overline{B}$  and the occurrence of  $p_2$  is fully contained in  $\overline{C}$ , or
2. The occurrence of  $p_1$  is a crossing occurrence for  $\overline{B}, \overline{C}$  and the occurrence of  $p_2$  is not, or
3. The occurrence of  $p_2$  is a crossing occurrence for  $\overline{B}, \overline{C}$ .

Let  $(i, j)$  be a co-occurrence of Type 1. It must have the property that  $i$  is the rightmost occurrence of  $p_1$  fully contained in  $\overline{A}[\dots |\overline{B}| - 1]$ ,  $j$  is the leftmost occurrence of  $p_2$  in  $\overline{A}[|\overline{B}| \dots]$ , and there are no occurrences of  $p_1, p_2$  in between. As we store all crossing occurrences for  $B, C$ , the rightmost occurrences of  $p_1, p_2$  in  $\overline{B}$  and the leftmost occurrences of  $p_1, p_2$  in  $\overline{C}$ , we can check whether  $(i, j)$  exists and compute it in  $O(1)$  time by Observation 3.14.

A co-occurrence  $(i, j)$  of Type 2 must satisfy the following properties:

1.  $j$  cannot be in  $\overline{A}[\dots |\overline{B}| - 1]$ , since it is not a crossing occurrence and  $p_2$  is not a substring of  $p_1$ ;
2. Since  $i$  and  $j$  are consecutive, and  $i$  is a crossing occurrence,  $i$  must be the rightmost crossing occurrence and  $j$  must be the leftmost occurrence of  $p_2$  in  $\overline{A}[|\overline{B}| \dots] = \overline{C}$ .

We retrieve  $i$ , the rightmost crossing occurrence of  $p_1$ , and  $j$ , the leftmost occurrence of  $p_2$  in  $\overline{A}[|\overline{B}| \dots]$ . It remains to check that there is no occurrence of  $p_1$  in  $(i, j]$  and no occurrence of  $p_2$  in  $[i, j)$ . If there is an occurrence of  $p_1$  in  $(i, j]$ , it can only be the leftmost occurrence of  $p_1$  in  $\overline{A}[|\overline{B}| \dots] = \overline{C}$ . If there is an occurrence of  $p_2$  in  $[i, j)$ , it can only be a crossing occurrence for  $A$ . Both conditions can be tested in constant time by Observation 3.14.

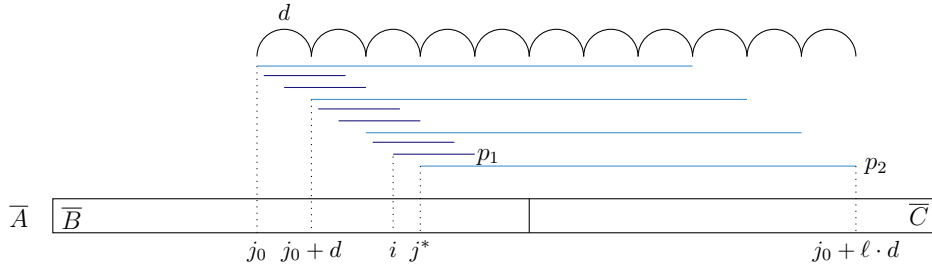


Figure 3.2: Co-occurrences with crossing occurrences of  $p_2$  forming an arithmetic progression.

For Type 3, consider two cases. First, consider the case when  $j$  is the leftmost crossing occurrence. Let  $i' \leq j$  be the rightmost occurrence of  $p_1$  such that  $i' + |p_1| - 1 < |\overline{B}|$ , which we can find in  $O(1)$  time by Proposition 3.16 and  $i''$  be the rightmost crossing occurrence of  $p_1$  that is at most  $j$ , which we can find in  $O(1)$  time as well using Observation 3.14. By definition, the only candidate for a co-occurrence containing  $j$  is  $(i = \max\{i', i''\}, j)$ . By construction of  $i'$  and  $i''$ , there can't be any occurrence of  $p_1$  in  $(i, j]$  and it suffices to check whether there are occurrences of  $p_2$  in  $[i, j)$ . If there is such an occurrence, it must be the rightmost occurrence of  $p_2$  in  $\overline{B}$ , and we can check if it is the case in constant time.

Consider now the case when  $j$  is not the leftmost crossing occurrence of  $p_2$  for  $A$ . By Corollary 2.2, all crossing occurrences of  $p_2$  form an arithmetic progression. Let  $j_0$  be the leftmost occurrence,  $d$  be the difference and  $\ell$  the length of this progression. By Corollary 2.2,  $\overline{A}[j_0 \dots j_0 + \ell \cdot d + |p_2| - 1]$  is periodic with period  $d$ . Let  $1 \leq k \leq \ell$  be the largest such that the occurrence  $j^* = j_0 + k \cdot d$  forms a co-occurrence with an occurrence  $i$  of  $p_1$ . By definition of a co-occurrence,  $j_0 \leq j^* - d < i \leq j^*$ . Furthermore, since  $p_2$  is not a substring of  $p_1$ , we have  $i + |p_1| - 1 < j^* + |p_2| - 1$ . Hence, by periodicity,  $(i - k' \cdot d, j^* - k' \cdot d)$  is a co-occurrence for all  $1 - k \leq k' \leq \ell - k$ . (In particular, by maximality of  $j^*$ , we have  $j^* = j_0 + \ell \cdot d$ .) Hence, it suffices to find the co-occurrence for  $k' = 1 - k$ , i.e. to find the occurrence of  $p_1$  preceding  $j_0 + d$ . This can be done in constant time similarly to the case above. This case is illustrated in Figure 3.2.

By Fact 3.11, the algorithm takes  $O(g + m)$  time. □

Finally, we report all co-occurrences of  $p_1, p_2$  in the text given the primary co-occurrences for the non-terminals of  $G$ . Using the approach of [329, Section 6.4], also used in Lemma 2.33 of the previous Chapter, it can be done in  $O(g + \text{output})$  time. Observation 3.18 guarantees that we report all co-occurrences.

## 5 Gapped and Top- $k$ Consecutive Pattern Matching

We now explain how to modify the algorithm to report only the co-occurrences with a bounded gap (Corollary 3.2) and only the top- $k$  co-occurrences in the text (Corollary 3.3).

*Bounded-gap co-occurrences.* We run the algorithm of Section 4 in  $O(g + m)$  time to generate a description of all primary co-occurrences (the elements of this description are single co-occurrences and arithmetic progressions of co-occurrences with a fixed gap) and select the elements of this description with a gap in  $[a, b]$ . For each selected element, we apply the approach of [329, Section 6.4] to generate all co-occurrences with a gap in the interval  $[a, b]$  in time  $O(g + m + \text{output})$ .



*Top- $k$  co-occurrences.* To report the top- $k$  co-occurrences, we first generate a description of all primary co-occurrences (the elements of this description are single co-occurrences and arithmetic progressions of co-occurrences with a fixed gap) in  $O(g + m)$  time as in Section 4. Second, we arrange the elements of the description in a heap in  $O(g)$  time sorted by the gaps. Then, conceptually, we attach to each node of the heap a path containing the secondary co-occurrences that originate from the element stored in the node. We finally select the  $k$  co-occurrences with smallest gaps in  $O(k)$  time using Frederickson’s heap selection algorithm [46]. The algorithm and its analysis requires the min-heap property, the fact that all nodes have constant degree, and to have quick access to the children of an already visited node. The first two properties are guaranteed by construction, and the method of [329, Section 6.4]) guarantees that the children of an already visited node can be accessed in constant amortised time.

## 6 Proof of Lemma 3.10

Consider two strings  $s, t$ . Let  $x, y$  be  $p$ -suffix information of  $s$  and  $u, v$   $p$ -prefix information of  $t$ . To find all crossing occurrences of  $p$  in a string  $st$ , it suffices to look at occurrences in  $xyuv$  as  $xy$  contains  $\text{suffix}_p(s)$  and  $uv$  contains  $\text{prefix}_p(t)$ .

We can also assume that  $x$  is a prefix of  $p$  and  $v$  a suffix of  $p$  because there is an occurrence of  $p$  in  $xyuv$  if and only if there is an occurrence of  $p$  in  $\text{suffix}_p(x)y\text{uprefix}_p(v)$ . Here and below, whenever we replace a string  $x$  with  $\text{suffix}_p(x)$  and or a string  $v$  with  $\text{prefix}_p(v)$ , we assume to compute them using Fact 3.11.

By Corollary 2.2, the crossing occurrences of  $p$  form a single arithmetic progression. We will consider several cases and for each case will report an arithmetic progression of occurrences, but in the end they can be merged into a single one. We repeatedly make use of the following procedure:

**Proposition 3.20.** *Let  $\ell$  be a prefix of  $p$ ,  $r$  a suffix of  $p$ , and  $c$  a concatenation of at most three substrings of  $p$ . One can report all occurrences of  $p$  in  $\ell c$  starting at positions  $i \leq |\ell|/2$  and all occurrences in  $cr$  ending at positions  $j \geq |c| + |r|/2$  using a constant number of longest common prefix and longest common suffix queries. The occurrences are output as an arithmetic progression.*

*Proof.* We show how to proceed for the occurrences in  $s = \ell c$ , the proof for the occurrences in  $cr$  is symmetric. Assume that there is an occurrence of  $p$  at position  $i \leq |\ell|/2$ . As  $\ell$  is a prefix of  $p$ ,  $i = \alpha \cdot d$ , where  $0 \leq \alpha \leq |\ell|/2d$  is an integer and  $d$  is the period of  $\ell$ .

After a classical shared linear-time preprocessing the period of any prefix of  $p$  can be extracted in  $O(1)$  time [16]. If  $d > |\ell|/2$ , then the only candidate is  $i = 0$  and we can test whether  $p$  occurs at this position using a constant number of longest common prefix and suffix queries. We now assume  $d \leq |\ell|/2$ . Let  $\alpha_{\max} \leq |\ell|/(2d)$  be the rightmost position such that  $\alpha_{\max} \cdot d + m \leq |s|$ . If there are none, then  $|s| < m$  and there are no occurrences of  $p$  in  $s$ . Let  $k \geq |\ell|$  be the rightmost position such that  $p[0 \dots k - 1]$  has period  $d$ ;  $k$  can be computed by one longest common prefix and suffix query on  $p$  and  $p[d \dots m - 1]$ . Furthermore, using  $O(1)$  more longest common prefix and suffix queries, one can check if  $p[0 \dots k - 1]$  occurs at position  $\alpha_{\max} \cdot d$  and if not, compute the first mismatching position.

Consider first the case where  $p[0 \dots k - 1]$  occurs at position  $\alpha_{\max} \cdot d$ . If  $k = m$ , then  $p$  occurs at every position  $\alpha \cdot d$  with  $0 \leq \alpha \leq \alpha_{\max}$ . If  $k < m$ , then  $p$  cannot occur at a position  $\alpha d$  with  $\alpha \leq \alpha_{\max}$  by the maximality of  $k$ . It suffices to check if  $p$  occurs at position  $\alpha_{\max} \cdot d$  using  $O(1)$  longest common prefix and suffix queries and report it accordingly. Now assume that  $p[0 \dots k - 1]$  does not occur at position  $\alpha_{\max} \cdot d$  and let  $p[0 \dots i - 1]$  be the longest prefix

starting at position  $\alpha_{\max} \cdot d$  in  $s$ , meaning  $p[i] \neq s[\alpha_{\max} \cdot d + i]$ . By construction,  $d$  is a period of  $s[0 \dots \alpha_{\max} \cdot d + i - 1]$ . Consequently, no occurrence of  $p[0 \dots k - 1]$  in  $\ell c$  can cross position  $i$ , meaning there is no occurrence of  $p$  in  $\ell c$  starting at position  $\alpha \cdot d$  with  $\alpha \cdot d + k > \alpha_{\max} \cdot d + i$ . Thus, occurrences can only be at positions  $\alpha \cdot d \leq \alpha_{\max} \cdot d + i - k$ . If  $k = m$ , by the  $d$ -periodicity of  $s[0 \dots \alpha_{\max} \cdot d + i - 1]$ , any such position is valid and we can report the occurrences as a single arithmetic progression. If  $k < m$ , the only possible candidate is the maximal  $\alpha \cdot d$  such that  $\alpha \cdot d + k < \alpha_{\max} \cdot d + i$  (by maximality of  $k$ ), and we can check whether there is an occurrence of  $p$  via  $O(1)$  longest common prefix and suffix queries as above, and report it accordingly.  $\square$

We start by applying Proposition 3.20 on  $\ell = x$  and  $c = yuv$  to report all occurrences starting before  $|x|/2$  and then apply it again on  $\ell = \text{suffix}_p(x[|x|/2 \dots])$  and  $c = yuv$ , which gives all occurrences of  $p$  in  $xyuv$  starting before  $3|x|/4$ . Symmetrically, we can report all occurrences of  $p$  ending after  $|xyu| + |v|/4$ .

It remains to report the occurrences of  $p$  in a string  $x'yu v'$ , where  $x' = \text{suffix}_p(x[3|x|/4 \dots])$  and  $v' = \text{prefix}_p(v[\dots |v|/4])$ . As  $|x|, |v| \leq m$ , we have  $|x'|, |v'| \leq m/4$ . For an occurrence  $i$  of  $p$  in  $x'yu v'$ , consider three (overlapping) cases:

1. The occurrence is fully contained in  $x'yu$ ;
2. The occurrence fully contains  $yu$ ;
3. The occurrence is fully contained in  $yu v'$ .

Consider Case 1. By applying Proposition 3.20 on  $c = x'y$  and  $r = u$ , we can assume that  $|u| \leq m/2$ . We then have three subcases: (a) either an occurrence of  $p$  is fully contained in  $x'y$ , or (b) it contains  $y$ , or (c) it is fully contained in  $yu$ . In Case 1(a), as  $|x'| \leq m/4$  and  $|y| \leq m$ , any occurrence of  $p$  in  $x'y$  ends in the second half of  $y$  and hence we can report all occurrences by applying Proposition 3.20 once to  $x'$  and  $\text{prefix}_p(y)$ . Case 1(c) is analogous. We repeat the argument for Case 3. Thus, it remains to report all occurrences of  $p$  in a string  $h = efg$ , where  $|e|, |g| \leq m/4$ ,  $e$  is a prefix of  $p$ ,  $g$  is a suffix of  $p$ , such they fully contain  $f$ .

Recall that  $f$  is given by its starting and ending positions in  $p$ , let  $f = p[i \dots j]$ . If the length of  $f$  is smaller than  $m/2$ , then  $|h| < m$  and there are no occurrences of  $p$  in  $h$ . Assume now that  $|f| \geq m/2$ .

By Corollary 3.6, after  $O(m)$ -time and  $O(m)$ -space preprocessing we can find the arithmetic progression of the occurrences of  $f$  in  $p$  in constant time. If there are only two occurrences, we test if they extend in  $e$  and  $g$  to an occurrence of  $p$  using two longest common prefix and suffix queries.

Assume now that there are at least three occurrences. Let  $p_{\text{mid}}$  be the minimal substring of  $p$  which contains all occurrences of  $f$ . By Corollary 2.2, the period of  $p_{\text{mid}}$  equals the period of  $f$ ,  $d$ . Let  $p = p_{\text{left}} p_{\text{mid}} p_{\text{right}}$ . Next, using two longest prefix and suffix queries, we compute the maximal substring  $f'$  of  $h$  that starts and ends with an occurrence of  $f$ . Namely, by Corollary 2.2, it suffices to check how far the periodicity in  $f$  extends beyond  $f$ :  $f'$  must be periodic with period  $d$ , must fully contain  $f$ , and must start at a position  $|e| - \alpha \cdot d$  and end at a position  $|e| + |f| + \alpha \cdot \beta$  for some integers  $\alpha, \beta$ . Let  $h = e' f' g'$ . By Corollary 2.2, the occurrences of  $f$  in  $h$  start at positions  $|e'| + \alpha \cdot d$  for integer  $0 \leq \alpha \leq (|f'| - |f|)/d$ . Hence, if  $p_{\text{left}}$  is not empty, then the only possible position where  $p$  can occur in  $h$  is  $|e'| - |p_{\text{left}}|$ , and we can test whether it is the case using  $O(1)$  longest common prefix and suffix queries. If  $p_{\text{right}}$  is not empty, then the only possible position where  $p$  can occur in  $h$  is  $|e'| + |f'| - |p_{\text{left}} p_{\text{mid}}|$ . Otherwise, if both  $p_{\text{left}}$  and  $p_{\text{right}}$  are empty, the arithmetic progression of occurrences of  $p = p_{\text{mid}}$  in  $h$  is simply  $|e'| + \alpha \cdot d$  for  $0 \leq \alpha \leq (|f'| - |p_{\text{mid}}|)/d$ .  $\square$



# Run Reporting Over General Alphabets

## Publication

This chapter corresponds to the extended version of the following publication: Jonas Ellert, Pawel Gawrychowski, and Garance Gourdel, “Optimal Square Detection Over General Alphabets”, in: *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, ed. by Nikhil Bansal and Viswanath Nagarajan, SIAM, 2023, pp. 5220–5242, DOI: 10.1137/1.9781611977554.ch189. The results specific to this extended version are the reporting of all runs in optimal time.

Squares (fragments of the form  $xx$ , for some string  $x$ ) are arguably the most natural type of repetition in strings. The basic algorithmic question concerning squares is to check if a given string of length  $n$  is square-free, that is, does not contain a fragment of such form. Main and Lorentz [J. Algorithms 1984] designed an  $\mathcal{O}(n \log n)$  time algorithm for this problem, and proved a matching lower bound assuming the so-called general alphabet, meaning that the algorithm is only allowed to check if two characters are equal. However, their lower bound also assumes that there are  $\Omega(n)$  distinct symbols in the string. As an open question, they asked if there is a faster algorithm if one restricts the size of the alphabet. Crochemore [Theor. Comput. Sci. 1986] designed a linear-time algorithm for constant-size alphabets, and combined with more recent results his approach in fact implies such an algorithm for linearly-sortable alphabets. Very recently, Ellert and Fischer [ICALP 2021] significantly relaxed this assumption by designing a linear-time algorithm for general ordered alphabets, that is, assuming a linear order on the characters that permits constant time order comparisons. However, the open question of Main and Lorentz from 1984 remained unresolved for general (unordered) alphabets. In this paper, we show that testing square-freeness of a length- $n$  string over general alphabet of size  $\sigma$  can be done with  $\mathcal{O}(n \log \sigma)$  comparisons, and cannot be done with  $o(n \log \sigma)$  comparisons. We complement this result with an  $\mathcal{O}(n \log \sigma)$  time algorithm in the Word RAM model. Finally, we extend the algorithm to reporting all the runs (maximal repetitions) in the same complexity.

## 1 Introduction

The notion of repetition is a central concept in combinatorics on words and algorithms on strings. In this context, a word or a string is simply a sequence of characters from some finite alphabet  $\Sigma$ . In the most basic version, a repetition consists of two (or more) consecutive occurrences of the same fragment. Repetitions are interesting not only from a purely theoretical point of view, but are also very relevant in bioinformatics [90]. A repetition could be a square, defined as two consecutive occurrences of the same fragment, a higher power (for example, a cube), or a run, which is a length-wise maximal periodic substring. For example, both **anan** and **nana** are squares with two occurrences each in **banananas**, and they belong to the same run **ananana**. In this paper, we start by focusing on squares, then generalize our results for runs.

The study of squares in strings goes back to the work of Thue published in 1906 [1], who considered the question of constructing an infinite word with no squares. It is easy to see that any sufficiently long binary word must contain a square, and Thue proved that there exists an

infinite ternary word with no squares. His result has been rediscovered multiple times, and in 1979 Bean, Ehrenfeucht and McNulty [19] started a systematic study of the so-called avoidable repetitions, see for example the survey by Currie [98].

**Combinatorics on words.** The basic tool in the area of combinatorics on words is the so-called periodicity lemma. A period of a string  $T[1..n]$  is an integer  $d$  such that  $T[i] = T[i + d]$  for every  $i \in [1, n - d]$ , and the periodicity lemma states that if  $p$  and  $q$  are both such periods and  $p + q \leq n + \gcd(p, q)$  then  $\gcd(p, q)$  is also a period [7]. This was generalised in a myriad of ways, for strings [64, 68, 92], partial words (words with don't cares) [62, 120, 79, 97, 76, 180, 357], Abelian periods [103, 168], parametrized periods [118], order-preserving periods [224, 316], approximate periods [138, 154, 193]. Now, a square can be defined as a fragment of length twice its period. The string  $\mathbf{a}^n$  contains  $\Omega(n^2)$  such fragments, thus from the combinatorial point of view it is natural to count only distinct squares. Fraenkel and Simpson [59] showed an upper bound of  $2n$  and a lower bound of  $n - \Theta(\sqrt{n})$  for the maximum number of distinct squares in a length- $n$  string. After a sequence of improvements [113, 201, 326], the upper bound was very recently improved to  $n$  [350]. The last result was already generalised to higher powers [358]. Another way to avoid the trivial examples such as  $\mathbf{a}^n$  is to count only maximal periodic fragments, that is, fragments with period at most half of their length and that cannot be extended to the left or to the right without breaking the period. Such fragments are usually called runs. Kolpakov and Kucherov [65] showed an upper bound of  $\mathcal{O}(n)$  on their number, and this started a long line of work on determining the exact constant [109, 126, 121, 123, 131, 150], culminating in the paper of Bannai et al. [236] showing an upper bound of  $n$ , and followed by even better upper bounds for binary strings [203, 245]. This was complemented by a sequence of lower bounds [122, 125, 136, 146].

**Algorithms on strings.** In this paper, we are interested in the algorithmic aspects of detecting repetitions in strings. The most basic question in this direction is checking if a given length- $n$  string contains at least one square, while the most general version asks for computing all the runs. Testing square-freeness was first considered by Main and Lorentz [27], who designed an  $\mathcal{O}(n \log n)$  time algorithm based on a divide-and-conquer approach and a linear-time procedure for finding all new squares obtained when concatenating two strings. In fact, their algorithm can be used to find (a compact representation of) all squares in a given string within the same time complexity. They also proved that any algorithm based on comparisons of characters needs  $\Omega(n \log n)$  such operations to test square-freeness in the worst case. Here, comparisons of characters means checking if characters at two positions of the input string are equal. However, to obtain the lower bound they had to consider instances consisting of even up to  $n$  distinct characters, that is, over alphabet of size  $n$ . This is somewhat unsatisfactory, and motivates the following open question that was explicitly asked by Main and Lorentz [27]:

**Question 1.1.** *Is there a faster algorithm to determine if a string is square-free if we restrict the size of the alphabet?*

Crochemore [22] gave another  $\mathcal{O}(n \log n)$  time algorithm for finding all repetitions, and also showed that for constant-size alphabets testing square-freeness can be done in  $\mathcal{O}(n)$  time [30]. In fact, the latter algorithm works in  $\mathcal{O}(n \log \sigma)$  time for alphabets of size  $\sigma$  with a linear order on the characters. That is, it needs to test if the character at some position is smaller than the character at another position. In the remaining part of the paper, we will refer to this model as general ordered alphabet, while the model in which we can only test equality of characters will be called general (unordered) alphabet. Later, Kosaraju [49] showed that in fact, assuming constant-size alphabet,  $\mathcal{O}(n)$  time is enough to find the shortest square starting at each position of the input string. Apostolico and Preparata [24] provide another  $\mathcal{O}(n \log n)$  time algorithm

assuming a general ordered alphabet, based more on data structure considerations than combinatorial properties of words. Finally, a number of alternative  $\mathcal{O}(n \log n)$  and  $\mathcal{O}(n \log \sigma)$  time algorithms (respectively, for general unordered and general ordered alphabets) can be obtained from the work on online [124, 183, 208] and parallel [51] square detection (interestingly, this cannot be done efficiently in the related streaming model [300, 359]).

Faster algorithms for testing square-freeness of strings over general ordered alphabets were obtained from more general results on finding all runs. Kolpakov and Kucherov [65] not only proved that any length- $n$  string contains only  $\mathcal{O}(n)$  runs, but also showed how to find them in the same time assuming linearly-sortable alphabet. Every square is contained in a run, and every run contains at least one square, thus this in particular implies a linear-time algorithm for testing square-freeness over such alphabets. For general ordered alphabets, Kosolobov [207] showed that the decision tree complexity of this problem is only  $\mathcal{O}(n)$ , and later complemented this with an efficient  $\mathcal{O}(n(\log n)^{2/3})$  time algorithm [221] (still using only  $\mathcal{O}(n)$  comparisons). The time complexity was then improved to  $\mathcal{O}(n \log \log n)$  by providing a general mechanism for answering longest common extension (LCE) queries for general ordered alphabets [217], and next to  $\mathcal{O}(n\alpha(n))$  by observing that the LCE queries have additional structure [214]. Finally, Ellert and Fischer provided an elegant  $\mathcal{O}(n)$  time algorithm, thus fully resolving the complexity of square detection for general ordered alphabets. However, for general (unordered) alphabets the question of Main and Lorentz remains unresolved, with the best upper bound being  $\mathcal{O}(n \log n)$ , and only known to be asymptotically tight for alphabets of size  $\Theta(n)$ .

**General alphabets.** While in many applications one can without losing generality assume some ordering on the characters of the alphabet, no such ordering is necessary for defining what a square is. Thus, it is natural from the mathematical point of view to seek algorithms that do not require such an ordering to efficiently test square-freeness. Similar considerations have lead to multiple beautiful results concerning the pattern matching problem, such as constant-space algorithms [25, 42], or the works on the exact number of required equality comparisons [55]. More recent examples include the work of Duval, Lecroq, and Lefebvre [177] on computing the unbordered conjugate/rotation, and Kosolobov [222] on finding the leftmost critical point.

### Main results.

We consider the complexity of checking if a given string  $T[1..n]$  containing  $\sigma$  distinct characters is square-free. The input string can be only accessed by issuing comparisons  $T[i] \stackrel{?}{=} T[j]$ , and the value of  $\sigma$  is not assumed to be known. We start by analysing the decision tree complexity of the problem. That is, we only consider the required and necessary number of comparisons, without worrying about an efficient implementation. We show that, even if the value of  $\sigma$  is assumed to be known,  $\Omega(n \log \sigma)$  comparisons are required.

**Theorem 4.1.** *For any integers  $n$  and  $\sigma$  with  $8 \leq \sigma \leq n$ , there is no deterministic algorithm that performs at most  $n \ln \sigma - 3.6n = \mathcal{O}(n \ln \sigma)$  comparisons in the worst case, and determines whether a length- $n$  string with at most  $\sigma$  distinct symbols from a general unordered alphabet is square-free.*

Next, we show that  $\mathcal{O}(n \log \sigma)$  comparisons are sufficient. We stress that the value of  $\sigma$  is not assumed to be known. In fact, as a warm-up for the above theorem, we first prove that finding a sublinear multiplicative approximation of this value requires  $\Omega(n\sigma)$  comparisons. This does not contradict the claimed upper bound, as we are only saying that the number of comparisons used on a particular input string is at most  $\mathcal{O}(n \log \sigma)$ , but might actually be smaller. Thus, it is not possible to extract any meaningful approximation of the value of  $\sigma$  from the number of used comparisons.

**Theorem 4.2.** *Testing square-freeness of a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be done with  $\mathcal{O}(n \log \sigma)$  comparisons.*

The proof of the above result is not efficient in the sense that it only restricts the overall number of comparisons, and not the time to actually figure out which comparisons should be used. A direct implementation results in a quadratic time algorithm. We first show how to improve this to  $\mathcal{O}(n \log \sigma + n \log^* n)$  time (while still keeping the asymptotically optimal  $\mathcal{O}(n \log \sigma)$  number of comparisons), and finally to  $\mathcal{O}(n \log \sigma)$ . In this part of the paper, we assume the Word RAM model with word of length  $\Omega(\log n)$ . We stress that the input string is still assumed to consist of characters that can be only tested for equality, that is, one should think that we are given oracle access to a functions that, given  $i$  and  $j$ , checks whether  $T[i] = T[j]$ .

**Theorem 4.3.** *Testing square-freeness of a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be implemented in  $\mathcal{O}(n \log \sigma)$  comparisons and time.*

Finally, we also generalize this result to the computation of runs.

**Theorem 4.4.** *Computing all runs in a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be implemented in  $\mathcal{O}(n \log \sigma)$  comparisons and time.*

Altogether, our results fully resolve the open question of Main and Lorentz for the case of general unordered alphabets and deterministic algorithms. We leave extending our lowerbound to randomised algorithms as an open question.

#### Overview of the methods.

As mentioned before, Main and Lorentz [27] designed an  $\mathcal{O}(n \log n)$  time algorithm for testing square-freeness of length- $n$  strings over general alphabets. The high-level idea of their algorithm goes as follows. They first designed a procedure for checking, given two strings  $x$  and  $y$ , if their concatenation contains a square that is not fully contained in  $x$  nor  $y$  in  $\mathcal{O}(|x| + |y|)$  time. Then, a divide-and-conquer approach can be used to detect a square in the whole input string in  $\mathcal{O}(n \log n)$  total time. For general alphabets of unbounded size this cannot be improved, but Crochemore [30] showed that, for general ordered alphabets of size  $\sigma$ , a faster  $\mathcal{O}(n \log \sigma)$  time algorithm exists. The gist of his approach is to first obtain the so-called  $f$ -factorisation of the input string (related to the well-known Lempel-Ziv factorisation), that in a certain sense “discovers” repetitive fragments. Then, this factorisation can be used to apply the procedure of Main and Lorentz on appropriately selected fragments of the input strings in such a way that the leftmost occurrence of every distinct square is detected, and the total length of the strings on which we apply the procedure is only  $\mathcal{O}(n)$ . The factorisation can be found in  $\mathcal{O}(n \log \sigma)$  time for general ordered alphabets of size  $\sigma$  by, roughly speaking, constructing some kind of suffix structure (suffix array, suffix tree or suffix automaton).

For general (unordered) alphabets, computing the  $f$ -factorisation (or anything similar) seems problematic, and in fact we show (as a corollary of our lower bound on approximating the alphabet size) that computing the  $f$ -factorisation or Lempel-Ziv-factorisation (LZ-factorisation) of a given length- $n$  string containing  $\sigma$  distinct characters requires  $\Omega(n\sigma)$  equality tests. Thus, we need another approach. Additionally, the  $\mathcal{O}(n)$  time algorithm of Ellert and Fischer [334] hinges on the notion of Lyndon words, which is simply not defined for strings over general alphabets. Thus, at first glance it might seem that  $\Theta(n\sigma)$  is the right time complexity for testing square-freeness over length- $n$  strings over general alphabets of size  $\sigma$ . However, due to the  $\Omega(n \log n)$  lower bound of Main and Lorentz for testing square-freeness of length- $n$  string consisting of up to  $n$  distinct characters, one might hope for an  $\mathcal{O}(n \log \sigma)$  time algorithm when there are only  $\sigma$  distinct characters.

We begin our paper with a lower bound of  $\Theta(n \log \sigma)$  for such strings. Intuitively, we show that testing square-freeness has the direct sum property:  $\frac{n}{\sigma}$  instances over length- $\sigma$  strings can be combined into a single instance over length- $n$  string. As in the proof of Main and Lorentz, we use the adversarial method. While the underlying calculation is essentially the same, we

need to appropriately combine the smaller instances, which is done using the infinite square-free Prouhet-Thue-Morse sequence, and use significantly more complex rules for resolving the subsequent equality tests. As a warm-up for the adversarial method, we prove that computing any meaningful approximation of the number of distinct characters requires  $\Omega(n\sigma)$  such tests, and that this implies the same lower bound on computing the  $f$ -factorisation and the Lempel-Ziv factorisation (if the size of the alphabet is unknown in advance).

We then move to designing an approach that uses  $\mathcal{O}(n \log \sigma)$  equality comparisons to test square-freeness. As discussed earlier, one way of detecting squares uses the  $f$ -factorisation of the string, which is similar to its LZ factorisation. However, as we prove in Corollary 4.10 and 4.11, we cannot compute either of these factorisations over a general unordered alphabet in  $\mathcal{O}(n\sigma)$  comparisons. Therefore, we will instead use a novel type of factorisation,  $\Delta$ -approximate LZ factorisation, that can be seen as an approximate version of the LZ factorisation. Intuitively, its goal is to “capture” all sufficiently long squares, while the original LZ factorisation (or  $f$ -factorisation) captures all squares. Each phrase in a  $\Delta$ -approximate LZ factorisation consists of a head of length at most  $\Delta$  and a tail (possibly empty) that must occur at least once before, such that the whole phrase is at least as long as the classical LZ phrase starting at the same position. Contrary to the classical LZ factorisation, this factorisation is not unique. The advantage of our modification is that there are fewer phrases (and there is more flexibility as to what they should be), and hence one can hope to compute such factorisation more efficiently.

To design an efficient construction method for  $\Delta$ -approximate LZ factorisation, we first show how to compute a sparse suffix tree while trying to use only a few symbol comparisons. This is then applied on a set of positions from a so-called difference cover with some convenient synchronizing properties. Then, a  $\Delta$ -approximate LZ factorisation allows us to detect squares of length  $\geq 8\Delta$ .

The first warm-up algorithm fixes  $\Delta$  depending on  $n$  and  $\sigma$  (assuming that  $\sigma$  is known), and uses the approximate LZ factorisation to find all squares of length at least  $8\Delta$ . It then finds all the shorter squares by dividing the string in blocks of length  $8\Delta$ , and applying the original algorithm by Main and Lorentz on each block pair. Our choice of  $\Delta$  leads to  $\mathcal{O}(n(\lg \sigma + \lg \lg n))$  comparisons.

The improved algorithm does not need to know  $\sigma$ , and instead starts with a large  $\Delta = \Omega(n)$ , and then progressively decreases  $\Delta$  in at most  $\mathcal{O}(\lg \lg n)$  phases, where later phases detect shorter squares. As soon as we notice that there are many distinct characters in the alphabet, by carefully adjusting the parameters we can afford switching to the approach of Main and Lorentz on sufficiently short fragments of the input string. Since we cannot afford  $\Omega(n)$  comparisons per phase, we use a deactivation technique, where whenever we perform a large number of comparisons in a phase, we will discard a large part of the string in all following phases. More precisely, during a given phase, we avoid looking for squares in a fragment fully contained in a tail from an earlier phase. This leads to optimal  $\mathcal{O}(n \lg \sigma)$  comparisons.

The above approach uses an asymptotically optimal number of equality tests in the worst case, but does not result in an efficient algorithm. The main bottleneck is constructing the sparse suffix trees. However, it is not hard to provide an efficient implementation using the general mechanism for answering LCE queries for strings over general alphabets [217]. Unfortunately, the best known approach for answering such queries incurs an additional  $\mathcal{O}(n \log^* n)$  in the time complexity, even if the size of the alphabet is constant. We overcome this technical hurdle by carefully deactivating fragments of the text to account for the performed work.

Many of our techniques can easily be modified to compute all runs rather than detecting squares. We exploit that the approximate factorisation reveals long substrings with an earlier occurrence. Hence we compute runs only for the first occurrence of such substrings, while for later occurrences we simply copy the already computed runs. By carefully arranging the order



of the computation, we ensure that the total time for copying is bounded by the number of runs, which is known to be  $\mathcal{O}(n)$ . This way, we achieve  $\mathcal{O}(n \lg \sigma)$  time and comparisons to compute all runs.

## 2 Preliminaries

**Strings.** A string of length  $n$  is a sequence  $T[1] \dots T[n]$  of characters from a finite alphabet  $\Sigma$  of size  $\sigma$ . The substring  $T[i..j]$  is the string  $T[i] \dots T[j]$ , whereas the fragment  $T[i..j]$  refers to the specific occurrence of  $T[i..j]$  starting at position  $i$  in  $T$ . If  $i > j$ , then  $T[i..j]$  is the empty string. A suffix of  $T$  has the form  $T[i..n]$ . We say that a fragment  $T[i'..j']$  is properly contained in another fragment  $T[i..j]$  if  $i < i' \leq j' < j$ . A substring is properly contained in  $T[i..j]$ , if it equals a fragment that is properly contained in  $T[i..j]$ . We write  $T[i..j)$  as a shortcut for  $T[i..j - 1]$ . Similarly, we write  $[i, j) = [i, j + 1)$  as a shortcut for the integer interval  $\{i, \dots, j\}$ . Given two positions  $i \leq j$ , their longest common extension (LCE) is the length of the longest common prefix between suffixes  $T[i..n]$  and  $T[j..n]$ , formally defined as  $\text{LCE}(i, j) = \text{LCE}(j, i) = \max\{\ell \in \{0, \dots, n - j + 1\} \mid T[i..i + \ell) = T[j..j + \ell)\}$ .

**Definition 4.5.** A positive integer  $p$  is a period of a string  $T[1..n]$  if  $T[i] = T[i + p]$  for every  $i \in \{1, \dots, n - p\}$ . The smallest such  $p$  is called the period of  $T[1..n]$ , and we call a string periodic if its period  $p$  is at most  $\frac{n}{2}$ .

**Computational model.** For a general unordered alphabet  $\Sigma$ , the only allowed operation on the characters is comparing for equality. In particular, there is no linear order on the alphabet. Unless explicitly stated otherwise, we will only use such comparisons. A general ordered alphabet has a total order, such that comparisons of the type less-equals are possible.

In the algorithmic part of the paper, we assume the standard unit-cost Word RAM model with words of length  $\Omega(\log n)$ , but the algorithm is only allowed to access the input string  $T[1..n]$  by comparisons  $T[i] \stackrel{?}{=} T[j]$ , which are assumed to take constant time. We say that a string of length  $n$  is over a linearly-sortable alphabet, if we can sort the  $n$  symbols of the string in  $\mathcal{O}(n)$  time. Note that whether or not an alphabet is linearly-sortable depends not only on the alphabet, but also on the string. For example, the alphabet  $\Sigma = \{1, \dots, m^{\mathcal{O}(1)}\}$  is linearly-sortable for strings of length  $n = \Omega(m)$  (e.g., using radix sort), but it is unknown whether it is linearly-sortable for all strings of length  $n = o(m)$  [83]. Our algorithm will internally use strings over linearly-sortable alphabets. We stress that in such strings the characters are not the characters from the input string, but simply integers calculated by the algorithm. Note that every linearly-sortable alphabet is also a general ordered alphabet.

**Squares and runs.** A square is a length- $2\ell$  fragment of period  $\ell$ . The following theorem is a classical result by Main and Lorentz [27].

**Theorem 4.6.** Testing square-freeness of  $T[1..n]$  over a general alphabet can be implemented in  $\mathcal{O}(n \log n)$  time and comparisons.

The proof of the above theorem is based on running a divide-and-conquer procedure using the following lemma.

**Lemma 4.7.** Given two strings  $x$  and  $y$  over a general alphabet, we can test if there is a square in  $xy$  that is not fully contained in  $x$  nor  $y$  in  $\mathcal{O}(|x| + |y|)$  time and comparisons.

A repetition is a length- $\ell$  fragment of period at most  $\frac{\ell}{2}$ . A run is a maximal repetition. Formally, a repetition in  $T[1..n]$  is a triple  $\langle s, e, p \rangle$  with  $s, e \in [1, n]$  and  $p \in [1, \frac{e-s+1}{2}]$  such that

$p$  is the smallest period of  $T[s..e]$ . A run is a repetition  $\langle s, e, p \rangle$  that cannot be extended to the left nor to the right with the same period, in other words  $s = 1$  or  $T[s - 1] \neq T[s - 1 + p]$  and  $e = n$  or  $T[e + 1] \neq T[e + 1 - p]$ . The celebrated runs conjecture, proven by Bannai et al. [236], states that the number of runs in any length- $n$  string is less than  $n$ . Ellert and Fischer [334] showed that all runs in a string over a general ordered alphabet can be computed in  $\mathcal{O}(n)$  time. As mentioned earlier, each run contains a square, and each square is contained in a run. Thus, the string contains a square if and only if it contains a run, and it follows:

**Theorem 4.8.** *Computing all runs (and thus testing square-freeness) of  $T[1..n]$  over a general ordered alphabet can be implemented in  $\mathcal{O}(n)$  time.*

**Lempel-Ziv factorisation.** The unique LZ phrase starting at position  $s$  of  $T[1..n]$  is a fragment  $T[s..e]$  such that  $T[s..(e - 1)]$  occurs at least twice in  $T[1..(e - 1)]$  and either  $e = n$  or  $T[s..e]$  occurs only once in  $T[1..e]$ . The Lempel-Ziv factorisation of  $T$  consists of  $z$  phrases  $f_1, \dots, f_z$  such that the concatenation  $f_1 \dots f_z$  is equal to  $T[1..n]$  and each  $f_i$  is the unique LZ phrase starting at position  $1 + \sum_{j=1}^{i-1} |f_j|$ .

**Tries.** Given a collection  $\mathcal{S} = \{T_1, \dots, T_k\}$  of strings over some alphabet  $\Sigma$ , its trie is a rooted tree with edge labels from  $\Sigma$ . For any node  $v$ , the concatenation of the edge labels from the root to the node  $v$  spells a string. The string-depth of a node is the length of the string that it spells. No two nodes spell the same string, i.e., for any node, the labels of the edges to its children are pairwise distinct. Each leaf spells one of the  $T_i$ , and each  $T_i$  is spelled by either an internal node or a leaf.

The compacted trie of  $\mathcal{S}$  can be obtained from its (non-compacted) trie by contracting each path between a leaf or a branching node and its closest branching ancestor into a single edge (i.e., by contraction we eliminate all non-branching internal nodes). The label of the new edge is the concatenation of the edge labels of the contracted path in root to leaf direction. Since there are at most  $k$  leaves and all internal nodes are branching, there are  $\mathcal{O}(k)$  nodes in the compacted trie. Each edge label is some substring  $T_i[s..e]$  of the string collection, and we can avoid explicitly storing the label by instead storing the reference  $(i, s, e)$ . Thus  $\mathcal{O}(k)$  words are sufficient for storing the compacted trie. Consider a string  $T'$  that is spelled by a node of the non-compacted trie. We say that  $T'$  is explicit, if and only if it is spelled by a node of the compacted trie. Otherwise  $T'$  is implicit.

The suffix tree of a string  $T[1..n]$  is the compacted trie containing exactly its suffixes, i.e., a trie over the string collection  $\{T[i..n] \mid i \in \{1, \dots, n\}\}$ . It is one of the most fundamental data structures in string algorithmics, and is widely used, e.g., for compression and indexing [57]. The suffix tree can be stored in  $\mathcal{O}(n)$  words of memory, and for linearly-sortable alphabets it can be computed in  $\mathcal{O}(n)$  time [56]. The sparse suffix tree of  $T$  for some set  $B \subseteq \{1, \dots, n\}$  of sample positions is the compacted trie containing exactly the suffixes  $\{T[i..n] \mid i \in B\}$ . It can be stored in  $\mathcal{O}(|B|)$  words of memory.

We assume that  $T$  is terminated by some special symbol  $T[n] = \$$  that occurs nowhere else in  $T$ . This ensures that each suffix is spelled by a leaf, and we label the leaves with the respective starting positions of the suffixes. Note that for any two leaves  $i \neq j$ , their lowest common ancestor (i.e., the deepest node that is an ancestor of both  $i$  and  $j$ ) spells a string of length  $\text{LCE}(i, j)$ .

### 3 Lower Bounds

In this section, we show lower bounds on the number of symbol comparisons required to compute a meaningful approximation of the alphabet size (Section 3.1) and to test square-freeness (Section 3.2). For both bounds we use an adversarial method, which we briefly outline now.

The present model of computation may be interpreted as follows. An algorithm working on a string over a general unordered alphabet has no access to the actual string. Instead, it can only ask an oracle whether or not there are identical symbols at two positions. The number of questions asked is exactly the number of performed comparisons. In order to show a lower bound on the number of comparisons required to solve some problem, we describe an adversary that takes over the role of the oracle, forcing the algorithm to perform as many symbol comparisons as possible.

We use a conflict graph  $G = (V, E)$  with  $V = \{1, \dots, n\}$  and  $E \subseteq V^2$  to keep track of the answers given by the adversary. The nodes directly correspond to the positions of the string. Initially, we have  $E = \emptyset$  and all nodes are colorless, which formally means that they have color  $\gamma(i) = \perp$ . During the algorithm execution, the adversary may assign colors from the set  $\Sigma = \{0, \dots, n-1\}$  to the nodes, which can be seen as permanently fixing the alphabet symbol at the corresponding position (i.e., each node gets colored at most once). The rule used for coloring nodes depends on the lower bound that we want to show (we describe this in detail in the respective sections). Apart from this coloring rule, the general behaviour of the adversary is as follows. Whenever the algorithm asks whether  $T[i] = T[j]$  holds, the adversary answers “yes” if and only if  $\gamma(i) = \gamma(j) \neq \perp$ . Otherwise, it answers “no” and inserts an edge  $(i, j)$  into  $E$ . Whenever the adversary assigns the color of a node, it has to choose a color that is not used by any of the adjacent nodes in the conflict graph. This ensures that the coloring does not contradict the answers given in the past.

Let us define a set  $\mathcal{T} \subseteq \Sigma^n$  of strings that is consistent with the answers given by the adversary. A string  $T \in \Sigma^n$  is a member of  $\mathcal{T}$  if

$$\forall i \in V : \gamma(i) \in \{\perp, T[i]\} \quad \wedge \quad \forall i, j \in V : (T[i] = T[j]) \implies (i, j) \notin E.$$

Note that  $\mathcal{T}$  changes over time. Initially (before the algorithm starts), we have  $\mathcal{T} = \Sigma^n$ . With every question asked, the algorithm might eliminate some strings from  $\mathcal{T}$ . However, there is always at least one string in  $\mathcal{T}$ , which can be obtained by coloring each colorless node in a previously entirely unused color.

### 3.1 Approximating the Alphabet Size

Given a string  $T[1..n]$  of unknown alphabet size  $\sigma \geq 2$ , assume that we want to compute an approximation of  $\sigma$ . We show that if an algorithm takes at most  $\frac{n\sigma}{8}$  comparisons in the worst-case, then it cannot distinguish strings with at most  $\sigma$  distinct symbols from strings with at least  $\frac{n}{2}$  distinct symbols. Thus, any meaningful approximation of  $\sigma$  requires  $\Omega(n\sigma)$  comparisons.

For the sake of the proof, consider an algorithm that performs at most  $\frac{n\sigma}{8}$  comparisons when given a length- $n$  string with at most  $\sigma \geq 2$  distinct symbols. We use an adversary as described at the beginning of Section 3, and ensure that the set  $\mathcal{T}$  of strings consistent with the adversary’s answers always contains a string with at most  $\sigma$  distinct symbols. Thus, the algorithm terminates after at most  $\frac{n\sigma}{8}$  comparisons. At the same time, we ensure that  $\mathcal{T}$  also contains a string with at least  $\frac{n}{2}$  distinct symbols, which yields the desired result. The adversary is equipped with the following coloring rule. All colors are from  $\{1, \dots, \sigma\}$ . Whenever the degree of a node in the conflict graph becomes  $\sigma - 1$ , we assign its color. We avoid the colors of the  $\sigma - 1$  adjacent nodes in the conflict graph. At any moment in time, we could hypothetically complete the coloring by assigning one of the colors  $\{1, \dots, \sigma\}$  to each colorless node, avoiding the colors of adjacent nodes. This way, each node gets assigned one of the  $\sigma$  colors, which means that  $\mathcal{T}$  contains a string with at most  $\sigma$  distinct symbols. It follows that the algorithm terminates after at most  $\frac{n\sigma}{8}$  comparisons. Each comparison may increase the degree of two nodes by one. Thus, after  $\frac{n\sigma}{8}$  comparisons, there are at most  $\frac{n\sigma}{8} \cdot \frac{2}{\sigma-1} \leq \frac{n}{2}$  nodes with degree at least  $\sigma - 1$ . Therefore,

at least  $\frac{n}{2}$  nodes are colorless. We could hypothetically color them in  $\frac{n}{2}$  distinct colors, which means that  $\mathcal{T}$  contains a string with at least  $\frac{n}{2}$  distinct symbols. This leads to the following result.

**Theorem 4.9.** *For any integers  $n$  and  $\sigma$  with  $2 \leq \sigma < \frac{n}{2}$ , there is no deterministic algorithm that performs at most  $\frac{n\sigma}{8}$  equality-comparisons in the worst case, and is able to distinguish length- $n$  strings with at most  $\sigma$  distinct symbols from length- $n$  strings with at least  $\frac{n}{2}$  distinct symbols.*

The theorem implies lower bounds on the number of comparisons needed to compute the LZ factorisation (as defined in Section 2) and the  $f$ -factorisation. In the unique  $f$ -factorisation  $T = f_1 f_2 \dots f_z$ , each factor  $f_i$  is either a single symbol that does not occur in  $f_1 \dots f_{i-1}$ , or it is the fragment of maximal length such that  $f_i$  occurs twice in  $f_1 \dots f_i$ .

**Corollary 4.10.** *For any integers  $n$  and  $\sigma$  with  $2 \leq \sigma < \frac{n}{4}$ , there is no deterministic algorithm that performs at most  $\frac{(n-1)\sigma}{16}$  equality-comparisons in the worst case, and computes the  $f$ -factorisation of a length- $n$  string with at most  $\sigma$  distinct symbols.*

*Proof.* For some string  $T = T[1]T[2] \dots T[\frac{n}{2}]$  with  $\sigma$  distinct symbols, consider the length- $n$  string  $T' = T[1]T[1]T[2]T[2] \dots T[\frac{n}{2}]T[\frac{n}{2}]$  with  $\sigma$  distinct symbols constructed by doubling each character of  $T$ . The alphabet size of  $T$  is exactly the number of length-one phrases in the  $f$ -factorisation of  $T'$  starting at odd positions in  $T'$ . Thus, by Theorem 4.9, we need  $\frac{n\sigma}{16} = \frac{|T|\sigma}{8}$  comparisons to find the  $f$ -factorisation of  $T'$ . We assumed that  $n$  is even, and account for odd  $n$  by adjusting the bound to  $\frac{(n-1)\sigma}{16}$ .  $\square$

**Corollary 4.11.** *For any integers  $n$  and  $\sigma$  with  $3 \leq \sigma < \frac{n}{6} + 1$ , there is no deterministic algorithm that performs at most  $\frac{(n-2)(\sigma-1)}{24}$  equality-comparisons in the worst case, and computes the Lempel-Ziv factorisation of a length- $n$  string with at most  $\sigma$  distinct symbols.*

*Proof.* For some string  $T = T[1]T[2] \dots T[\frac{n}{3}]$  with  $\sigma - 1$  distinct symbols, let  $T'$  be the length- $n$  string with  $\sigma$  distinct symbols constructed by doubling every character of  $T$  with a separator in between, i.e.,  $T' = T[1]T[1]\#T[2]T[2]\# \dots \#T[\frac{n}{3}]T[\frac{n}{3}]\#$ . The first occurrence of character  $x$  in  $T$  corresponds to the first occurrence of  $xx\#$  in  $T'$ , thus the preceding phrase (possibly of length one) ends at the first  $x$  in the first occurrence of  $xx\#$ , and the subsequent phrase must be  $x\#$ . Then, for the later occurrences of  $xx\#$  we cannot have that  $x\#$  is a phrase. Consequently, the alphabet size of  $T$  is exactly the number of length-two phrases in the Lempel-Ziv factorisation of  $T'$  starting at positions  $i \equiv 2 \pmod{3}$  in  $T'$ . Thus, by Theorem 4.9, we need  $\frac{n(\sigma-1)}{24} = \frac{|T|(\sigma-1)}{8}$  comparisons to find the LZ factorisation of  $T'$ . We assumed that  $n$  is divisible by 3, and account for this by adjusting the bound to  $\frac{(n-2)(\sigma-1)}{24}$ .  $\square$

## 3.2 Testing Square-Freeness

In this section, we prove that testing square-freeness requires at least  $n \ln \sigma - 3.6n$  comparisons (even if  $\sigma$  is known). The proof combines the idea behind the original  $\Omega(n \lg n)$  lower bound by Main and Lorentz [27] with the adversary described at the beginning of Section 3. This time, we ensure that  $\mathcal{T}$  always contains a square-free string with at most  $\sigma$  distinct symbols. At the same time, we try to ensure that  $\mathcal{T}$  also contains a string with at least one square. We will show that we can maintain this state until at least  $n \ln \sigma - 3n$  comparisons have been performed.

The string (or rather family of strings) constructed by the adversary is organized in  $\left\lceil \frac{4n}{\sigma} \right\rceil$  non-overlapping blocks of length  $\frac{\sigma}{4}$  (we assume  $\frac{\sigma}{4} \in \mathbb{N}$  and  $8 \leq \sigma \leq n$ ). Each block begins with a special separator symbol. More precisely, the first symbol of the  $k$ -th block is the  $k$ -th symbol

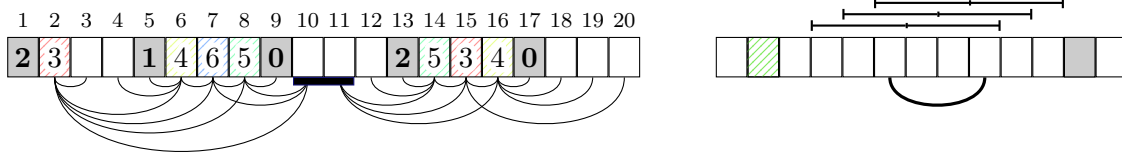


Figure 4.1: Example conflict graph of the adversary described in Section 3.2. The alphabet  $\{0, \dots, 15\}$  is of size  $\sigma = 16$ . The blocks are of length  $\frac{\sigma}{4} = 4$ . The gray nodes are exactly the starting positions of the blocks and contain the symbols of the ternary Thue-Morse sequence  $v = 2, 1, 0, 2, 0, 1, 2, \dots$ , which is square-free. We assume that the colored nodes were colored in the following order: 2, 6, 8, 7, 15, 16, 14. At the time of coloring node 8, we had to avoid colors 0, 1, 2 (because they are reserved for the separator positions), 3 (because the adjacent node 2 already has color 3), and 4 (because node 6 is in the same block and already has color 4). The algorithm has not eliminated all squares yet. For example, nodes 10 and 11 with absent edge  $(10, 11) \notin E$  are adjacent to nodes of colors  $\{3, 6, 5\} \cup \{5, 3, 4\}$ . Thus, any of the colors  $\{0, 1, 2\} \cup \{7, \dots, 15\}$  can be assigned to both nodes, enforcing the square  $T[10..11]$ . As visualized on the right, an edge of length  $\ell$  eliminates at most  $\ell$  squares.

of a ternary square-free word over the alphabet  $\{0, 1, 2\}$  (e.g., the distance between the  $k^{\text{th}}$  and  $(k+1)^{\text{th}}$  occurrence of 0 in the Prouhet-Thue-Morse sequence, also known as the ternary Thue-Morse-Sequence, see [58, Corollary 1]). Initially, the adversary colors the nodes that correspond to the separator positions in their respective colors from  $\{0, 1, 2\}$ . All remaining nodes will later get colors other than  $\{0, 1, 2\}$ . Any fragment crossing a block boundary can be projected on the colors  $\{0, 1, 2\}$ , and by construction the string cannot contain a square. Thus, the separator symbols ensure that there is no square crossed by a block boundary, which implies that the string is square-free if and only if each of its blocks is square-free.

During the algorithm execution, we use the following coloring rule. The available colors are  $\{3, \dots, \sigma - 1\}$ . Whenever the degree of a node becomes  $\frac{\sigma}{4}$ , we assign its color. We avoid not only the at most  $\frac{\sigma}{4}$  colors of already colored neighbors in the conflict graph, but also the less than  $\frac{\sigma}{4}$  colors of nodes within the same block (due to  $\sigma \geq 8$ , there are at least  $\sigma - 3 - \frac{\sigma}{2} \geq 1$  colors available). An example of the conflict graph is provided in Fig. 4.1. At any moment in time, we could hypothetically complete the coloring by assigning one of the colors  $\{3, \dots, \sigma - 1\}$  to each colorless node, avoiding colors of adjacent nodes and colors of nodes in the same block. Afterwards, each node holds one of the  $\sigma$  colors, but no two nodes within the same block have the same color. Thus, each block is square-free, and therefore  $\mathcal{T}$  always contains a square-free string with at most  $\sigma$  distinct symbols.

Now we consider the state of the conflict graph *after the algorithm has terminated*. We are particularly concerned with consecutive ranges of colorless nodes. The following lemma states that for each such range, the algorithm either performed many comparisons, or we can enforce a square within the range.

**Lemma 4.12.** *Let  $R = \{i, \dots, j\} \subset V$  be a consecutive range of  $m = j - i + 1$  colorless nodes in the conflict graph. Then either  $|E \cap R^2| \geq \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m-2\ell+1}{\ell}$ , or there is a string  $T \in \mathcal{T}$  with at most  $\sigma$  distinct symbols such that  $T[i..j]$  contains a square.*

*Proof.* We say that an integer interval  $[x, x + 2\ell - 1]$  with  $i \leq x < (x + 2\ell - 1) \leq j$  has been *eliminated*, if for some  $y$  with  $x \leq y < x + \ell$  there is an edge  $(y, y + \ell)$  in the conflict graph. If such an edge exists, then (by the definition of  $\mathcal{T}$ ) all strings  $T \in \mathcal{T}$  satisfy  $T[y] \neq T[y + \ell]$ . Thus

$T[x..x + 2\ell - 1]$  is not a square for any of them.

Now we show that if  $[x, x + 2\ell - 1]$  has not been eliminated, then there exists a string  $T \in \mathcal{T}$  such that  $T[x..x + 2\ell - 1]$  is a square. For this purpose, consider any position  $y$  with  $x \leq y < x + \ell$ , i.e., a position in the first half of the potential square. Since  $[x, x + 2\ell - 1]$  has not been eliminated,  $(y, y + \ell)$  is not an edge in the conflict graph. It follows that we could assign the same color to  $y$  and  $y + \ell$ . We only have to avoid the at most  $2 \cdot (\frac{\sigma}{4} - 1)$  colors of adjacent nodes of both  $y$  and  $y + \ell$  in the conflict graph. Thus there are  $\frac{\sigma}{2} + 2$  appropriate colors that can be assigned to both nodes. Unlike during the algorithm execution, we do not need to avoid the special separator colors or the colors in the same block; since we are trying to enforce a square, we do not have to worry about accidentally creating one. By applying this coloring scheme for all possible choices of  $y$ , we enforce that all strings  $T \in \mathcal{T}$  have a square  $T[x..x + 2\ell - 1]$ . Note that by coloring additional nodes after the algorithm terminated, we only remove elements from  $\mathcal{T}$ . Thus, the strings with square  $T[x..x + 2\ell - 1]$  were already in  $\mathcal{T}$  when the algorithm terminated. It follows that, if the algorithm actually guarantees square-freeness, then it must have eliminated all possible intervals  $[x, x + 2\ell - 1]$  with  $i \leq x < (x + 2\ell - 1) \leq j$ .

While each interval needs at least one edge to be eliminated, a single edge eliminates multiple intervals. However, all the intervals eliminated by an edge must be of the same length. Now we give a lower bound on the number of edges needed to eliminate all intervals of length  $2\ell$ . Any edge  $(y, y + \ell)$  eliminates  $\ell$  intervals, namely the intervals  $[x, x + 2\ell - 1]$  that satisfy  $x \leq y < x + \ell$ . Within  $R$ , we have to eliminate  $m - 2\ell + 1$  intervals of length  $2\ell$ , namely the intervals  $[x, x + 2\ell - 1]$  that satisfy  $i \leq x \leq j - 2\ell + 1$  (see right side of Fig. 4.1). Thus we need at least  $\frac{m - 2\ell + 1}{\ell}$  edges to eliminate all squares of length  $2\ell$ . Finally, by summing over all possible values of  $\ell$ , we need at least  $\sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m - 2\ell + 1}{\ell}$  edges to eliminate all intervals in  $R$ . Note that the edges used for elimination have both endpoints in  $R$ , and are thus contained in  $E \cap R^2$ . Consequently, if  $|E \cap R^2| < \sum_{\ell=1}^{\lfloor m/2 \rfloor} \frac{m - 2\ell + 1}{\ell}$ , then not all intervals have been eliminated, and there is a string in  $\mathcal{T}$  that contains a square.  $\square$

Finally, we show that the algorithm either performed at least  $\Omega(n \lg \sigma)$  comparisons, or there is a string  $T \in \mathcal{T}$  that contains a square. Let  $c_1, c_2, \dots, c_k$  be exactly the colored nodes. Initially (before the algorithm execution), the adversary colored  $\lceil \frac{4n}{\sigma} \rceil$  nodes. Thus  $k \geq \lceil \frac{4n}{\sigma} \rceil$ , and there are  $k - \lceil \frac{4n}{\sigma} \rceil$  nodes that have been colored after their degree reached  $\frac{\sigma}{4}$ . Therefore, the sum of degrees of all colored nodes is at least  $(k - \lceil \frac{4n}{\sigma} \rceil) \cdot \frac{\sigma}{4} \geq \frac{\sigma k - 4n - \sigma}{4} \geq \frac{\sigma k - 5n}{4}$ . Each comparison may increase the degree of two nodes by one. Thus, the colored nodes account for at least  $\frac{\sigma k - 5n}{8}$  comparisons. There are  $k$  non-overlapping maximal colorless ranges of nodes, namely  $\{c_i + 1, \dots, c_{i+1} - 1\}$  for  $1 \leq i \leq k$  with auxiliary value  $c_{k+1} = n + 1$ . According to Lemma 4.12, each respective range accounts for  $e_i = \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell}$  edges, where  $m_i = c_{i+1} - c_i - 1$ . (No edge gets counted more than once because the ranges are non-overlapping, and both endpoints of the respective edges are within the range.) Thus, in order to verify square-freeness, the algorithm must have performed at least  $\sum_{i=1}^k e_i + \frac{\sigma k - 5n}{8}$  comparisons. The remainder of the proof consists of simple algebra. First, we provide a convenient lower bound for  $e_i$  (explained below):

$$\begin{aligned}
 e_i &= \sum_{\ell=1}^{\lfloor m_i/2 \rfloor} \frac{m_i - 2\ell + 1}{\ell} = \sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{m_i - 2\ell + 1}{\ell} \geq (m_i + 1) \left( \left( \sum_{\ell=1}^{\lceil m_i/2 \rceil} \frac{1}{\ell} \right) - 1 \right) \\
 &> (m_i + 1) \cdot \left( \ln \frac{m_i}{2} - \frac{1}{2} \right) \\
 &= (m_i + 1) \cdot \ln \frac{m_i}{2\sqrt{e}} \\
 &\geq (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}
 \end{aligned}$$

We can replace  $\lfloor m_i/2 \rfloor$  with  $\lceil m_i/2 \rceil$  because if  $m_i$  is odd the additional summand equals zero. The first inequality uses simple arithmetic operations. The second inequality uses the classical lower bound  $(\ln x + \frac{1}{2}) < H_x$  of harmonic numbers. The last inequality holds for  $m_i \geq 4$ . For  $m_i < 4$  the result becomes negative and is thus still a correct lower bound for the number of comparisons. We obtain:

$$\begin{aligned}
 \underbrace{\sum_{i=1}^k (m_i + 1) \cdot \ln \frac{m_i + 1}{2.5\sqrt{e}}}_{\text{comparisons within colorless ranges}} &+ \underbrace{\frac{\sigma k - 5n}{8}}_{\text{comparisons for colored nodes}} \geq n \cdot \ln \frac{n}{2.5\sqrt{e}k} + \frac{\sigma k - 5n}{8} \\
 &= n \cdot \ln \frac{\sigma}{2.5\sqrt{e}x} + \frac{xn - 5n}{8} \\
 &= n \cdot \ln \sigma + n \cdot \left( \frac{x - 5}{8} - \ln 2.5\sqrt{e}x \right) \\
 &> n \cdot \ln \sigma - 3.12074n
 \end{aligned}$$

The first step follows from  $\sum_{i=1}^k (m_i + 1) = n$  and the log sum inequality (see [104, Theorem 2.7.1]). In the second step we replace  $k$  by using  $x = \frac{\sigma k}{n}$ . The third step uses simple arithmetic operations. The last step is reached by substituting  $x = 8$ , which minimizes the equation. Finally, we assumed that  $\sigma$  is divisible by 4. We account for this by adjusting the lower bound to  $n \ln(\sigma - 3) - 3.12074n$ , which is larger than  $n \ln \sigma - 3.6n$  for  $\sigma \geq 8$ .

**Theorem 4.1.** *For any integers  $n$  and  $\sigma$  with  $8 \leq \sigma \leq n$ , there is no deterministic algorithm that performs at most  $n \ln \sigma - 3.6n = \mathcal{O}(n \ln \sigma)$  comparisons in the worst case, and determines whether a length- $n$  string with at most  $\sigma$  distinct symbols from a general unordered alphabet is square-free.*

## 4 Upper Bound

In this section, we consider the problem of testing square-freeness of a given string. We introduce an algorithm that decides whether or not a string is square-free using only  $\mathcal{O}(n \lg \sigma)$  comparisons, matching the lower bound from Section 3.2. Note that this algorithm is not yet time efficient because, apart from the performed symbol comparisons, it uses other operations that are expensive in the Word RAM model. A time efficient implementation of the algorithm will be presented in Section 5, where we first achieve  $\mathcal{O}(n \lg \sigma + n \log^* n)$  time, and then improve this to  $\mathcal{O}(n \lg \sigma)$  time. In Section 6, we generalize the result to compute all runs in the same time complexity.

## 4.1 Sparse Suffix Trees and Difference Covers

**Lemma 4.13.** *The sparse suffix tree containing any  $b$  suffixes  $T[i_1..n], \dots, T[i_b..n]$  of  $T[1..n]$  can be constructed using  $\mathcal{O}(b\sigma \log b)$  comparisons plus  $\mathcal{O}(n)$  comparisons shared by all invocations of the lemma.*

*Proof.* We maintain a union-find structure over the positions of  $T[1..n]$ . Initially, each position is in a separate component. Before issuing a query  $T[x] \stackrel{?}{=} T[y]$ , we check if  $x$  and  $y$  are in the same component of the union-find structure, and if so immediately return that  $T[x] = T[y]$  without performing any comparisons. Otherwise, we issue the query and if it returns that  $T[x] = T[y]$  we merge the components of  $x$  and  $y$ . Thus, the total number of issued queries with positive answer, over all invocations of the lemma, is less than  $n$ , and it remains to bound the number of issued queries with negative answer.

We insert the suffixes  $T[i_j..n]$  one-by-one into an initially empty sparse suffix tree. To insert the next suffix, we descend from the root of the tree to identify the node  $u$  that corresponds to the longest common prefix between  $T[i_j..n]$  and any of the already inserted suffixes. We then make  $u$  explicit unless it is explicit already, and add an edge from  $u$  to a new leaf corresponding to the whole  $T[i_j..n]$ . We say that the insertion procedure terminates at  $u$ . Node  $u$  can be identified with only  $\mathcal{O}(\sigma \log b)$  comparisons with negative answers as follows. Let  $v$  be the current node (initially, the root of the tree), and let  $v_1, \dots, v_d$  be its children, where  $d \leq \sigma$ . Here,  $v$  can be either explicit or implicit, in the latter case  $d = 1$ . We arrange the children of  $v$  so that the number of leaves in the subtree rooted at  $v_1$  is at least as large as the number of leaves in the subtree rooted at any other child of  $v$ . Then, we compare the character on the edge leading to  $v_1$  with the corresponding character of the current suffix. If they are equal we continue with  $v_1$ , otherwise we compare the characters on the edges leading to  $v_2, \dots, v_d$  with the corresponding character of the current suffix one-by-one. Then, we either continue with some  $v_j$ ,  $j \geq 2$ , or terminate at  $v$ . To bound the number of comparisons with negative answer, observe that such comparisons only occur when we either terminate at  $v$  or continue with  $v_j$ ,  $j \geq 2$ . Whenever we continue with  $v_j$ ,  $j \geq 2$ , the number of leaves in the current subtree rooted at  $v_j$  decreases at least by a factor of 2 compared to subtree rooted at  $v$  (as the subtree rooted at  $v_1$  had the largest number of leaves). Thus, during the whole descent from the root performed during an insertion this can happen only at most  $1 + \log b$  times. Every time we do not continue in the subtree  $v_1$  we might have up to  $d \leq \sigma$  comparisons with negative answer, thus the total number of such comparisons is as claimed<sup>1</sup>.  $\square$

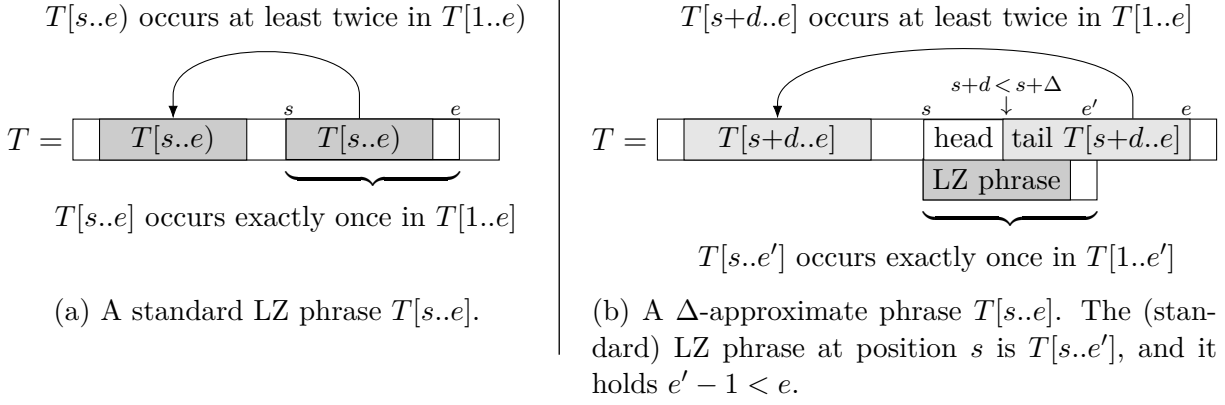
Now we describe the sample positions that we will later use to compute the approximate LZ factorisation. A set  $\mathbf{S} \subseteq \mathbb{N}$  is called a  $t$ -cover of  $\{1, \dots, n\}$  if there is a constant-time computable function  $h$  such that, for any  $1 \leq i, j \leq n - t + 1$ , we have  $0 \leq h(i, j) < t$  and  $i + h(i, j), j + h(i, j) \in \mathbf{S}$ . A possible construction of  $t$ -covers is given by the lemma below, and visualized in Fig. 4.2.

**Lemma 4.14.** *For any  $n$  and  $t \leq n$ , there exists a  $t$ -cover  $\mathbf{D}(t)$  of  $\{1, \dots, n\}$  with size  $\mathcal{O}(n/\sqrt{t})$ . Furthermore, its elements can be enumerated in time proportional to their number.*

*Proof.* We use the well-known combinatorial construction known as difference covers, see e.g. [29]. Let  $r = \lfloor \sqrt{t} \rfloor$  and define  $\mathbf{D}(t) = \{i \in \{1, \dots, n\} : i \bmod r = 0 \text{ or } i \bmod r^2 \in \{0, \dots, r-1\}\}$ . By definition,  $|\mathbf{D}(t)| \leq \lfloor n/r \rfloor + \lfloor n/r^2 \rfloor r = \mathcal{O}(n/r) = \mathcal{O}(n/\sqrt{t})$ . The function  $h(i, j)$  is defined as  $a + b \cdot r$ , where  $a = (r - i) \bmod r$  and  $b = (r - \lfloor (j + a)/r \rfloor) \bmod r$ .

<sup>1</sup>In the descent, if all children are sorted according to their subtree size, the number of comparisons decreases to  $\mathcal{O}(b(\sigma/\log \sigma) \log b)$ , but this appears irrelevant for our final algorithm.




 Figure 4.2: Positions in a  $\Delta$ -difference cover.

 Figure 4.3: Illustration of the definition of a LZ-phrase and a  $\Delta$ -approximate phrase.

Note that  $i + h(i, j) \leq n$  and  $j + h(i, j) \leq n$ . Then,  $i + (a + b \cdot r) \equiv 0 \pmod{r}$ , while  $\lfloor (j + (a + b \cdot r))/r \rfloor = \lfloor (j + a)/r + b \rfloor \equiv 0 \pmod{r}$  implies  $j + h(i, j) \pmod{r^2} \in \{0, \dots, r-1\}$ , thus  $i + h(i, j), j + h(i, j) \in \mathbf{D}(t)$  as required.  $\square$

## 4.2 Detecting Squares with a $\Delta$ -Approximate LZ Factorisation

A crucial notion in our algorithm is the following variation on the standard Lempel-Ziv factorisation:

**Definition 4.15** ( $\Delta$ -approximate LZ factorisation). *For a positive integer parameter  $\Delta$ , the fragment  $T[s..e]$  is a  $\Delta$ -approximate LZ phrase if it can be split into a head and a tail  $T[s..e] = \text{head}(T[s..e])\text{tail}(T[s..e])$  such that  $|\text{head}(T[s..e])| < \Delta$  and additionally*

- *$\text{tail}(T[s..e])$  is either empty or occurs at least twice in  $T[1..e]$ , and*
- *the unique (standard) LZ phrase  $T[s..e']$  starting at position  $s$  satisfies  $e' - 1 \leq e$ .*

*In a  $\Delta$ -approximate LZ factorisation  $T = b_1 b_2 \dots b_z$ , each factor  $b_i$  is a  $\Delta$ -approximate phrase  $T[s..e]$  with  $s = 1 + \sum_{j=1}^{i-1} |b_j|$  and  $e = \sum_{j=1}^i |b_j|$ .*

Note that a standard LZ phrase is not a  $\Delta$ -approximate phrase. Also, while the LZ phrase starting at each position (and thus also the LZ factorisation) is uniquely defined, there may be multiple different  $\Delta$ -approximate phrases starting at each position. This also means that a single string can have multiple different  $\Delta$ -approximate factorisations. The definitions of both standard and  $\Delta$ -approximate LZ phrases are illustrated in Fig. 4.3.

The intuition behind the above definition is that constructing the  $\Delta$ -approximate LZ factorisation becomes easier for larger values of  $\Delta$ . In particular, for  $\Delta = n$  one phrase is enough. We formalise this in the following lemma, which is made more general for the purpose of obtaining the final result in this section.

**Lemma 4.16.** *For any parameter  $\Delta \in [1, m]$ , a  $\Delta$ -approximate LZ factorisation of any fragment  $T[x..y]$  of length  $m$  can be computed with  $\mathcal{O}(m\sigma \log m / \sqrt{\Delta})$  comparisons plus  $\mathcal{O}(n)$  comparisons shared by all invocations of the lemma.*

*Proof.* By Lemma 4.14, there exists a  $\Delta$ -cover  $\mathbf{D}(\Delta)$  of  $\{1, \dots, n\}$  with size  $\mathcal{O}(n/\sqrt{\Delta})$ . Let  $S = \mathbf{D}(\Delta) \cap \{x, x+1, \dots, y\}$ . Let  $S = \{i_1, i_2, \dots, i_b\}$ . It is straightforward to verify that the construction additionally guarantees  $b = \mathcal{O}(m/\sqrt{\Delta})$ . We apply Lemma 4.13 on the suffixes  $T[i_1..n], \dots, T[i_b..n]$  to obtain their sparse suffix tree  $T$  with  $\mathcal{O}(b\sigma \log b)$  comparisons plus  $\mathcal{O}(n)$  comparisons shared by all invocations of the lemma.  $T$  allows us to obtain the longest common prefix of any two fragments  $T[i..y]$  and  $T[j..y]$ , for  $i, j \in S$ , with no additional comparisons. By the properties of  $\mathbf{D}(\Delta)$ , for any  $i, j \in \{x, x+1, \dots, y-\Delta+1\}$  we have  $0 \leq h(i, j) < \Delta$  and  $i + h(i, j), j + h(i, j) \in S$ .

We compute the  $\Delta$ -approximate LZ factorisation of  $T[x..y]$  phrase-by-phrase. Denoting the remaining suffix of the whole  $T[x..y]$  by  $T[x'..y]$ , we need to find  $x' \leq y' \leq y$  such that  $T[x'..y']$  is a  $\Delta$ -approximate phrase. This is done as follows. We iterate over every  $x' \leq x'' < x' + \Delta$  such that  $x'' \in S$ . For every such  $x''$ , we consider every  $x \leq a' < x'$  such that  $a' \in S$ , and compute the length  $\ell$  of the longest common prefix of  $T[x''..y]$  and  $T[a'..y]$ . Among all such  $x'', a'$  we choose the pair that results in the largest value of  $x'' - x' + \ell - 1$  and choose the next phrase to be  $T[x'..(x'' + \ell - 1)]$ , with the head being  $T[x'..(x'' - 1)]$  and the tail  $T[x''..(x'' + \ell) - 1]$ . Finally, if there is no such pair, or the value of  $x'' - x' + \ell - 1$  corresponding to the found pair is less than  $\Delta - 2$ , we take the next phrase to be  $T[x'..\min\{x' + \Delta - 1, y\}]$  (with empty tail). Selecting such a pair requires no extra comparisons, as for every  $x'', a' \in S$  we can use the sparse suffix tree to compute  $\ell$ . While it is clear that the generated  $\Delta$ -approximate phrase has the required form, we need to establish that it is sufficiently long.

Let  $T[x'..y'']$  be the (unique) standard LZ phrase of  $T[x..y]$  that is prefix of  $T[x'..y]$ . If  $y'' < x' + \Delta - 1$  then we only need to ensure that the generated  $\Delta$ -approximate phrase is of length at least  $\min\{\Delta - 1, y - x' + 1\}$ , which is indeed the case. Therefore, it remains to consider the situation when  $y'' \geq x' + \Delta - 1$ . Let  $T[a..b]$  be the previous occurrence of  $T[x'..(y'' - 1)]$  in  $T[x..y]$  (because  $T[x'..y'']$  is a phrase this is well defined). Thus,  $T[a..b] = T[x'..(y'' - 1)]$  and  $a < x'$ . Because  $y'' \geq x' + \Delta - 1$  and  $y'' \leq y$ , as explained above  $0 \leq h(a, x') < \Delta$  and  $a + h(a, x'), x' + h(a, x') \in S$ . We will consider  $x'' = x' + h(a, x')$  and  $a' = a + h(a, x')$  in the above procedure. Next,  $T[a'..b] = T[x''..(y'' - 1)]$ , so when considering this pair we will obtain  $\ell \geq |T[x''..(y'' - 1)]|$ . Thus, for the found pair we will have  $x'' + \ell - 1 \geq y'' - 1$  as required in the definition of a  $\Delta$ -approximate phrase.  $\square$

Next, we show that even though the  $\Delta$ -approximate LZ factorisation does not capture all distinct squares, as it is the case for the standard LZ factorisation, it is still helpful in detecting all sufficiently long squares. A crucial component is the following property of the  $\Delta$ -approximate LZ factorisation.

**Lemma 4.17.** *Let  $b_1 b_2 \dots b_z$  be a  $\Delta$ -approximate LZ factorisation of a string  $T$ . For every square  $T[s..s + 2\ell - 1]$  of length  $2\ell \geq 8\Delta$ , there is at least one phrase  $b_i$  with  $|\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$  such that  $\text{tail}(b_i)$  and the right-hand side  $T[s + \ell..s + 2\ell - 1]$  of the square intersect.*

*Proof.* Assume that all tails that intersect  $T[s + \ell..s + 2\ell - 1]$  are of length less than  $\frac{\ell}{4}$ , then the respective phrases of these tails are of length at most  $\frac{\ell}{4} + \Delta - 1$  (because each head is of length less than  $\Delta$ ). This means that  $T[s + \ell..s + 2\ell - 1]$  intersects at least  $\left\lceil \ell / (\frac{\ell}{4} + \Delta - 1) \right\rceil \geq \left\lceil \ell / (\frac{\ell}{2} - 1) \right\rceil = 3$  phrases. Thus there is some phrase  $b_i = T[x..y]$  properly contained in  $T[s + \ell..s + 2\ell - 1]$ , formally  $s + \ell < x \leq y < s + 2\ell - 1$ . However, this contradicts the definition of the  $\Delta$ -approximate LZ factorisation because  $T[x..s + 2\ell]$  is the prefix of a standard LZ phrase (due to

$T[x..s + 2\ell - 1] = T[x - \ell..s + \ell - 1]$ ), and the  $\Delta$ -approximate phrase  $b_i = T[x..y]$  must satisfy  $y \geq s + 2\ell - 1$ . The contradiction implies that  $T[s + \ell..s + 2\ell - 1]$  intersects a tail of length at least  $\frac{\ell}{4} \geq \Delta$ .  $\square$

**Lemma 4.18.** *Given a  $\Delta$ -approximate LZ factorisation  $T = b_1 b_2 \dots b_z$ , we can detect a square of size  $\geq 8\Delta$  in  $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)| + z\right)$  time and  $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)|\right)$  comparisons.*

*Proof.* We consider each phrase  $b_i = T[a_1..a_3]$  with  $\text{head}(b_i) = T[a_1..a_2 - 1]$  and  $\text{tail}(b_i) = T[a_2..a_3]$  separately. Let  $k = |\text{tail}(b_i)|$ . If  $k \geq \Delta$ , we apply Lemma 4.7 to  $x_1 = T[a_2 - 8k..a_2 - 1]$  and  $y_1 = T[a_2..a_3 + 4k - 1]$ , as well as  $x_2 = T[a_2 - 8k..a_3 - 1]$  and  $y_2 = T[a_3..a_3 + 4k - 1]$  trimmed to  $T[1..n]$ . This takes  $\mathcal{O}(|\text{tail}(b_i)|)$  time and comparisons, or  $\mathcal{O}\left(\sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)|\right)$  time and comparisons for all phrases. We need additional  $\mathcal{O}(z)$  time to check if  $k \geq \Delta$  for each phrase.

Now we show that the described strategy detects a square of size at least  $8\Delta$ . Let  $T[s..s + 2\ell - 1]$  be any such square. Due to Lemma 4.17, the right-hand side  $T[s + \ell..s + 2\ell - 1]$  of this square intersects some tail  $\text{tail}(b_i) = T[a_2..a_3]$  of length  $k = |\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$ . Due to the intersection, we have  $a_2 \leq s + 2\ell - 1$  and  $a_3 \geq s + \ell$ . Thus, when processing  $b_i$  and applying Lemma 4.7, the starting position of  $x_1$  and  $x_2$  satisfies  $a_2 - 8k \leq s + 2\ell - 1 - 8\frac{\ell}{4} = s - 1$ , while the end position of  $y_1$  and  $y_2$  satisfies  $a_3 + 4k - 1 \geq s + \ell + 4\frac{\ell}{4} - 1 = s + 2\ell - 1$ . Therefore, the square is entirely contained in the respective fragments corresponding to  $x_1 y_1$  and  $x_2 y_2$ . If  $s < a_2 \leq s + 2\ell - 1$ , we find the square with our choice of  $x_1$  and  $y_1$ . If  $s < a_3 \leq s + 2\ell - 1$ , we find the square with our choice of  $x_2$  and  $y_2$ . Otherwise,  $T[s..s + 2\ell - 1]$  is entirely contained in tail  $T[a_2..a_3]$ , and we find another occurrence of the square further to the left.  $\square$

### 4.3 Simple Algorithm for Detecting Squares

Now we have all the tools to introduce our simple method for testing square-freeness of  $T[1..n]$  using  $\mathcal{O}(n(\log \sigma + \log \log n))$  comparisons, assuming that  $\sigma$  is known in advance. Let  $\Delta = (\sigma \log n)^2$ . We partition  $T[1..n]$  into blocks of length  $8\Delta$ , and denote the  $k^{\text{th}}$  block by  $B_k$ . A square of length at most  $8\Delta$  can be found by invoking Theorem 4.6 on  $B_1 B_2$ ,  $B_2 B_3$ , and so on. This takes  $\mathcal{O}(\Delta \log \Delta) = \mathcal{O}(\Delta(\log \sigma + \log \log n))$  comparisons for each pair of adjacent blocks, or  $\mathcal{O}(n(\log \sigma + \log \log n))$  comparisons in total. It remains to test for squares of length exceeding  $8\Delta$ . This is done by first invoking Lemma 4.16 to compute a  $\Delta$ -approximate LZ factorisation of  $T[1..n]$  with  $\mathcal{O}(n\sigma \log n / \sqrt{\Delta}) = \mathcal{O}(n)$  comparisons, and then using Lemma 4.18, which adds another  $\mathcal{O}(n)$  comparisons. The total number of comparisons is dominated by the  $\mathcal{O}(n(\log \sigma + \log \log n))$  comparisons needed to apply Theorem 4.6 to the block pairs.

### 4.4 Improved Algorithm for Detecting Squares

We are now ready to describe the algorithm that uses only  $\mathcal{O}(n \log \sigma)$  comparisons without knowing the value of  $\sigma$ . Intuitively, we will proceed in phases, trying to “guess” the value of  $\sigma$ . We first observe that Lemma 4.16 can be extended to obtain the following.

**Lemma 4.19.** *There is an algorithm that, given any parameter  $\Delta \in [1, m]$ , estimate  $\tilde{\sigma}$  and fragment  $T[x..y]$  of length  $m$ , uses  $\mathcal{O}(m\tilde{\sigma} \log m / \sqrt{\Delta})$  comparisons plus  $\mathcal{O}(n)$  comparisons shared by all invocations of the lemma, and either computes a  $\Delta$ -approximate LZ factorisation of  $T[x..y]$  or determines that  $\sigma > \tilde{\sigma}$ .*

*Proof.* We run the procedure described in the proof of Lemma 4.16 and keep track of the number of comparisons with negative answer. As soon as it exceeds  $\mathcal{O}(m\tilde{\sigma} \log m / \sqrt{\Delta})$  (where the

constant follows from the complexity analysis) we know that necessarily  $\sigma > \tilde{\sigma}$ , so we can terminate. Otherwise, the algorithm obtains a  $\Delta$ -approximate LZ factorisation with  $\mathcal{O}(m\tilde{\sigma} \log m / \sqrt{\Delta})$  comparisons. Comparisons with positive answer are paid for globally.  $\square$

Now we describe how to find any square using  $\mathcal{O}(n \lg \sigma)$  comparisons. We define the sequence  $\sigma_t = 2^{\lceil \log \log n \rceil - t}$ , for  $t = 0, 1, \dots, \lceil \log \log n \rceil$ . We observe that  $\sigma_{t-1} = (\sigma_t)^2$ , and proceed in phases corresponding to the values of  $t$ . In the  $t^{\text{th}}$  phase we are guaranteed that any square of length at least  $(\sigma_t)^2$  has been already detected, and we aim to detect square of length less than  $(\sigma_t)^2$ , and at least  $\sigma_t$ . We partition the whole  $T[1..n]$  into blocks of length  $(\sigma_t)^2$ , and denote the  $k^{\text{th}}$  block by  $B_k$ . A square of length less than  $(\sigma_t)^2$  is fully contained within some two consecutive blocks  $B_i B_{i+1}$ , hence we consider each such pair  $B_1 B_2$ ,  $B_2 B_3$ , and so on. We first apply Lemma 4.19 with  $\Delta = \sigma_t/8$  and  $\tilde{\sigma} = (\sigma_t)^{1/4} / \log(\sigma_t)$  to find an  $(\sigma_t/8)$ -approximate LZ factorisation of the corresponding fragment of  $T[1..n]$ , and then use Lemma 4.18 to detect squares of length at least  $\sigma_t$ . We cannot always afford to apply Lemma 4.18 to all block pairs. Thus, we have to deactivate some of the blocks, which we explain when analysing the number of comparisons performed by the algorithm. If any of the calls to Lemma 4.19 in the current phase detects that  $\sigma > \tilde{\sigma}$ , we switch to applying Theorem 4.6 on every pair of blocks  $B_i B_{i+1}$  of the current phase and then terminate the whole algorithm.

We now analyse the total number of comparisons, ignoring the  $\mathcal{O}(n)$  comparisons shared by all invocations of Lemma 4.19. Throughout the  $t^{\text{th}}$  phase, we use  $\mathcal{O}(n \cdot \tilde{\sigma} \log \sigma_t / \sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4} / \log(\sigma_t) \cdot \log(\sigma_t) / \sqrt{\sigma_t}) = \mathcal{O}(n / (\sigma_t)^{1/4})$  comparisons to construct the  $\Delta$ -approximate factorisations (using Lemma 4.19) until we either process all pairs of blocks or detect that  $\sigma > (\sigma_t)^{1/4} / \log(\sigma_t)$ . In the latter case, we finish off the whole computation with  $\mathcal{O}(n \log(\sigma_t))$  comparisons (using Theorem 4.6), and by assumption on  $\sigma$  this is  $\mathcal{O}(n \log \sigma)$  as required. Until this happens (or until we reach phase  $t = \lceil \log \log n \rceil - 3$  where  $\sigma_t \leq 256$ ), we use  $\mathcal{O}(\sum_{t=0}^{t'} n / (\sigma_t)^{1/4})$  comparisons to construct the  $\Delta$ -approximate factorisations, for some  $0 \leq t' \leq \lceil \log \log n \rceil$ . To analyse the sum, we need the following bound (made more general for the purpose of the next section).

**Lemma 4.20.** *For any  $0 \leq x \leq y$  and  $c \geq 0$  we have  $\sum_{i=x}^y 2^{ic} / 2^{2^i} = \mathcal{O}(2^{xc} / 2^{2^x})$ .*

*Proof.* We observe that the sequence of exponents  $2^i$  is strictly increasing from  $i = 0$ , hence

$$\sum_{i=x}^y \frac{2^{ic}}{2^{2^i}} \leq \sum_{i=2^x}^{2^y} \frac{i^c}{2^i} \leq \sum_{i=2^x}^{\infty} \frac{i^c}{2^i} = \sum_{i=0}^{\infty} \frac{(2^x + i)^c}{2^{(2^x + i)}} \leq \sum_{i=0}^{\infty} \frac{2^{xc} \cdot (i+1)^c}{2^{(2^x + i)}} = \frac{2^{xc}}{2^{2^x}} \cdot \sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}.$$

$\sum_{i=0}^{\infty} \frac{(i+1)^c}{2^i}$  is a series of positive terms, we thus use Alembert's ratio test  $\frac{(i+2)^c}{2^{i+1}} \cdot \frac{2^i}{(i+1)^c} = \frac{1}{2} \frac{(i+2)^c}{(i+1)^c}$  which tends to  $\frac{1}{2}$  when  $i$  goes to the infinity, thus the series converges to a constant.  $\square$

**Corollary 4.21.** *For any  $0 \leq t' \leq \lceil \log \log n \rceil$ , it holds that  $\sum_{t=0}^{t'} n \cdot \text{polylog}(\sigma_t) / (\sigma_t)^{1/4} = \mathcal{O}(n)$ .*

*Proof.* We have to show that  $\sum_{t=0}^{t'} n \log^c(\sigma_t) / (\sigma_t)^{1/4} = \mathcal{O}(n)$  for any constant  $c \geq 0$ . We achieve this by splitting the sum and applying Lemma 4.20.

$$\begin{aligned} \sum_{t=0}^{t'} \frac{n \log^c(\sigma_t)}{(\sigma_t)^{1/4}} &\leq \sum_{t=0}^{\lceil \log \log n \rceil} \frac{n \cdot (2^{\lceil \log \log n \rceil - t})^c}{(2^{2^{\lceil \log \log n \rceil - t}})^{1/4}} = \sum_{t=0}^{\lceil \log \log n \rceil} \frac{n \cdot (2^t)^c}{(2^{2^t})^{1/4}} \\ &= n \cdot \sum_{t=0}^{\lceil \log \log n \rceil} \frac{2^{tc}}{2^{2^{t-2}}} = n \cdot \left( \frac{1}{2^{2^{-2}}} + \frac{2^c}{2^{2^{-1}}} + 4 \cdot \sum_{t=0}^{\lceil \log \log n \rceil - 2} \frac{2^{tc}}{2^{2^t}} \right) = \mathcal{O}(n) \end{aligned} \quad \square$$

Thus, all invocations of Lemma 4.19 cause  $\mathcal{O}(\sum_{t=0}^{t'} n / (\sigma_t)^{1/4}) = \mathcal{O}(n)$  comparisons.

## Deactivating Block Pairs

It remains to analyse the number of comparisons used by Lemma 4.18 throughout all phases. As mentioned earlier, we cannot actually afford to apply Lemma 4.18 to all block pairs. Thus, we introduce a mechanism that deactivates some of the pairs.

First, note that there are  $\mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^2) \subseteq \mathcal{O}(\sum_{t=0}^{t'} n/(\sigma_t)^{1/4}) = \mathcal{O}(n)$  block pairs in all phases. For each pair, we store whether it has been deactivated or not, where being deactivated broadly means that we do not have to investigate the pair because it does not contain a leftmost distinct square. For each block pair  $B_i B_{i+1}$  in the current phase  $t$ , we first check if it has been marked as deactivated. If not, we also check if it has been *implicitly* deactivated, i.e., if any of the two pairs from the previous phase that contain  $B_i B_{i+1}$  are marked as deactivated. If  $B_i B_{i+1}$  has been implicitly deactivated, then we mark it as deactivated and do not apply Lemma 4.19 and Lemma 4.18 (the implicit deactivation serves the purpose of propagating the deactivation to all later phases). Note that if some position of the string is not contained in any active block pair in some phase, then it is also not contained in any active block pair in all later phases. This is because it always holds that  $\sigma_{t-1} = (\sigma_t)^2$  (with no rounding required), which guarantees that block boundaries of earlier phases do not intersect blocks of later phases.

We only apply Lemma 4.19 and then Lemma 4.18 to  $B_i B_{i+1}$  if the pair has neither explicitly nor implicitly been deactivated. When applying Lemma 4.18, a tail  $T[a..a + \ell)$  contributes  $\mathcal{O}(\ell)$  comparisons if  $\ell \geq \Delta = \sigma_t/8$  (and otherwise it contributes no comparisons). As the whole  $T[a..a + \ell)$  occurs earlier, it cannot contain the leftmost occurrence of a square in the whole  $T$ . Thus, any block pair (of any phase) contained in  $T[a..a + \ell)$  also cannot contain such an occurrence, and thus such block pairs can be deactivated.

The mechanism used for deactivation works as follows. Let  $T[a..a + \ell)$  be a tail contributing  $\mathcal{O}(\ell)$  comparisons with  $\ell \geq \Delta = \sigma_t/8$  in phase  $t$ . We mark all block pairs of phase  $t + 2$  that are entirely contained in  $T[a..a + \ell)$  as deactivated. Note that blocks in phase  $t + 2$  are of length  $\sqrt{\sigma_t}$ , and consider the fragment  $T[a + 2\sqrt{\sigma_t}..a + \ell - 2\sqrt{\sigma_t})$ . In phase  $t + 2$ , and by implicit deactivation in all later phases, this fragment overlaps (either partially or fully) only block pairs that have been deactivated. Thus, after phase  $t + 1$ , we will never inspect any of the symbols in  $T[a + 2\sqrt{\sigma_t}..a + \ell - 2\sqrt{\sigma_t})$  again. We say that tail  $T[a..a + \ell)$  deactivated the fragment of length  $\ell - 4\sqrt{\sigma_t} = \Omega(\ell)$ , which is positive until phase  $t = \lceil \log \log n \rceil - 3$  because  $\sigma_t > 256$ . Since the number of deactivated positions is linear in the number of comparisons that the tail contributes to Lemma 4.18, it suffices to show that each position gets deactivated at most a constant number of times. In a single phase, any position gets deactivated at most twice. This is because the tails of each factorisation do not overlap by definition, but the tails of the two factorisations of adjacent block pairs  $B_i B_{i+1}$  and  $B_{i+1} B_{i+2}$  can overlap. If a position gets deactivated for the first time in phase  $t$ , then (as explained earlier) we will not consider it in any of the phases  $t' \geq t + 2$ . Thus, it can only be that we deactivate the position again in phase  $t + 1$ , but not in any later phases. In total, each position gets deactivated at most four times. Hence Lemma 4.18 contributes  $\mathcal{O}(n)$  comparisons in total.

We have shown:

**Theorem 4.2.** *Testing square-freeness of a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be done with  $\mathcal{O}(n \log \sigma)$  comparisons.*

## 5 Algorithm

In this section we show how to implement the approach described in the previous section to work in  $\mathcal{O}(n \log \sigma)$  time. The main difficulty is in efficiently implementing the sparse suffix tree construction algorithm, and then computing a  $\Delta$ -approximate factorisation. We first how to

obtain an  $\mathcal{O}(n \log \sigma + n \log^* n)$  time algorithm that still uses only  $\mathcal{O}(n \log \sigma)$  comparisons, and then further improve its running time to  $\mathcal{O}(n \log \sigma)$ .

## 5.1 Constructing the Suffix Tree and $\Delta$ -Approximate Factorisation

To give an efficient algorithmic construction of the sparse suffix tree from Lemma 4.13, we will use a restricted version of LCEs, where a query  $\text{ShortLCE}_x(i, j)$  (for any positive integer  $x$ ) returns  $\min(x, \text{LCE}(i, j))$ . The following result was given by Gawrychowski et al. [217]:

**Lemma 4.22** (Lemma 14 in [217]). *For a length- $n$  string over a general unordered alphabet<sup>2</sup>, a sequence of  $q$  queries  $\text{ShortLCE}_{4^{k_i}}$  for  $i \in \{1, \dots, q\}$  can be answered online in total time  $\mathcal{O}(n \log^* n + s)$  and  $\mathcal{O}(n + q)$  comparisons<sup>3</sup>, where  $s = \sum_{i=1}^q (k_i + 1)$ .*

In the lemma, apart from the  $\mathcal{O}(n \log^* n)$  time, each  $\text{ShortLCE}_{4^{k_i}}$  query accounts for  $\mathcal{O}(k_i + 1)$  time. Note that we can answer the queries online, without prior knowledge of the number and length of the queries. Also, computing an LCE in a fragment  $T[x..y]$  of length  $m$  trivially reduces to a  $\text{ShortLCE}_{4^{\lceil \log_4 m \rceil}}$  query on  $T$ . Thus, we have:

**Corollary 4.23.** *A sequence of  $q$  longest common extension queries on a fragment  $T[x..y]$  of length  $m$  over a general unordered alphabet can be answered in  $\mathcal{O}(q \log m)$  time plus  $\mathcal{O}(n \log^* n)$  time shared by all invocations of the lemma. The number of comparisons is  $\mathcal{O}(q)$ , plus  $\mathcal{O}(n)$  comparisons shared by all invocations of the lemma.*

While constructing the sparse suffix tree, we will maintain a heavy-light decomposition using a rebuilding scheme introduced by Gabow [40]. Let  $L(u)$  denote the number of leaves in the subtree of a node  $u$ . We use the following recursive construction of a heavy-light decomposition: Starting from a node  $r$  (initially the root of the tree), we find the deepest descendant node  $e$  such that  $L(e) \geq \frac{5}{6}L(r)$  (possibly  $e = r$ ). The path  $p$  from the root  $r(p) = r$  to  $e(p) = e$  is a heavy path. Any edge  $(u, v)$  on this path satisfies  $L(v) \geq \frac{5}{6}L(u)$ , and we call those edges *heavy*. As a consequence, a node  $u$  can have at most one child  $v$  such that  $(u, v)$  is heavy. For each edge  $(u, v)$  where  $u$  is on the heavy path and  $v$  is not, we recursively build a new heavy path construction starting from  $v$ .

When inserting a new suffix in our tree, we keep track of the insertion in the following way: for every root of a heavy path, we maintain  $I(u)$  the number of insertions made in the subtree of  $u$  since we built the heavy-light decomposition of this subtree. When  $I(u) \geq \frac{1}{6}L(u)$  we recalculate the values of  $L(v)$  for all nodes  $v$  in the subtree of  $u$  and rebuild the heavy-light decomposition for the subtree of  $u$ .

This insures that, despite insertion, for any heavy path starting at node  $r$  and a node  $u$  on that heavy path,  $L(e) \geq \frac{2}{3}L(r)$ . When crossing a non-heavy edge the number of nodes in the subtree reduces by a factor at least  $\frac{5}{6}$  which leads to the following property:

**Observation 4.24.** *The path from any node to the root crosses at most  $\mathcal{O}(\log m)$  heavy paths.*

<sup>2</sup>Lemma 14 in [217] does not explicitly mention that it works for general unordered alphabet. However, the proof of the lemma relies solely on equality tests.

<sup>3</sup>Lemma 14 in [217] does not explicitly mention that it requires  $\mathcal{O}(n + q)$  comparisons. However, they use a union-find approach where there can be at most  $\mathcal{O}(n)$  comparisons with outcome "equal", and each LCE query performs only one comparison with outcome "not-equal", similarly to what we describe in the proof of Lemma 4.13.

Additionally, rebuilding a subtree of size  $s$  takes  $\mathcal{O}(s)$  time and adding a suffix  $T[i_j..y]$  to the tree increases  $I(r)$  for each path  $p$  from the root  $r$  to the new leaf. Those are at most  $\mathcal{O}(\log m)$  nodes, and thus maintaining the heavy path decomposition takes amortized time  $\mathcal{O}(\log n)$  time per insertion.

With these building blocks now clearly defined, we are ready to describe the construction of the sparse suffix tree.

**Lemma 4.25.** *The sparse suffix tree containing any  $b$  suffixes  $T[i_1..y], \dots, T[i_b..y]$  of  $T[x..y]$  with  $m = |T[x..y]|$  can be constructed using  $\mathcal{O}(b\sigma \log b \log m)$  time plus  $\mathcal{O}(n \log^* n)$  time shared by all invocations of the lemma.*

*Proof.* As in the proof of Lemma 4.13, we consider the insertion of a suffix  $T[i_j..y]$  into the sparse suffix tree with suffixes  $T[i_1..y], T[i_2..y] \dots T[i_{j-1}..y]$ . At all times, we maintain the heavy path decomposition. Additionally, we maintain for each heavy path a predecessor data structure, where given some length  $\ell$ , we can quickly identify the deepest explicit node on the heavy path that spells a string of length at least  $\ell$ . The data structure can, e.g., be a balanced binary search tree with insertion and search operations in  $\mathcal{O}(\log b)$  time (the final sparse suffix tree and thus each heavy path contains  $\mathcal{O}(b)$  nodes). When rebuilding a subtree of the heavy path decomposition, we also have to rebuild the predecessor data structure for each of its heavy paths. Thus, rebuilding a size- $q$  subtree takes  $\mathcal{O}(q \log b)$  time (each node is on exactly one heavy path and has to be inserted into one predecessor data structure), and the amortized insertion time increases from  $\mathcal{O}(\log m)$  to  $\mathcal{O}(\log m \cdot \log b)$ . Whenever we insert a suffix, we make at most one node explicit, and thus have to perform at most one insertion into a predecessor data structure. The time for this is  $\mathcal{O}(\log b)$ , which is dominated by the previous term.

When inserting  $T[i_j..y]$ , we look for the node  $u$  corresponding to the longest common prefix between  $T[i_j..y]$  and the inserted suffixes, make  $u$  explicit if necessary and add a new leaf corresponding to  $T[i_j..y]$  attached to  $u$ . Let  $v$  be the current node (initialized by the root, and always an explicit node) and  $v_1, \dots, v_d$  be its (explicit) children. If there is a heavy edge  $(v, v_a)$  for  $1 \leq a \leq d$ , let  $p$  be the corresponding heavy path. For each heavy path  $p$ , we store the label of one leaf (i.e., the starting position of one suffix) that is contained in the subtree of  $e(p)$ . Thus, we can use Corollary 4.23 to compute the longest common extension between the string spelled by  $e(p)$  and  $T[i_j..y]$ . Now we use the predecessor data structure on the heavy path to find the deepest (either explicit or implicit) node  $v'$  on the path that spells a prefix of  $T[i_j..y]$ . If  $v'$  is implicit, we make it explicit and add the leaf. If  $v'$  is explicit and  $v' \neq v$ , we use  $v'$  as the new current node and continue. Otherwise, we have  $v' = v$ , i.e., the suffix does not belong to the subtree rooted in  $v_a$ . In this case, we issue  $d$  LCE queries between  $T[i_j..y]$  and each of the strings spelled by the nodes  $v_1, \dots, v_d$ . This either reveals that we can continue using one of the  $v_a$  as the new current node, or that we can create a new explicit node on some  $(v, v_a)$  edge and attach the leaf to it, or that we can simply attach a new leaf to  $v$ .

Now we analyse the time spent while inserting one suffix. We spent  $\mathcal{O}(b \cdot \log m \cdot \log b)$  total time for inserting  $\mathcal{O}(b)$  nodes into the dynamic heavy path decomposition and the predecessor data structures. In each step of the insertion process, we either (i) move as far as possible along some heavy path or (ii) move along some non-heavy edge. For (i), we issue one LCE query and one predecessor query. For (ii) we issue  $\mathcal{O}(\sigma)$  LCE queries. Due to Observation 4.24, both (i) and (ii) happen at most  $\mathcal{O}(\log b)$  times per suffix. Thus, for all suffixes, we perform  $\mathcal{O}(b \log b)$  predecessor queries and  $\mathcal{O}(b\sigma \log b)$  LCE queries. The total time is  $\mathcal{O}(b \log^2 b)$  for predecessor queries, and  $\mathcal{O}(b\sigma \log b \log m)$  for LCE queries (apart from the  $n \log^* n$  time shared by all invocations of Corollary 4.23).  $\square$

**Lemma 4.26.** *For any parameter  $\Delta \in [1, m]$ , a  $\Delta$ -approximate LZ factorisation of any fragment  $T[x..y]$  of length  $m$  can be computed in  $\mathcal{O}(m\sigma \log^2 m / \sqrt{\Delta})$  time plus  $\mathcal{O}(n \log^* n)$  time shared by all invocations of the lemma.*

*Proof.* Let  $T' = T[x..y]$ , and let  $\{i_1, i_2, \dots, i_b\}$  be a  $\Delta$ -cover of  $\{1, \dots, m\}$ , which implies  $b = \Theta(m/\sqrt{\Delta})$ . We obtain a sparse suffix tree of the suffixes  $T'[i_1..m], \dots, T'[i_b..m]$ , which takes  $\mathcal{O}(b\sigma \log b \log m) \subseteq \mathcal{O}(m\sigma \log^2 m / \sqrt{\Delta})$  time according to Lemma 4.25, plus  $\mathcal{O}(n \log^* n)$  time shared by all invocations of the lemma. Now we compute a  $\Delta$ -approximate LZ factorisation of  $T'$  from the sparse suffix tree in  $\mathcal{O}(b)$  time.

In the following proof, we use  $i_1, i_2, \dots, i_b$  interchangeably to denote both the difference cover positions, as well as their corresponding leaves in the sparse suffix tree. Assume that the order of difference cover positions is  $i_1 < i_2 < \dots < i_b$ . First, we determine for each  $i_k > i_1$ , the position  $\text{src}(i_k) = i_h$  and the length  $\text{len}(i_k) = \text{LCE}(i_h, i_k)$ , where  $i_h \in \{i_1, \dots, i_{k-1}\}$  is a position that maximizes  $\text{LCE}(i_h, i_k)$ . This is similar to what was done in [262] for the LZ77 factorisation. We start by assigning labels from  $\{1, \dots, b\}$  to the nodes of the sparse suffix tree. A node has label  $k$  if and only if  $i_k$  is its smallest descendant leaf. We assign the labels as follows. Initially, all nodes are unlabelled. We assign label 1 to each node on the path from  $i_1$  to the root. Then, we process the remaining leaves  $i_2, \dots, i_b$  in increasing order. For each  $i_k$ , we follow the path from  $i_k$  to the root. We assign label  $k$  to each unlabelled node that we encounter. As soon as we reach a node that has already been labelled, say, with label  $h$  and string-depth  $\ell$ , we are done processing leaf  $i_k$ . It should be easy to see that  $i_h$  is also exactly the desired index that maximizes  $\text{LCE}(i_h, i_k)$ , and we have  $\text{LCE}(i_h, i_k) = \ell$ . Thus, we have found  $\text{src}(i_k) = i_h$  and  $\text{len}(i_k) = \ell$ . The total time needed is linear in the number of sparse suffix tree nodes, which is  $\mathcal{O}(b)$ .

Finally, we obtain a  $\Delta$ -approximate LZ factorisation using  $\text{src}$  and  $\text{len}$ . The previously computed values can be interpreted as follows:  $i_k$  could become the starting position of a length- $\text{len}(i_k)$  tail (with previous occurrence at position  $\text{src}(i_k)$ ). For the  $\Delta$ -approximate LZ factorisation, we will create the factors greedily in a left-to-right manner. Assume that we already factorised  $T'[1..s-1]$ , then the next phrase starts at position  $s$ , and thus the next tail starts within  $T'[s..s+\Delta)$  (as a reminder, the head is by definition shorter than  $\Delta$ ). Let  $S = \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s+\Delta-1\}$ . If there is no  $i_k \in S$  with  $i_k + \text{len}(i_k) > s + \Delta - 1$ , then the next phrase is simply  $T'[s..\min(|T'|, s + \Delta - 1))$  with empty tail. Otherwise, the next phrase has (possibly empty) head  $T'[s..i_k]$  and tail  $T'[i_k..i_k + \text{len}(i_k)]$  (with previous occurrence  $\text{src}(i_k)$ ), where  $i_k$  is chosen from  $S$  such that it maximizes  $i_k + \text{len}(i_k)$ . Creating the phrase in this way clearly takes  $\mathcal{O}(|S|)$  time. Since the next phrase starts at least at position  $s + \Delta - 1$ , none of the positions from  $S \setminus \{s + \Delta - 1\}$  will ever be considered as starting positions of other tails. Thus, every  $i_k$  is considered during the creation of at most two phrases, and the total time needed to create all phrases is  $\mathcal{O}(b)$ .

It remains to be shown that the computed factorisation is indeed a  $\Delta$ -approximate LZ factorisation, i.e., if we output a phrase  $T'[s..e]$ , then the unique (non-approximate) LZ phrase  $T'[s..e']$  starting at position  $s$  satisfies  $e' - 1 \leq e$ . First, note that for the created approximate phrases (except possibly the last phrase of  $T$ ) we have  $s + \Delta - 2 \leq e$ . Assume  $e' < s + \Delta$ , then clearly  $e' - 1 \leq e$ . Thus, we only have to consider  $e' > s + \Delta - 1$ . Since  $T'[s..e']$  is an LZ phrase, there is some  $s' < s$  such that  $\text{LCE}(s', s) = e' - s$ . Let  $h$  be the constant-time computable function that defines the  $\Delta$ -cover, and let  $i_{k'} = s' + h(s', s)$  and  $i_k = s + h(s', s)$ . Note that  $i_{k'} \in \{i_1, i_2, \dots, i_{k-1}\}$  and  $i_k \in \{i_1, i_2, \dots, i_b\} \cap \{s, \dots, s + \Delta - 1\}$ . Therefore, we have  $\text{len}(i_k) \geq \text{LCE}(i_{k'}, i_k) = \text{LCE}(s', s) - h(s', s) = (e' - s) - (i_k - s) = e' - i_k$ . While computing the  $\Delta$ -approximate phrase  $T'[s..e]$ , we considered  $i_k$  as the starting positions of the tail, which implies  $e \geq i_k + \text{len}(i_k) - 1 \geq e' - 1$ .  $\square$



**Lemma 4.27.** *There is an algorithm that, given any parameter  $\Delta \in [1, m]$ , estimate  $\tilde{\sigma}$  and fragment  $T[x..y]$  of length  $m$ , takes  $\mathcal{O}(m\tilde{\sigma} \log^2 m / \sqrt{\Delta})$  time plus  $\mathcal{O}(n \log^* n)$  time shared by all invocations of the lemma, and either computes a  $\Delta$ -approximate LZ factorisation of  $T[x..y]$  or determines  $\sigma > \tilde{\sigma}$ .*

*Proof.* We simply use Lemma 4.26 to compute the factorisation. In the first step, we have to construct the sparse suffix tree using the algorithm from Lemma 4.25. While this algorithm takes  $\mathcal{O}(m\sigma \log^2 m / \sqrt{\Delta})$  time, it is easy to see that a more accurate time bound is  $\mathcal{O}(md \log^2 m / \sqrt{\Delta})$ , where  $d$  is the maximum degree of any node in the sparse suffix tree. If during construction the maximum degree of a node becomes  $\tilde{\sigma} + 1$ , we immediately stop and return that  $\sigma > \tilde{\sigma}$ . Otherwise, we finish the construction in the desired time.  $\square$

Now we can describe the algorithm that detects squares in  $\mathcal{O}(n \lg \sigma + n \log^* n)$  time and  $\mathcal{O}(n \lg \sigma)$  comparisons. We simply use the algorithm from Section 4, but use Lemma 4.27 instead of Lemma 4.19. Next, we analyse the time needed apart from the  $\mathcal{O}(n \log^* n)$  time shared by all invocations of Lemma 4.27. Throughout the  $t^{\text{th}}$  phase, we use  $\mathcal{O}(n \cdot \tilde{\sigma} \cdot \log^2(\sigma_t) / \sqrt{\Delta}) = \mathcal{O}(n \cdot (\sigma_t)^{1/4} / \log(\sigma_t) \cdot \log^2(\sigma_t) / \sqrt{\sigma_t}) = \mathcal{O}(n \log(\sigma_t) / (\sigma_t)^{1/4})$  comparisons to construct all the  $\Delta$ -approximate factorisations. As before, if at any time we discover that  $\tilde{\sigma} > (\sigma_t)^{1/4} / \log(\sigma_t)$ , then we use Theorem 4.6 to finish the computation in  $\mathcal{O}(n \lg \sigma_t) = \mathcal{O}(n \lg \sigma)$  time. Until then (or until we finished all  $\lceil \log \log n \rceil$  phases), we use  $\mathcal{O}(\sum_{t=0}^{t'} n \log(\sigma_t) / (\sigma_t)^{1/4})$  time, and by Corollary 4.21 this is  $\mathcal{O}(n)$ . For detecting squares, we still use Lemma 4.18, which as explained in Section 4 takes  $\mathcal{O}(n)$  time and comparisons in total, plus additional  $\mathcal{O}(Z)$  time, where  $Z$  is the number of approximate LZ factors considered during all invocations of the lemma. We apply the lemma to each approximate LZ factorisation exactly once, and by construction each factor in phase  $t$  has size at least  $\Delta = \Omega(\sigma_t)$ . Also, each text position is covered by at most two tails per phase. Hence  $Z = \mathcal{O}(\sum_{t=0}^{t'} n / \sigma_t)$ , which is  $\mathcal{O}(n)$  by Corollary 4.21.

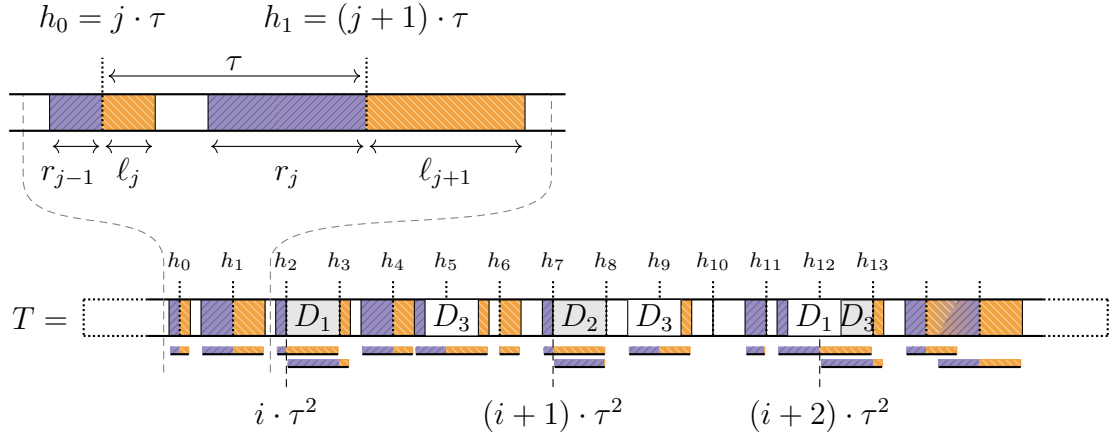
The last thing that remains to be shown is how to implement the bookkeeping of blocks, i.e., in each phase we have to efficiently deactivate block pairs as described at the end of Section 4. We maintain the block pairs in  $\lceil \log \log n \rceil$  bitvectors of total length  $\mathcal{O}(n)$ , where a set bit means that a block pair has been deactivated (recall that there are  $\mathcal{O}(n)$  pairs in total). Bitvector  $t$  contains at position  $j$  the bit corresponding to block pair  $B_j B_{j+1} = T[i..i + 2(\sigma_t)^2]$  with  $i = (\sigma_t)^2 \cdot (j - 1)$ . Note that translating between  $i$  and  $j$  takes constant time. For each sufficiently long tail in phase  $t$ , we simply iterate over the relevant block pairs in phase  $t + 2$  and deactivate them, i.e., we set the corresponding bit. This takes time linear in the number of deactivated blocks. Since there are  $\mathcal{O}(n)$  block pairs, and each block pair gets deactivated at most a constant number of times, the total cost for this bookkeeping is  $\mathcal{O}(n)$ .

The number of comparisons is dominated by the  $\mathcal{O}(n \lg \sigma)$  comparisons used when finishing the computation with Theorem 4.6. The only other comparisons are performed by Lemma 4.18, which we already bounded by  $\mathcal{O}(n)$ , and by LCE queries via Corollary 4.23. Since we ask  $\mathcal{O}(n)$  such queries in total, the number of comparisons is also  $\mathcal{O}(n)$ . We have shown:

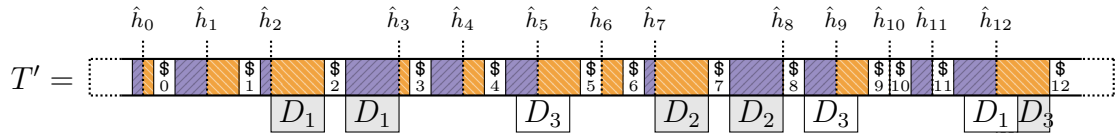
**Lemma 4.28.** *The square detection algorithm from Section 4 can be implemented in  $\mathcal{O}(n \lg \sigma + n \log^* n)$  time and  $\mathcal{O}(n \lg \sigma)$  comparisons.*

## 5.2 Final Improvement

For our final improvement we need to replace the LCE queries implemented by Corollary 4.23 with our own mechanism. The goal will remain the same, that is, given a parameter  $\Delta$  and estimate  $\tilde{\sigma}$  of the alphabet size, find a  $\Delta$ -approximate LZ factorisation of any fragment  $T[x..y]$  in  $\mathcal{O}(m\tilde{\sigma} \log m / \sqrt{\Delta})$  time, where  $m = |T[x..y]|$  (with  $m = \Theta(\Delta^2)$ , as otherwise we are not required to detect anything).



(a) Sampling dense fragments and cutting the text into chunks. Dotted lines indicate chunk boundaries, and  $h_x = (j+x) \cdot \tau$  for some integer  $j$  and  $x \in [0, 13]$  are positions of chunk boundaries. The dense fragments are  $D_1 = T[h_2..h_3]$ ,  $D_2 = T[h_7..h_8]$ , and  $D_3 = T[h_{12}..h_{13}]$ . The primary occurrences of dense fragments are grey, while the secondary occurrences (the ones that we aim to find) are white. A purple box in the text, and the matching purple line underneath the text, correspond to some substring  $T[j \cdot \tau - r_{j-1}..j \cdot \tau]$ . Similarly, the orange boxes and lines correspond to substrings  $T[j \cdot \tau..j \cdot \tau + \ell_j]$ .



(b) The string  $T'$  used to find all the occurrences of dense fragments. Each position  $\hat{h}_x$  maps to position  $h_x$  in Fig. 4.4a. The substring indicated by the purple box preceding  $h_x = (j+x)\tau$  and the orange box succeeding  $h_x$  is exactly  $T[h_x - r_{j+x-1}..h_x + \ell_{j+x}]$ . Each  $\$$  is a distinct separator symbol that is unique within  $T'$ .

Figure 4.4: Supplementary drawings for Section 5.2.

As in the previous section, the algorithm might detect that the size of the alphabet is larger than  $\tilde{\sigma}$ , and in such case we revert to the divide-and-conquer algorithm. Let  $\tau = \lfloor \sqrt{\Delta} \rfloor$ .

Initially, we only consider some fragments of  $T[x..y]$ . We say that  $T[i \cdot \tau^2..i \cdot \tau^2 + \tau)$  is a dense fragment. We start by remapping the characters in all dense fragments that intersect  $T[x..y]$  to a linearly-sortable alphabet. This can be done in  $\mathcal{O}(\tilde{\sigma})$  time for each position by maintaining a list of the already seen distinct characters. For each position in a dense fragment, we iterate over the characters in the list, and possibly append a new character to the list if it is not present. As soon as the size of the list exceeds  $\tilde{\sigma}$ , we terminate the procedure and revert to the divide-and-conquer algorithm. Otherwise, we replace each character by its position in the list. Overall, there are  $\mathcal{O}(m/\sqrt{\Delta})$  positions in the dense fragments of  $T[x..y]$ , and the remapping takes  $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$  time.

Next, we construct two generalised suffix trees [57], the first one of all dense fragments, and the second one of their reversals. (The generalised suffix tree of a collection of strings is the compacted trie that contains all suffixes of all strings in the collection.) Again, because we now work with a linearly-sortable alphabet this takes only  $\mathcal{O}(m/\sqrt{\Delta})$  time [56]. We consider fragments of the form  $T[i \cdot \tau..(i+1) \cdot \tau)$  having non-empty intersection with  $T[x..y]$ . We call such fragments chunks. We note that there are  $\mathcal{O}(m/\sqrt{\Delta})$  chunks, and their total length is  $\mathcal{O}(m)$ . For each chunk, we find its longest prefix  $T[i \cdot \tau..i \cdot \tau + \ell_i)$  and longest suffix  $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$  that occur in one of the dense fragments. Fig. 4.4a visualizes the dense fragments, chunks, and longest prefixes and suffixes. This can be done efficiently by following the heavy path decomposition of the generalised suffix tree of all dense fragments and their reversals, respectively. On each current heavy path, we just naively match the characters as long as possible. In case of a mismatch, we spend  $\mathcal{O}(\tilde{\sigma})$  time to descend to the appropriate subtree, which happens at most  $\mathcal{O}(\log m)$  times due to the heavy path decomposition. After having found  $\ell_i$  and  $r_i$ , we test square-freeness of  $T[i \cdot \tau..i \cdot \tau + \ell_i)$  and  $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$ . Because they both occur in dense fragments, and we have remapped the alphabet of all dense fragments, we can use Theorem 4.8 to implement this in  $\mathcal{O}(\ell_i + r_i)$  time. Thus, the total time per chunk is thus  $\mathcal{O}(\tilde{\sigma} \log m)$  plus  $\mathcal{O}(\ell_i + r_i)$ . The former sums up to  $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$ , and we will later show that the latter can be amortised by deactivating blocks on the lower levels.

The situation so far is that we have remapped the alphabet of all dense fragments to linearly-sortable, and for every chunk we know its longest prefix and suffix that occur in one of the dense fragments. We concatenate all fragments of the form  $T[i \cdot \tau - r_{i-1}..i \cdot \tau + \ell_i)$  (intersected with  $T[x..y]$ ) while adding distinct separators in between to form a new string  $T'$ . We stress that, because we have remapped the alphabet of all dense fragments, and the found longest prefix and suffix of each chunk also occur in some dense fragment,  $T'$  is over linearly-sortable alphabet. Thus, we can build the suffix tree  $ST$  of  $T'$  in  $\mathcal{O}(|T'|)$  time [56]. A visualization of  $T'$  is provided in Fig. 4.4b

Let  $\mathcal{D} = \{D_1, D_2, \dots\}$  be the set of distinct dense fragments. We would like to construct the set of all occurrences of the strings in  $\mathcal{D}$  in  $T[x..y]$ . Using the suffix tree of  $T'$  we can retrieve all occurrences of every  $D_j$  in  $T'$ . We observe that, because of how we have defined  $T[i \cdot \tau..i \cdot \tau + \ell_i)$  and  $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau)$ , this will in fact give us all occurrences of every  $D_j$  in the original  $T[x..y]$ . To implement this efficiently, we proceed as follows. First, for every  $i$  we traverse  $ST$  starting from its root to find the (explicit or implicit) node corresponding to the dense fragment  $T[i \cdot \tau^2..i \cdot \tau^2 + \tau)$ . This takes only  $\mathcal{O}(m\tilde{\sigma}/\sqrt{\Delta})$  time. Then, all leaves in every subtree rooted at such a node correspond to occurrences of some  $D_j$ , and can be reported by traversing the subtree in time proportional to its size, so at most  $\mathcal{O}(|T'|)$  in total. Finally, remapping the occurrences back to  $T[x..y]$  can be done in constant time per occurrence by precomputing, for every position in  $T'$ , its corresponding position in  $T[x..y]$ , which can be done in  $\mathcal{O}(|T'|)$  time when constructing  $T'$ . Thus, in  $\mathcal{O}(|T'|)$  time, we obtain the set  $S$  of starting positions of all

occurrences of the strings in  $\mathcal{D}$ . We summarize the properties of  $S$  below.

**Proposition 4.29.**  *$S$  admits the following properties:*

1. For every  $i \in [x, y]$  such that  $i \equiv 0 \pmod{\tau^2}$ ,  $i \in S$ .
2. For every  $i \in [x, y - \tau]$ ,  $i \in S$  if and only if  $T[i..i + \tau) \in \mathcal{D}$ .
3.  $|S| \leq |T'|$ .

We now define a parsing of  $T[x..y]$  based on  $S$ . Let  $i_1 < i_2 < \dots < i_k$  be all the positions in  $S$ , that is,  $(i_j, i_{j+1}) \cap S = \emptyset$  for every  $j = 1, 2, \dots, k-1$ . For every  $j = 1, 2, \dots, k-1$ , we create the phrase  $T[i_j..i_{j+1} + \tau)$ . We add the last phrase  $T[i_k..y]$ . We stress that consecutive phrases overlap by  $\tau$  characters, and each phrase begins with a length- $\tau$  fragment starting at a position in  $S$ . This, together with property 2 of  $S$ , implies the following property.

**Observation 4.30.** *The set of distinct phrases is prefix-free.*

We would like to construct the compacted trie  $\mathcal{T}_{\text{phrase}}$  of all such phrases, so that (in particular) we identify identical phrases. We first notice that each phrase begins with a fragment  $T[i_j..i_j + \tau)$  that has its corresponding occurrence in  $T'$ . We note that, given a set of positions  $P$  in  $T$ , we can find their corresponding positions in  $T'$  (if they exist) by sorting and scanning in  $\mathcal{O}(|P| + |T'|)$  time.

Thus, we can assume that for each  $i_j$  we know its corresponding position  $i'_j$  in  $T'$ . Next, for each node of  $ST$  we precompute its unique ancestor at string depth  $\tau$  in  $\mathcal{O}(|T'|)$  time. Then, for every fragment  $T[i_j..i_j + \tau)$  we can access its corresponding (implicit or explicit) node of  $ST$ . This allows us to partition all phrases according to their prefixes of length  $\tau$ . In fact, this gives us the top part of  $\mathcal{T}_{\text{phrase}}$  containing all such prefixes in  $\mathcal{O}(m/\sqrt{\Delta})$  time, and for each phrase we can assume that we know the node of  $\mathcal{T}_{\text{phrase}}$  corresponding to its length- $\tau$  prefix.

To build the remaining part of  $\mathcal{T}_{\text{phrase}}$ , we partition the phrases into short and long.  $T[i_j..i_{j+1} + \tau)$  is short when  $i_{j+1} \leq i_j + \tau$  (meaning that its length is at most  $2\tau$ ), and long otherwise.

We begin with constructing the compacted trie  $\mathcal{T}'_{\text{phrase}}$  of all short phrases. This can be done similarly to constructing the top part of  $\mathcal{T}_{\text{phrase}}$ , except that now the fragments have possibly different lengths. However, every short phrase  $T[i_j..i_{j+1} + \tau)$  occurs in  $T'$  as  $T'[i'_j..i'_{j+1} + \tau)$ . We claim that the nodes of  $ST$  corresponding to every  $T'[i'_j..i'_{j+1} + \tau)$  can be found in  $\mathcal{O}(|T'|)$  time. This can be done by traversing  $ST$  in the depth-first order while maintaining a stack of all explicit nodes with string depth at least  $\tau$  on the current path. Then, when visiting the leaf corresponding to the suffix of  $T'$  starting at position  $i'_j$ , we iterate over the current stack to find the sought node. This takes at most  $\mathcal{O}(|T'[i_j + \tau..i_{j+1} + \tau]|)$  time, which sums up to  $\mathcal{O}(|T'|)$ . Having found the node of  $ST$  corresponding to  $T[i_j..i_{j+1} + \tau)$ , we extract  $\mathcal{T}'_{\text{phrase}}$  from  $ST$  in  $\mathcal{O}(|T'|)$  time.

With  $\mathcal{T}'_{\text{phrase}}$  in hand, we construct the whole  $\mathcal{T}_{\text{phrase}}$  as follows. We begin with taking the union of  $\mathcal{T}'_{\text{phrase}}$  and the already obtained top part of  $\mathcal{T}_{\text{phrase}}$ , this can be obtained in  $\mathcal{O}(|T'|)$  time. For each long phrase  $T[i_j..i_{j+1} + \tau)$ , we know the node corresponding to  $T[i_j..i_j + \tau)$  and would like to insert the whole string  $T[i_j..i_{j+1} + \tau)$  into  $\mathcal{T}_{\text{phrase}}$ . We perform the insertions in increasing order of  $i_j$  (this will be crucial for amortising the time later). This is implemented with a dynamic heavy path decomposition similarly as in Section 5.1, however with one important change. Namely, we fix a heavy path decomposition of the part of  $\mathcal{T}_{\text{phrase}}$  corresponding to the union of  $\mathcal{T}'_{\text{phrase}}$  and the top part of  $\mathcal{T}_{\text{phrase}}$ , and maintain a dynamic heavy path decomposition of every subtree hanging off from this part. Thanks to this change, the time to maintain the dynamic trie and all heavy path decompositions is  $\mathcal{O}(m \log m / \sqrt{\Delta})$ , as there are only  $\mathcal{O}(m/\sqrt{\Delta})$

long phrases. Next, for each long phrase  $T[i_j..i_{j+1} + \tau)$ , we begin the insertion at the already known node corresponding to  $T[i_j..i_j + \tau)$ , and continue the insertion by following the heavy paths, first in the static heavy path decomposition in the part of  $\mathcal{T}_{\text{phrase}}$  corresponding to  $\mathcal{T}'_{\text{phrase}}$ , second in the dynamic heavy path decomposition in the appropriate subtree. On each heavy path, we naively match the characters as long as possible. The time to insert a single phrase  $T[i_j..i_{j+1} + \tau)$  is  $\mathcal{O}(\log m)$  (twice) plus the length of the longest prefix of  $T[i_j + \tau..i_{j+1} + \tau)$  equal to a prefix of  $T[i_{j'} + \tau..i_{j'+1} + \tau)$ , for some  $j' < j$ . The former sums up to another  $\mathcal{O}(m \log m / \sqrt{\Delta})$ , and we will later show that the latter can be amortised by deactivating blocks on the lower levels.

$\mathcal{T}_{\text{phrase}}$  allows us to form metacharacters corresponding to the phrases, and transform  $T[x..y]$  into a string  $T_{\text{parse}}$  of length  $\mathcal{O}(|T'|)$  consisting of these metacharacters. We build a suffix tree  $\mathcal{S}_{\text{parse}}$  over this string over linearly-sortable metacharacters in  $\mathcal{O}(|T'|)$  time. Next, we convert it into the sparse suffix tree  $\mathcal{S}'_{\text{parse}}$  of all suffixes  $T[i_j..y]$  as follows. Consider an explicit node  $u \in \mathcal{S}_{\text{parse}}$  with children  $v_1, v_2, \dots, v_d$ ,  $d \geq 2$ . We first compute the subtree  $\mathcal{T}_u$  of  $\mathcal{T}_{\text{phrase}}$  induced by the leaves corresponding to the first metacharacters on the edges  $(u, v_i)$ , for  $i = 1, 2, \dots, d$ , and connect every  $v_i$  to the appropriate leaf of  $\mathcal{T}_u$ . This can be implemented in  $\mathcal{O}(d)$  time, assuming constant-time lowest common ancestor queries on  $\mathcal{T}_{\text{phrase}}$  [67] and processing the leaves from left to right with a stack, similarly as in the Cartesian tree construction algorithm [21]. We note that the order on the leaves is the same as the order on the metacharacters, and hence no extra sorting is necessary. Overall, this sums up to  $\mathcal{O}(|T'|)$ . Next, we observe that, unless  $u$  is the root of  $\mathcal{S}_{\text{parse}}$ , all metacharacters on the edges  $(u, v_i)$  correspond to strings starting with the same prefix of length  $\tau$ . We obtain the subtree  $\mathcal{T}'_u$  by truncating this prefix (or taking  $\mathcal{T}_u$  if  $u$  is the root). Finally, we identify the root of  $\mathcal{T}'_u$  with  $u$ , and every child  $v_i$  with its corresponding leaf of  $\mathcal{T}'_u$ . Because we truncate the overlapping prefixes of length  $\tau$ , after this procedure is executed on every node of  $\mathcal{S}_{\text{parse}}$  we obtain a tree  $\mathcal{S}'_{\text{parse}}$  with the property that each leaf corresponds to a suffix  $T[i_j..y]$ . Also, by Observation 4.30, the edges outgoing from every node start with different characters as required.

By following an argument from the proof of Lemma 4.26,  $\mathcal{S}'_{\text{parse}}$  allows us to determine, for every suffix  $T[i_j..y]$ , its longest prefix equal to a prefix of some  $T[i'..y]$  with  $i' < i_j$ , as long as its length is at least  $\tau$ . Indeed, in such case we must have  $i' \in S$  by property 2, so in fact  $i' = i_{j'}$  and it is enough to maximise the length of the common prefix with all earlier positions in  $S$ , which can be done using  $\mathcal{S}'_{\text{parse}}$ . Thus, we either know that the length of this longest prefix is less than  $\tau$ , or know its exact value (and the corresponding position  $i' \in S$ ).

**Lemma 4.31.** *For any parameter  $\Delta \in [1, m]$  and estimate  $\tilde{\sigma}$  of the alphabet size, a  $(\Delta + \tau)$ -approximate LZ factorisation of any fragment  $T[x..y]$  can be computed in  $\mathcal{O}(m/\sqrt{\Delta})$  time with  $m = |T[x..y]|$  (assuming the preprocessing described earlier in this section).*

*Proof.* Let  $e \in [x, y]$  and suppose we have already constructed the factorisation of  $T[x..e - 1]$  and are now trying to construct the next phrase. Let  $e'$  be the next multiple of  $\tau^2$ , we have that  $e' - e < \tau^2 \leq \Delta$  and  $T[e'..e' + \tau)$  is a dense fragment. Thus, by property 1 we have  $e' \in S$ .

The first possibility is that the longest common prefix between  $T[e'..y]$  and any suffix starting at an earlier position is shorter than  $\tau$ . In this case, we can simply set the head of the new phrase to be  $T[e'..e' + \tau)$  and the tail to be empty. Otherwise, we know the length  $\ell$  of this longest prefix by the preprocessing described above. We set the head of the new phrase to be  $T[e'..e')$  and the tail to be  $T[e'..e' + \ell)$ . This takes constant time per phrase, and each phrase is of length at least  $\tau$ , giving the claimed overall time complexity. It remains to argue correctness of every step.

Let  $T[e..s]$  be the longest LZ phrase starting at position  $e$ , to show that we obtain a valid  $(\Delta + \tau)$ -approximate phrase it suffices to show that  $s \leq e' + \max(\tau, \ell)$ . Let the previous occurrence

of  $T[e..s]$  be at position  $p < e$ . If  $s - e' < \tau$  then there is nothing to prove. Otherwise,  $T[e'..s]$  is a string of length at least  $\tau$  that also occurs starting earlier at position  $p + e' - e < e'$ . Thus, we will correctly determine that  $\ell \geq \tau$ , and find a previous occurrence of the string maximising the value of  $\ell$ . In particular, we will have  $\ell \geq s - e'$  as required.  $\square$

To achieve the bound of Theorem 4.3, we now proceed as in Section 4.4, except that instead of Lemma 4.27 we use Lemma 4.31. For every  $T[x..y]$  with  $m = |T[x..y]|$  this takes  $\mathcal{O}(m\tilde{\sigma} \log m/\sqrt{\Delta})$  time plus the time used for computing the longest prefix and suffix of each chunk (the latter also accounts for constructing the suffix tree  $ST$  and other steps that have been estimated as taking  $\mathcal{O}(|T'|)$  in the above reasoning) plus the time for inserting  $T[i_j + \tau..i_{j+1} + \tau]$  into  $\mathcal{T}_{\text{phrase}}$  when  $i_{j+1} \geq i_j + \tau$ .

We observe that we can deactivate any block pair fully contained in  $T[i \cdot \tau..i \cdot \tau + \ell_i]$  and  $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau]$ , as we have already checked that these fragments are square-free. Also, we can deactivate any block pair fully contained in the longest prefix of  $T[i_j + \tau..i_{j+1} + \tau]$  equal to  $T[i_{j'} + \tau..i_{j'+1} + \tau]$ , for some  $j' < j$ , because such fragment cannot contain the leftmost occurrence of a square.

There are  $\mathcal{O}(m/\sqrt{\Delta})$  chunks and long phrases. If a chunk or a long phrase contributes  $x = \Omega(\sqrt[4]{\Delta})$  to the total time, then we explicitly deactivate the block pairs in phase  $t+3$  that are entirely contained in the corresponding fragment. Block pairs in phase  $t+3$  are of length  $\mathcal{O}(\sqrt[4]{\Delta})$ , and thus we deactivate  $\Omega(x)$  positions. Therefore, the time spent on such chunks and long phrases in all phases sums to  $\mathcal{O}(n)$ . The remaining chunks and long phrases contribute  $\mathcal{O}(\sqrt[4]{\Delta})$  to the total time, and there are  $\mathcal{O}(m/\sqrt{\Delta})$  of them, which adds up to  $\mathcal{O}(m/\sqrt[4]{\Delta})$ . In every phase, this is  $\mathcal{O}(n/\sqrt[4]{\Delta})$ , so  $\mathcal{O}(n)$  overall by Corollary 4.21.

## 6 Computing Runs

Now we adapt the algorithm such that it computes all runs. We start with the algorithm from Sections 4 and 5 without the final improvement from Section 5.2. First, note that the key properties of the  $\Delta$ -approximate LZ factorisation, in particular Lemmas 4.17 and 4.18, also hold for the computation of runs. This is expressed by the lemmas below.

**Lemma 4.32.** *Let  $b_1b_2 \dots b_z$  be a  $\Delta$ -approximate LZ factorisation of a string  $T$ . For every run  $\langle s, e, p \rangle$  of length  $e - s + 1 \geq 8\Delta$ , there is at least one phrase  $b_i$  with  $|\text{tail}(b_i)| \geq \frac{e-s+1}{8} \geq \Delta$  such that  $\text{tail}(b_i)$  and the right-hand side  $T[s + \lceil \frac{e-s+1}{2} \rceil .. e]$  of the run intersect.*

*Proof.* Let  $\ell = \frac{e-s+1}{2}$  and note that  $\frac{\ell}{4} \geq \Delta$  and  $e = s + 2\ell - 1$ . Assume that all tails that intersect  $T[s + \lceil \ell \rceil .. e]$  are of length less than  $\frac{\ell}{4}$ , then the respective phrases of these tails are of length at most  $\frac{\ell}{4} + \Delta - 1 \leq \frac{\ell}{2} - 1$  (because each head is of length less than  $\Delta$ ). This means that  $T[s + \lceil \ell \rceil .. e]$  (of length  $\lceil \ell \rceil$ ) intersects at least  $\lceil \lceil \ell \rceil / (\frac{\ell}{2} - 1) \rceil \geq 3$  phrases (the inequality holds for  $\ell \geq 4$ , which is implied by  $\Delta \geq 1$ ). Thus there is some phrase  $b_i = T[x..y]$  properly contained in  $T[s + \lceil \ell \rceil .. e]$ , formally  $s + \lceil \ell \rceil < x \leq y < e$ . However, this contradicts the definition of the  $\Delta$ -approximate LZ factorisation because  $T[x..e+1]$  is the prefix of a standard LZ phrase (due to  $T[x..e] = T[x-p..e-p]$ ). The contradiction implies that  $T[s + \lceil \ell \rceil .. e]$  intersects a tail of length at least  $\frac{\ell}{4}$ .  $\square$

Before we show how to algorithmically apply Lemma 4.32, we need to explain how Lemma 4.7 extends to computing runs, and then how this implies that the approach of Main and Lorentz [27] easily extends to computing all runs. We do not claim this to be a new result, but the original paper only talks about finding a representation of all squares, and we need to find runs, and hence include a description for completeness.

**Lemma 4.33.** *Given two strings  $x$  and  $y$  over a general alphabet, we can compute all runs in  $xy$  that include either the last character of  $x$  or the first character of  $y$  using  $\mathcal{O}(|x| + |y|)$  time and comparisons.*

*Proof.* Consider a run  $\langle s, e, p \rangle$  in  $t = xy$  that includes either the last character of  $x$  or the first character of  $y$ , meaning that  $s \leq |x| + 1$  and  $e \geq |x|$ . Let  $\ell = \lfloor \frac{e-s+1}{2} \rfloor \geq p$ . We separately compute all runs with  $s + \ell \leq |x| + 1$  and  $s + \ell > |x| + 1$ . Below we describe the former, and the latter is symmetric.

Due to  $s + \ell \leq |x| + 1$ , the length- $p$  substring  $x[|x| - p + 1..|x|]$  is fully within the run. This suggests the following strategy to generate all runs with  $s + \ell \leq |x| + 1$ . We iterate over the possible values of  $p = 1, 2, \dots, |x|$ . For a given  $p$ , we calculate the length of the longest common prefix of  $x[|x| - p + 1..|x|]y$  and  $y$ , denoted **pref**, and the length of the longest common suffix of  $x[1..|x| - p]$  and  $x$ , denoted **suf**. It is easy to see that  $t[|x| - p + 1 - \text{suf}..|x| + \text{pref}]$  is a lengthwise maximal  $p$ -periodic substring, and its length is  $\ell' = p + \text{suf} + \text{pref}$ . If  $\text{pref} + \text{suf} \geq p$  and  $s + \lfloor \ell'/2 \rfloor \leq |x| + 1$ , then we report the substring as a run. (The latter condition ensures that each run gets reported by exactly one of the two symmetric cases.)

We use a prefix table to compute the longest common prefixes. For a given string, this table contains at position  $i$  the length of the longest substring starting at position  $i$  that is also a prefix of the string. For computing the values **pref**, we use the prefix table of  $y\$xy$  (where  $\$$  is a new character that does not match any character in  $x$  nor  $y$ ). Similarly, for computing the values **suf**, we use the prefix table of the reversal of a new string  $x\$x$ . The tables can be computed in  $\mathcal{O}(|x| + |y|)$  time and comparisons (see, e.g., computation of table *lppattern* in [27]). Then, each value of  $p$  can be checked in constant time.  $\square$

**Lemma 4.34.** *Computing all runs in a length- $n$  string over a general unordered alphabet can be implemented in  $\mathcal{O}(n \log n)$  time and comparisons.*

*Proof.* Let the input string be  $T[1..n]$ . We apply divide-and-conquer. Let  $x = T[1..\lfloor n/2 \rfloor]$  and  $y = T[\lfloor n/2 \rfloor + 1..n]$ . First, we recursively compute all runs in  $x$  and  $y$ . Of the reported runs, we filter out all the ones that contain either the last character of  $x$  or the first character of  $y$ , which takes  $\mathcal{O}(|x| + |y|)$  time. In this way, if some reported run is a run with respect to  $x$  (or  $y$ ), but not with respect to  $xy$ , then it will be filtered out. We have generated all runs except for the ones that contain the last character of  $x$  or the first character of  $y$  (or both). Thus we simply invoke Lemma 4.33 on  $xy$ , which will output exactly the missing runs in  $\mathcal{O}(|x| + |y|)$  time and comparisons. There are  $\mathcal{O}(\log n)$  levels of recursion, and each level takes  $\mathcal{O}(n)$  time and comparisons in total.  $\square$

**Lemma 4.35.** *Let  $T = b_1b_2\dots b_z$  be a  $\Delta$ -approximate LZ factorisation of  $T$ , and  $\chi = \sum_{|\text{tail}(b_i)| \geq \Delta} |\text{tail}(b_i)|$ . We can compute in  $\mathcal{O}(\chi + z)$  time and  $\mathcal{O}(\chi)$  comparisons a multiset  $R$  of size  $\mathcal{O}(\chi)$  of runs with the property that a run  $T[s..e]$  is possibly not in  $R$  only if  $e - s + 1 < 8\Delta$  or there is some tail  $\text{tail}(b_i) = T[a_2..a_3]$  with  $a_2 < s$  and  $e < a_3$ .*

*Proof.* Let  $n = |T|$ . We consider each phrase  $b_i = T[a_1..a_3]$  with  $\text{head}(b_i) = T[a_1..a_2 - 1]$  and  $\text{tail}(b_i) = T[a_2..a_3]$  separately. Let  $k = |\text{tail}(b_i)|$ . If  $k \geq \Delta$ , we apply Lemma 4.33 to  $x_1 = T[a_2 - 8k..a_2 - 1]$  and  $y_1 = T[a_2..a_3 + 4k]$ , as well as  $x_2 = T[a_2 - 8k..a_3 - 1]$  and  $y_2 = T[a_3..a_3 + 4k]$  trimmed to  $T[1..n]$ . This takes  $\mathcal{O}(|\text{tail}(b_i)|)$  time and comparisons and reports  $\mathcal{O}(|\text{tail}(b_i)|)$  runs with respect to  $x_1y_1 = x_2y_2 = T[a_2 - 8k..a_3 + 4k]$  (trimmed to  $T[1..n]$ ). Of these runs, we filter out the ones that contain any of the positions  $a_2 - 8k$  (only if  $a_2 - 8k > 1$ ) and  $a_3 + 4k$  (only if  $a_3 + 4k < n$ ), which takes  $\mathcal{O}(|\text{tail}(b_i)|)$  time. This way, each reported run is not only a run with respect to  $x_1y_1$ , but also a run with respect to  $T$ . In total, we report  $\mathcal{O}(\chi)$  runs (including possible duplicates) and spend  $\mathcal{O}(\chi)$  time and comparisons when applying Lemma 4.33. Additional  $\mathcal{O}(z)$  time is needed to check if  $|\text{tail}(b_i)| \geq \Delta$  for each phrase.

Now we show that the described strategy computes all runs of length at least  $8\Delta$ , except for the ones that are properly contained in a tail. Let  $\langle s, e, p \rangle$  be a run of length  $2\ell$ , where  $\ell \geq 4\Delta$  is a multiple of  $\frac{1}{2}$ . Due to Lemma 4.32, the right-hand side  $T[s + \lceil \ell \rceil .. e]$  of this run intersects some tail  $\text{tail}(b_i) = T[a_2..a_3]$  of length  $k = |\text{tail}(b_i)| \geq \frac{\ell}{4} \geq \Delta$ . Due to the intersection, we have  $a_2 \leq e$  and  $a_3 \geq s + \lceil \ell \rceil$ . Thus, when processing  $b_i$  and applying Lemma 4.33, the starting position of  $x_1$  and  $x_2$  satisfies  $a_2 - 8k \leq e - 8\frac{\ell}{4} < s$ , while the end position of  $y_1$  and  $y_2$  satisfies  $a_3 + 4k \geq s + \lceil \ell \rceil + 4\frac{\ell}{4} > e$ . Therefore, the run is contained in the fragment  $T[a_2 - 8k..a_3 + 4k]$  (trimmed to  $T[1..n]$ ) corresponding to  $x_1y_1$  and  $x_2y_2$ , and the run does not contain positions  $a_2 - 8k$  and  $a_3 + 4k$ . If  $s \leq a_2 \leq e$ , we find the run when applying Lemma 4.33 to  $x_1$  and  $y_1$ . If  $s \leq a_3 \leq e$ , we find the run when applying Lemma 4.33 to  $x_2$  and  $y_2$ . Otherwise,  $T[s..e]$  is entirely contained in  $T[a_2 + 1..a_3 - 1]$  and we do not have to report the run.  $\square$

Now we describe how to compute all runs using  $\mathcal{O}(n \log \sigma)$  comparisons and  $\mathcal{O}(n \log \sigma + n \log^* n)$  time. We again use the sequence  $\sigma_t = 2^{\lceil \log \log n \rceil - t}$ , for  $t = 0, 1, \dots, \lceil \log \log n \rceil$ . We observe that  $\sigma_{t-1} = (\sigma_t)^2$ , and proceed in phases corresponding to the values of  $t$ . In the  $t^{\text{th}}$  phase we aim to compute runs of length at least  $\sigma_t$  and less than  $(\sigma_t)^2$ . We stress that this condition depends on the length of the run and not on its period. We partition the whole  $T[1..n]$  into blocks of length  $(\sigma_t)^2$ , and denote the  $k^{\text{th}}$  block by  $B_k$ . A run of length less than  $(\sigma_t)^2$  is fully contained within some two consecutive blocks  $B_i B_{i+1}$ , and there is always a pair of consecutive blocks such that the run contains neither the first nor the last position of the pair (unless the first position is  $T[1]$  or the last position is  $T[n]$  respectively). Hence we consider each pair  $B_1 B_2$ ,  $B_2 B_3$ , and so on. We first apply Lemma 4.27 with  $\Delta = \sigma_t/8$  and  $\tilde{\sigma} = (\sigma_t)^{1/4}/\log(\sigma_t)$  to find an  $(\sigma_t/8)$ -approximate LZ factorisation of the corresponding fragment of  $T[1..n]$ , and then use Lemma 4.35 to compute all runs of length at least  $\sigma_t$ , apart from possibly the ones that are properly contained in a tail. Of the computed runs, we discard the ones that contain the first or last position of the block pair (unless the first position is  $T[1]$  or the last position is  $T[n]$  respectively). This way, each reported run is a run not only with respect to the block pair, but with respect to the entire  $T[1..n]$ . If we do not report some run of length at least  $\sigma_t$  and less than  $(\sigma_t)^2$  in this way, then it is properly contained in one of the tails.

We cannot always afford to apply Lemmas 4.27 and 4.35 to all block pairs. Thus, we have to deactivate some of the blocks. During the current phase  $t$ , for each tail  $T[s..e]$  of length at least  $\Delta$ , we deactivate all block pairs in phase  $t + 3$  that are contained in  $T[s + 1..e - 1]$ . By similar logic as in Section 4, if a tail contributes  $e - s + 1$  comparisons and time to the application of Lemma 4.35, then it permanently deactivates  $\Omega(e - s + 1)$  positions of the string, and thus the total time and comparisons needed for all invocations of Lemmas 4.27 and 4.35 are bounded by  $\mathcal{O}(n)$  (apart from the additional  $\mathcal{O}(n \log^* n)$  total time for Lemma 4.27). Whenever we apply Lemma 4.27, we add all the tails of length at least  $\Delta$  to a list  $\mathcal{L}$ , where each tail is annotated with the position of its previous occurrence. After the algorithm terminates,  $\mathcal{L}$  contains all sufficiently long tails from all phases. We have already shown that the total time needed for Lemma 4.35 is bounded by  $\mathcal{O}(n)$ , and thus the total length of the tails in  $\mathcal{L}$  is at most  $\mathcal{O}(n)$ .

If any of the calls to Lemma 4.27 in the current phase detects that  $\sigma > \tilde{\sigma}$ , or if  $\tilde{\sigma} < 256$ , we immediately switch to applying Lemma 4.34 on every pair of blocks  $B_i B_{i+1}$  of the current phase, which takes  $\mathcal{O}(n \log \sigma)$  time (because the length of a block pair is polynomial in  $\tilde{\sigma}$ ). Again, after applying Lemma 4.27 to  $B_i B_{i+1}$ , we discard all runs that contain the first or last position of  $B_i B_{i+1}$  (unless the first position is  $T[1]$  or the last position is  $T[n]$ , respectively). After this procedure terminates, we have computed all runs, except for possibly some of the runs that were properly contained in a tail in list  $\mathcal{L}$ . We may have reported some duplicate runs, which we filter out as follows. The number of runs reported so far is  $r = \mathcal{O}(n \log \sigma)^4$ . We sort them in

<sup>4</sup>a more careful analysis would reveal that it is  $\mathcal{O}(n)$ , but this is not necessary for the proof



additional  $\mathcal{O}(n + r) = \mathcal{O}(n \log \sigma)$  time, e.g., by using radix sort, and remove duplicates. The running time so far is  $\mathcal{O}(n \log \sigma)$ .

## 6.1 Copying Runs From Previous Occurrences

Lastly, we have to compute the runs that were properly contained in a tail in  $\mathcal{L}$ . Consider such a run  $\langle r_s, r_e, p \rangle$ , and let  $T[s..e]$  be a tail in  $\mathcal{L}$  with  $s < r_s$  and  $r_e < e$ . If multiple tails match this criterion, let  $T[s..e]$  be the one that maximizes  $e$ . In  $\mathcal{L}$ , we annotated  $T[s..e]$  with its previous occurrence  $T[s - d..e - d]$ . Note that  $\langle r_s - d, r_e - d, p \rangle$  is also a run. Thus, if we compute the runs in an appropriate order, we can simply copy the missing runs from their respective previous occurrences. For this sake, we annotate each position  $i \in [1, n]$  with:

- a list of all the runs  $\langle i, e, p \rangle$  that we already computed, arranged in increasing order of end position  $e$ . We already sorted the runs for duplicate elimination, and can annotate all position in  $\mathcal{O}(n)$  time.
- a pair  $(e^*, d^*)$ , where  $e^* = d^* = 0$  if there is no tail  $T[s..e]$  such that  $s < i < e$ . Otherwise, among all tails  $T[s..e]$  with  $s < i < e$ , we choose the one that maximizes  $e$ . Let  $T[s - d..e - d]$  be its previous occurrence, then we use  $e^* = e$  and  $d^* = d$ . As explained earlier, the total length of all tails in  $\mathcal{L}$  is  $\mathcal{O}(n)$ , and thus we can simply scan each tail and update the annotation pair of each contained position whenever necessary.

Observe that, if a position is annotated with  $(0, 0)$ , then none of the runs starting at position  $i$  is fully contained in a tail, and thus we have already annotated position  $i$  with the complete list of the runs starting at  $i$ . Now we process the positions  $i \in [1, n]$  one at a time and in increasing order. We inductively assume that, at the time at which we process  $i$ , we have already annotated each  $j < i$  with the complete list of runs starting at  $j$ . Hence our goal is to complete the list of  $i$  such that it contains all runs starting at  $i$ . If  $i$  is annotated with  $(0, 0)$ , then the list is already complete. Otherwise,  $i$  is annotated with  $(e, d)$ , every missing run  $\langle i, e_r, p \rangle$  satisfies  $e_r < e$ , and the annotation list of  $i - d$  already contains the run  $\langle i - d, e_r - d, p \rangle$  (due to  $T[i - 1..e_r + 1] = T[i - d - 1..e_r - d + 1]$  and the inductive assumption). For each run  $\langle i - d, e_r - d, p \rangle$  in the annotation list of position  $i - d$ , we insert the run  $\langle i, e_r, p \rangle$  into the annotation list of  $i$ . We perform this step in a merging fashion, starting with the shortest runs of both lists and zipping them together. As soon as we are about to insert a run  $\langle i, e_r, p \rangle$  with  $e_r \geq e$ , we do not insert it and abort. Thus, the time needed for processing  $i$  is linear in the number of runs starting at position  $i$ . By the runs theorem [236], the total number of runs is less than  $n$ , making the total time for this step  $\mathcal{O}(n)$ .

Apart from the new steps in Section 6.1, the complexity analysis works exactly like in Section 4. Hence we have shown:

**Theorem 4.36.** *Computing all runs in a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be implemented in  $\mathcal{O}(n \log \sigma)$  comparisons and  $\mathcal{O}(n \log \sigma + n \log^* n)$  time.*

## 6.2 Final Improvement for Computing Runs

The goal is now to adapt the final algorithm to detect all runs. We can no longer stop as soon as we detect a square, and we cannot simply deactivate pairs of blocks that occur earlier. However, Theorem 4.8 is actually capable of reporting all runs in  $T[i \cdot \tau..i \cdot \tau + \ell_i]$  and  $T[(i+1) \cdot \tau - r_i..(i+1) \cdot \tau]$  in  $\mathcal{O}(\ell_i + r_i)$  time, and we do not need to terminate the algorithm if these fragments are not square-free. Thus, we can indeed deactivate any block pair fully contained in  $T[i \cdot \tau..i \cdot \tau + \ell_i]$

and  $T[(i + 1) \cdot \tau - r_i .. (i + 1) \cdot \tau)$ . Next, we also deactivate block pairs fully contained in the longest prefix of  $T[i_j + \tau .. i_{j+1} + \tau)$  equal to  $T[i_{j'} + \tau .. i_{j'+1} + \tau)$ , for some  $j' < j$ . Denoting the length of this prefix by  $\ell$ , we treat  $T[i_j + \tau .. i_j + \ell)$  as a tail and add it to the list  $\mathcal{L}$  (annotated with  $i_{j'}$ ). The total length of all fragments added to  $\mathcal{L}$  is still  $\mathcal{O}(n)$ .

**Theorem 4.37.** *Computing all runs in a length- $n$  string that contains  $\sigma$  distinct symbols from a general unordered alphabet can be implemented in  $\mathcal{O}(n \log \sigma)$  comparisons and  $\mathcal{O}(n \log \sigma)$  time.*



# Part II

## Approximate Matching for Bioinformatics

---



# Approximating Longest Common Substring with $k$ Mismatches

## Publication

This chapter corresponds to the extended version of the following publication: Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya, “Approximating Longest Common Substring with  $k$  mismatches: Theory and Practice”, in: *31<sup>st</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, June 17-19, 2020, Copenhagen, Denmark, ed. by Inge Li Gørtz and Oren Weimann, vol. 161, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 16:1–16:15, DOI: 10.4230/LIPIcs.CPM.2020.16.

In the problem of the longest common substring with  $k$  mismatches we are given two strings  $X, Y$  and must find the maximal length  $\ell$  such that there is a length- $\ell$  substring of  $X$  and a length- $\ell$  substring of  $Y$  that differ in at most  $k$  positions. The length  $\ell$  can be used as a robust measure of similarity between  $X, Y$ . In this work, we develop new approximation algorithms for computing  $\ell$  that are significantly more efficient than previously known solutions from the theoretical point of view. Our approach is simple and practical, which we confirm via an experimental evaluation, and is probably close to optimal as we demonstrate via a conditional lower bound.

## 1 Introduction

For decades, the edit distance and its variants remained the most relevant measure of similarity between biological sequences. However, there is strong evidence that the edit distance cannot be computed in strongly subquadratic time [254]. One possible approach to overcoming the quadratic time barrier is computing the edit distance approximately, and last year in the breakthrough paper Chakraborty et al. [255] showed a constant-factor approximation algorithm that computes the edit distance between two strings of length  $n$  in time  $\tilde{O}(n^{2-2/7})$ . Nevertheless, the algorithm is highly non-trivial and because of that is likely to be impractical.

A different approach is to consider alignment-free measures of similarities. Ideally, we want the measure to be robust and simple enough so that we could compute it efficiently. One candidate for such a measure is the length of the longest common substring with  $k$  mismatches. Formally, given two strings  $X, Y$  of lengths at most  $n$  and an integer  $k$ , we want to find the maximal length  $\text{LCS}_k(X, Y)$  of a substring of  $X$  that occurs in  $Y$  with at most  $k$  mismatches. Computing this value constitutes the **LCS with  $k$  Mismatches** problem.

The **LCS with  $k$  Mismatches** problem was first considered for  $k = 1$  [147, 204], with current best algorithm taking  $\mathcal{O}(n \log n)$  time and  $\mathcal{O}(n)$  space. The first algorithm for the general value of  $k$  was shown by Flouri et al. [204]. Their simple approach used quadratic time and linear space. Grabowski [206] focused on a data-dependent approach, namely, he showed two linear-space algorithms with running times  $\mathcal{O}(n((k+1)(\text{LCS}+1))^k)$  and  $\mathcal{O}(n^2k/\text{LCS}_k)$ , where  $\text{LCS}$  is the length of the longest common substring of  $X$  and  $Y$  and  $\text{LCS}_k$ , similarly to above, is the length

of the longest common substring with  $k$  mismatches of  $X$  and  $Y$ . Abboud et al. [192] showed a  $k^{1.5}n^2/2^{\Omega(\sqrt{(\log n)/k})}$ -time randomised solution to the problem via the polynomial method. Thankachan et al. [231] presented an  $\mathcal{O}(n \log^k n)$ -time,  $\mathcal{O}(n)$ -space solution for constant  $k$ . This approach was recently extended by Charalampopoulos et al. [257] to develop an  $\mathcal{O}(n)$ -time and  $\mathcal{O}(n)$ -space algorithm for the case of  $\text{LCS}_k = \Omega(\log^{2k+2} n)$ .

On the other hand, Kociumaka, Radoszewski, and Starikovskaya [294] showed that there is  $k = \Theta(\log n)$  such that the LCS with  $k$  Mismatches problem cannot be solved in strongly subquadratic time, even for the binary alphabet, unless the Strong Exponential Time Hypothesis (SETH) of Impagliazzo, Paturi, and Zane [71] is false. This conditional lower bound implies that there is little hope to improve existing solutions to LCS with  $k$  Mismatches. To overcome this barrier, they introduced an approximation approach to LCS with  $k$  Mismatches, inspired by the work of Andoni and Indyk [101].

**Problem 1** (LCS with Approximately  $k$  Mismatches). *Two strings  $X, Y$  of length at most  $n$ , an integer  $k$ , and a constant  $\varepsilon > 0$  are given. Return a substring of  $X$  of length at least  $\text{LCS}_k(X, Y)$  that occurs in  $Y$  with at most  $(1 + \varepsilon) \cdot k$  mismatches.*

Kociumaka, Radoszewski, and Starikovskaya [294] also showed that for any  $\varepsilon \in (0, 2)$  the LCS with Approximately  $k$  Mismatches problem can be solved in  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$  time and  $\mathcal{O}(n^{1+1/(1+\varepsilon)})$  space. Besides for superlinear space, their solution uses a very complex class of hash functions which requires  $n^{4/3+o(1)}$ -time preprocessing, and that is the underlying reason for the bounds on  $\varepsilon$ . In this work, we significantly improve the complexity of the LCS with Approximately  $k$  Mismatches problem and show the following results.

**Theorem 5.1.** *Let  $\varepsilon > 0$  be an arbitrary constant. The LCS with Approximately  $k$  Mismatches problem can be solved correctly with high probability:*

- 1) *In  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  time and  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  space assuming a constant-size alphabet;*
- 2) *In  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$  time and  $\mathcal{O}(n)$  space for alphabets of arbitrary size.*

Our first solution uses the Approximate Nearest Neighbour data structure [194] as a black box. The definition of this data structure is extremely involved, and we view this result as more of a theoretical interest. On the other hand, our second solution is simple and practical, which we confirm by experimental evaluation (see Section 4 for details).

As a final remark, we note that a construction similar to the one used to show a lower bound for the LCS with  $k$  Mismatches problem [294] gives a lower bound for LCS with Approximately  $k$  Mismatches.

**Fact 5.2.** *Assuming SETH, for every constant  $\delta > 0$ , there exists a constant  $\varepsilon = \varepsilon(\delta)^1$  such that any randomised algorithm that solves the LCS with Approximately  $k$  Mismatches problem for given  $X$  and  $Y$  of length at most  $n$  correctly with constant probability uses  $\Omega(n^{2-\delta})$  time.*

For completeness, we provide the proof of the fact in the full version of this paper.

**Related work.** In 2014, Leimster and Morgenstern [184] introduced a related similarity measure, the  $k$ -macs distance. Let  $\text{LCP}_k(X_i, Y_j) = \max\{\ell : d_H(X[i, i+\ell-1], Y[j, j+\ell-1]) \leq k\}$ , where  $d_H$  stands for Hamming distance, i.e. the number of mismatches between two strings. We have  $\text{LCS}_k = \max_{i,j} \text{LCP}_k(X_i, Y_j)$ . The  $k$ -macs distance, on the other hand, is defined as

<sup>1</sup>Here  $\delta$  is a function of  $\varepsilon$  for which the explicit form is not known (a condition inherited from [277]).

a normalised average of these values. Leimeister and Morgenstern [184] showed a heuristic algorithm for computing the  $k$ -macs distance, with no theoretical guarantees for the precision of the approximation; other heuristic approaches for computing the  $k$ -macs distance include [232, 250]. The only algorithm with provable theoretical guarantees is [231] and it computes the  $k$ -macs distance in  $\mathcal{O}(n \log^k n)$  time and  $\mathcal{O}(n)$  space.

## 2 Preliminaries

We assume that the alphabet of the strings  $X, Y$  is  $\Sigma = \{1, \dots, \sigma\}$ , where  $\sigma = n^{\mathcal{O}(1)}$ .

### Karp–Rabin fingerprints.

The Karp–Rabin fingerprint [35] of a string  $S = s_1 s_2 \dots s_\ell$  is defined as

$$\varphi(S) = \left( \sum_{i=1}^{\ell} r^{i-1} s_i \right) \bmod q,$$

where  $q = \Omega(\max\{n^5, \sigma\})$  is a prime number, and  $r \in \mathbb{F}_q$  is chosen uniformly at random. Obviously, if  $S_1 = S_2$ , then  $\varphi(S_1) = \varphi(S_2)$ . Furthermore, for any  $\ell \leq n$ , if the fingerprints of two  $\ell$ -length strings  $S_1, S_2$  are equal, then  $S_1, S_2$  are equal with probability at least  $1 - 1/n^4$  (for a proof, see e.g. [137]).

### Dimension reduction.

We will exploit a computationally efficient variant of the Johnson–Lindenstrauss lemma [26] which describes a low-distortion embedding from a high-dimensional Euclidean space into a low-dimensional one. Let  $\|\cdot\|$  be the Euclidean ( $L_2$ ) norm of a vector. We will exploit the following claim which follows immediately from [87, Theorem 1.1]:

**Lemma 5.3.** *Let  $P$  be a set of  $n$  vectors in  $\mathbb{R}^\ell$ , where  $\ell \leq n$ . Given  $\alpha = \alpha(n) > 0$  and a constant  $\beta > 0$ , there is  $d = \Theta(\alpha^{-2} \log n)$  and a scalar  $c > 0$  such that the following holds. Let  $M$  be a  $d \times \ell$  matrix filled with i.u.d.  $\pm 1$  random variables. For all  $U \in P$ , define  $\text{sk}_\alpha(U) = c \cdot MU$ . Then for all  $U, V \in P$  there is  $\|U - V\|^2 \leq \|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)\|U - V\|^2$  with probability at least  $1 - n^{-\beta}$ .*

Since the Hamming distance between binary strings  $U, V$  is equal to  $\|U - V\|^2$ , the matrix  $M$  defines a low-distortion embedding from an  $\ell$ -dimensional into a  $d$ -dimensional Hamming space as well. For non-binary strings, an extra step is required. Let the alphabet be  $\Sigma = \{1, 2, \dots, \sigma\}$  and consider a morphism  $\mu : \Sigma \rightarrow \{0, 1\}^\sigma$ , where  $\mu(a) = 0^{a-1} 1 0^{\sigma-a}$  for all  $a \in \Sigma$ . We extend  $\mu$  to strings in a natural way. Note that for two strings  $U, V$  over the alphabet  $\Sigma$  the Hamming distance between  $\mu(U), \mu(V)$  is exactly twice the Hamming distance between  $U, V$ . We therefore obtain:

**Corollary 5.4.** *Let  $P$  be a set of  $n$  strings in  $\Sigma^\ell$ , where  $\ell \leq n$ . Given  $\alpha = \alpha(n) > 0$  and a constant  $\beta > 0$ , there is  $d = \Theta(\alpha^{-2} \log n)$  and a scalar  $c > 0$  such that the following holds. Let  $M$  be a  $d \times (\sigma \cdot \ell)$  matrix filled with i.u.d.  $\pm 1$  random variables. For all  $U \in P$ , define  $\text{sk}_\alpha(U) = c \cdot M\mu(U)$ . Then for all  $U, V \in P$  there is  $d_H(U, V) \leq \|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)d_H(U, V)$  with probability at least  $1 - n^{-\beta}$ .*

We will use the corollary for dimension reduction, and also to design a simple test that checks whether the Hamming distance between two strings is at most  $k$ .

**Corollary 5.5.** *Let  $P$  be a set of  $n$  strings in  $\Sigma^\ell$ , where  $\ell \leq n$ . With probability at least  $1 - n^{-\beta}$ , for all  $U, V \in P$ :*

- 1) if  $\|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 \leq (1 + \alpha)k$ , then  $d_H(U, V) \leq (1 + \alpha) \cdot k$ ;
- 2) if  $\|\text{sk}_\alpha(U) - \text{sk}_\alpha(V)\|^2 > (1 + \alpha)k$ , then  $d_H(U, V) \geq k$ .



## 2.1 The Twenty Questions Game

Consider the following version of the classic game “Twenty Questions”. There are two players: Paul and Carole; Carole thinks of two numbers  $A, B$  between 0 and  $N$ , and Paul must return some number in  $[A, B]$ . He is allowed to ask questions of form “Is  $x \leq A$ ?”, for any  $x \in [0, N]$ . If  $x \leq A$ , Carole must return YES; If  $A < x \leq B$ , she can return anything; and if  $B < x$ , she must return NO. Paul must return the answer after having asked at most  $Q$  questions where Carole can tell at most  $\lceil \rho Q \rceil$  lies, and only in the case when  $x \leq A$ .

We show that Paul has a winning strategy for  $Q = \Theta(\log n)$  and any  $\rho < 1/3$  by a black-box reduction to the result of Dhagat, Gács, and Winkler [43] who showed a winning strategy for  $A = B$ .

**Theorem 5.6** ([43]). *For  $A = B$ , Paul has a winning strategy for all  $\rho < \frac{1}{3}$  asking  $Q = \left\lceil \frac{8 \log N}{(1-3\rho)^2} \right\rceil$  questions.*

This result is obtained by maintaining a stack of trusted intervals. Once Paul knows that  $A$  is between  $\ell$  and  $r$ , where  $\ell \leq r$ , he checks whether  $A$  is in the left or the right half of the interval  $[\ell, r]$ . If no inconsistencies appear (like  $A < \ell$  or  $r < A$ ), he pushes the new interval to the stack, else he removes the interval  $[\ell, r]$  from the stack of trusted intervals. After  $Q$  rounds, Paul returns the only number in the top interval in the stack, which is guaranteed to have length 1 and to contain  $A$ . We give the pseudocode of Paul’s strategy in Algorithm 5.2. By  $\text{Carole}(x)$ , we denote the answer of Carole for a question “Is  $x \leq A$ ?”.

---

### Algorithm 5.2 The Twenty Questions game

---

```

1:  $Q \leftarrow \left\lceil \frac{8 \log N}{(1-3\rho)^2} \right\rceil$ 
2:  $S \leftarrow \{[0, N]\}$ 
3: for  $i = 1, 2, \dots, Q/2$  do
4:    $I = [\ell, r] \leftarrow S.\text{top}()$ 
5:    $\text{mid} \leftarrow \left\lceil \frac{\ell+r}{2} \right\rceil$ 
6:   if  $\text{Carole}(\text{mid})$  then
7:     if  $\text{Carole}(r)$  then  $S.\text{pop}()$             $\triangleright$  The answer is inconsistent with  $I$ ; remove  $I$ .
8:     else  $S.\text{push}([\text{mid}, r])$ 
9:   else
10:    if  $\text{Carole}(\ell)$  then  $S.\text{push}([\ell, \text{mid} - 1])$ 
11:    else  $S.\text{pop}()$             $\triangleright$  The answer is inconsistent with  $I$ ; remove  $I$ .
```

---

We now show a winning strategy for our variant of the game.

**Corollary 5.7.** *For  $A \leq B$ , Paul has a winning strategy for all  $\rho < \frac{1}{3}$  asking  $Q = \frac{8 \log N}{(1-3\rho)^2}$  questions.*

*Proof.* We introduce just one change to Algorithm 5.2, namely, we return the argument of the largest YES obtained in the course of the algorithm. From the problem statement it follows that the answer is at most  $B$ . We shall now prove that the answer is at least  $A$ . If Carole ever returned YES for  $A < x \leq B$ , then it is obviously the case. Otherwise, Carole actually behaved as if she had  $A = B$  in mind: apart from the small fraction of erroneous answers, she returned YES for  $x \leq A$ , and NO for  $x > A$ . Thus, the strategy of Dhagat, Gács, and Winkler ends up with  $A$  as the answer (and this must be due to a YES for  $x = A$ ).  $\square$

### 3 LCS with Approximately $k$ Mismatches

In this section, we prove Theorem 5.1. Let us first introduce a decision variant of the LCS with Approximately  $k$  Mismatches problem.

**Problem 2.** Two strings  $X, Y$  of length at most  $n$ , integers  $k, \ell$ , and a constant  $\varepsilon > 0$  are given. We must return:

1. YES if  $\ell \leq \text{LCS}_k(X, Y)$ ;
2. Anything if  $\text{LCS}_k(X, Y) < \ell \leq \text{LCS}_{(1+\varepsilon)k}(X, Y)$ ;
3. NO if  $\text{LCS}_{(1+\varepsilon)k}(X, Y) < \ell$ .

If we return YES, we must also give a witness pair of length- $\ell$  substrings  $S_1$  and  $S_2$  of  $X$  and  $Y$ , respectively, such that  $d_H(S_1, S_2) \leq (1 + \varepsilon)k$ .

The decision variant of the LCS with Approximately  $k$  Mismatches problem can be reduced to the following  $(c, r)$ -Approximate Near Neighbour problem.

**Problem 3.** In the  $(c, r)$ -Approximate Near Neighbour problem with failure probability  $f$ , the aim is, given a set  $P$  of  $n$  points in  $\mathbb{R}^d$ , to construct a data structure supporting the following queries: given any point  $q \in \mathbb{R}^d$ , if there exists  $p \in P$  such that  $\|p - q\| \leq r$ , then return some point  $p' \in P$  such that  $\|p' - q\| \leq cr$  with probability at least  $1 - f$ .

Using the reduction, we will show our first solution to the LCS with Approximately  $k$  Mismatches decision problem based on the result of Andoni and Razenshteyn [194], who showed that for any constant  $f$ , there is a data structure for the  $(c, r)$ -Approximate Near Neighbour problem that has  $\mathcal{O}(n^{1+\rho+o(1)} + d \cdot n)$  size,  $\mathcal{O}(d \cdot n^{\rho+o(1)})$  query time, and  $\mathcal{O}(d \cdot n^{1+\rho+o(1)})$  preprocessing time, where  $\rho = 1/(2c^2 - 1)$ .

**Lemma 5.8.** Assume an alphabet of constant size  $\sigma$ . The decision variant of LCS with Approximately  $k$  Mismatches can be solved in space  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  and  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  time. The answer is correct with constant probability.

*Proof.* Let  $P$  be the set of all length- $\ell$  substrings of  $X$  and  $Q$  be the set of all length- $\ell$  substrings of  $Y$ , all encoded in binary using the morphism  $\mu$  (see Section 2). We start by applying the dimension reduction procedure of Corollary 5.4 to  $P$  and  $Q$  with  $\alpha = 1/(\log \log n)^{\Theta(1)}$  and  $\beta = 2$  to obtain sets  $P'$  and  $Q'$ . We can implement the procedure in  $\mathcal{O}(\sigma n \log^2 n (\log \log n)^{\Theta(1)}) = \mathcal{O}(n \log^{2+o(1)} n)$  time by encoding  $X, Y$  using  $\mu$  and running the FFT algorithm [14] for each of the  $\mathcal{O}(\log^{1+o(1)} n)$  rows of the matrix and  $\mu(X), \mu(Y)$ .

To solve the decision variant of LCS with Approximately  $k$  Mismatches, we build the data structure of Andoni and Razenshteyn [194] for  $(\sqrt{(1+\varepsilon)(1-\alpha)}, \sqrt{(1+\alpha)k})$ -Approximate Near Neighbour over  $Q'$ . We make a query for each string in  $P'$ . If, queried for  $\text{sk}_\alpha(S_1) \in P'$ , where  $S_1$  is a length- $\ell$  substring of  $X$ , the data structure outputs  $\text{sk}_\alpha(S_2) \in Q'$ , where  $S_2$  is a length- $\ell$  substring of  $Y$ , then we compute  $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S_2)\|^2$ . If it is at most  $(1 + \varepsilon)k$ , we output YES and the witness pair  $(S_1, S_2)$  of substrings. As the length of vectors in  $P', Q'$  is  $d = \mathcal{O}(\log^{1+o(1)} n)$ , we obtain the desired complexity.

To show that the algorithm is correct, suppose that there are length- $\ell$  substrings  $S_1$  and  $S_2$  of  $X$  and  $Y$ , respectively, with  $d_H(S_1, S_2) \leq k$ . By Corollary 5.4,  $\|\text{sk}_\alpha(S_1), \text{sk}_\alpha(S_2)\| \leq \sqrt{(1+\alpha)k}$  holds with probability at least  $1 - 1/n$ . Then, when querying for  $\text{sk}_\alpha(S_1)$ , with constant probability the data structure will output a string  $\text{sk}_\alpha(S'_2)$  such that  $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S'_2)\|^2 \leq (1 + \varepsilon)(1 - \alpha^2)k \leq (1 + \varepsilon)k$ . Then, our algorithm will return YES.

On the other hand, if we output YES with a witness pair  $(S_1, S_2)$ , then  $\|\text{sk}_\alpha(S_1) - \text{sk}_\alpha(S_2)\|^2 \leq (1 + \varepsilon)k$  implies  $d_H(S_1, S_2) \leq (1 + \varepsilon)k$  with high probability by Corollary 5.4.  $\square$

While this solution is very fast, it uses quite a lot of space. Furthermore, the data structure of [194] that we use as a black box applies highly non-trivial techniques. To overcome these two disadvantages, we will show a different solution based on a careful implementation of ideas first introduced in [101] that showed a data structure for approximate text indexing with mismatches. In [294], the authors developed these ideas further to show an algorithm that solves the LCS with Approximately  $k$  Mismatches problem in  $\mathcal{O}(n^{1+1/(1+\varepsilon)})$  space and  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$  time for  $\varepsilon \in (0, 2)$  with constant error probability. In this work, we significantly improve and simplify the approach to show the following result:

**Theorem 5.9.** *Assume an alphabet of arbitrary size  $\sigma = n^{\mathcal{O}(1)}$ . The decision variant of LCS with Approximately  $k$  Mismatches can be solved in  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$  time and  $\mathcal{O}(n)$  space. The answer is correct with constant probability.*

Let us defer the proof of the theorem until Section 3.1 and start by explaining how we use Lemma 5.8 and Theorem 5.9 and the Twenty Questions game to show Theorem 5.1.

*Proof of Theorem 5.1.* We will rely on the modified version of the Twenty Questions game that we described in Section 2.1. In our case,  $A = \text{LCS}_k(X, Y)$  and  $B = \text{LCS}_{(1+\varepsilon)k}(X, Y)$ . For Carole, we use either the algorithm of Lemma 5.8, or the algorithm of Theorem 5.9, with an additional procedure verifying the witness pair  $(S_1, S_2)$  character by character to check that it indeed satisfies  $d_H(S_1, S_2) \leq (1+\varepsilon)k$ . We output the longest pair of (honest) witness substrings found across all iterations. We will return a correct answer assuming that the fraction of errors is  $\rho < \frac{1}{3}$ . Recall that the algorithm solves the decision variant of the LCS with Approximately  $k$  Mismatches problem incorrectly with probability not exceeding a constant  $\delta$ , and we can ensure  $\delta < \frac{1}{3}$  by repeating it a constant number of times. It means that Carole can answer an individual question erroneously with probability less than  $\frac{1}{3}$ . Therefore, for a sufficiently large constant in the number of queries  $Q = \Theta(\log n)$ , the fraction of erroneous answers is  $\rho < \frac{1}{3}$  with high probability by Chernoff–Hoeffding bounds. The claim of the theorem follows immediately from Lemma 5.8 and Theorem 5.9.  $\square$

### 3.1 Proof of Theorem 5.9

We first give an algorithm for the decision version of the LCS with Approximately  $k$  Mismatches problem that uses  $\mathcal{O}(n \log n)$  space and  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n + \sigma n \log^2 n)$  time, and then we improve the space and time complexity.

We assume to have fixed a Karp–Rabin fingerprinting  $\varphi$  for a prime  $q = \Omega(\max\{n^5, \sigma\})$  and an integer  $r \in \mathbb{Z}_q$ . With error probability inverse polynomial in  $n$ , we can find such  $q$  in  $\mathcal{O}(\log^{\mathcal{O}(1)} n)$  time; see [165, 93].

Let  $\Pi$  be the set of all projections of strings of length  $\ell$  onto a single position, i.e., the value  $\pi_i(S)$  of the  $i$ -th projection on a string  $S$  of length  $\ell$  is simply its  $i$ -th character  $S[i]$ . More generally, for a length- $\ell$  string  $S$  and a function  $h = (\pi_{a_1}, \dots, \pi_{a_m}) \in \Pi^m$ , we define  $h(S)$  as  $S[a_1]S[a_2] \cdots S[a_m]$ .

Let  $p_1 = 1 - k/\ell$  and  $p_2 = 1 - (1+\varepsilon)k/\ell$ . We assume that  $(1+\varepsilon)k < \ell$  in order to guarantee  $p_1 > p_2 > 0$ ; the problem is trivial if  $(1+\varepsilon)k \geq \ell$ . Further, let  $m = \left\lceil \log_{p_2} \frac{1}{n} \right\rceil$ .

We choose a set  $\mathcal{H}$  of  $L = \Theta(n^{1/(1+\varepsilon)})$  hash functions in  $\Pi^m$  uniformly at random. Let  $C_\ell^\mathcal{H}$  be the multiset of all collisions of length- $\ell$  substrings of  $X$  and  $Y$  under the functions from  $\mathcal{H}$ , i.e.  $C_\ell^\mathcal{H} = \{(X[i, i+\ell-1], Y[j, j+\ell-1], h) : \varphi(h(X[i, i+\ell-1])) = \varphi(h(Y[j, j+\ell-1])), 1 \leq i \leq |X| - \ell, 1 \leq j \leq |Y| - \ell\}$ .

We will perform two tests. The first test chooses an arbitrary subset  $C' \subseteq C_\ell^\mathcal{H}$  of size  $|C'| = \min\{4nL, |C_\ell^\mathcal{H}|\}$  and, for each collision  $(S_1, S_2, h) \in C'$ , computes  $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2$ . If

this value is at most  $(1 + \varepsilon)k$ , then the algorithm returns YES and the pair  $(S_1, S_2)$  as a witness. The second test chooses a collision  $(S_1, S_2, h) \in C_\ell^\mathcal{H}$  uniformly at random and computes the Hamming distance between  $S_1$  and  $S_2$  character by character in  $\mathcal{O}(\ell) = \mathcal{O}(n)$  time. If the Hamming distance is at most  $(1 + \varepsilon)k$ , the algorithm returns YES and the witness pair  $(S_1, S_2)$ . Otherwise, the algorithm returns NO. See Algorithm 5.3.

---

**Algorithm 5.3** LCS with Approximately  $k$  Mismatches (decision variant)
 

---

- 1: Choose a set  $\mathcal{H}$  of  $L$  functions from  $\Pi^m$  uniformly at random
  - 2:  $C_\ell^\mathcal{H} = \{(S_1, S_2, h) : S_1, S_2 \text{ length-}\ell \text{ substrs. of } X, Y \text{ resp. and } \varphi(h(S_1)) = \varphi(h(S_2))\}$
  - 3: Choose an arbitrary subset  $C' \subseteq C_\ell^\mathcal{H}$  of size  $\min\{4nL, |C_\ell^\mathcal{H}|\}$
  - 4: Compute  $\text{sk}_\varepsilon(\cdot)$  sketches for all length- $\ell$  substrings of  $X, Y$
  - 5: **for**  $(S_1, S_2, h) \in C'$  **do**
  - 6:     **if**  $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2 \leq (1 + \varepsilon)k$  **then return** (YES,  $(S_1, S_2)$ )
  - 7: Draw a collision  $(S_1, S_2, h) \in C_\ell^\mathcal{H}$  uniformly at random
  - 8: **if**  $d_H(S_1, S_2) \leq (1 + \varepsilon)k$  **then return** (YES,  $(S_1, S_2)$ )
  - 9: **return** NO
- 

We must explain how we compute  $C_\ell^\mathcal{H}$  and choose the collisions that we test. We consider each hash function  $h \in \mathcal{H}$  in turn. Let  $h = (\pi_{a_1}, \dots, \pi_{a_m})$ . Recall that for a string  $S$  of length  $\ell$  we define  $h(S)$  as  $S[a_1]S[a_2] \cdots S[a_m]$ . Consequently,  $\varphi(h(S)) = (\sum_{i=1}^m r^{i-1} S[a_i]) \bmod q$ . We create a vector  $U$  of length  $\ell$  where each entry is initialised with 0. For each  $i$ , we add  $r^{i-1} \bmod q$  to the  $a_i$ -th entry of  $U$ . Finally, we run the FFT algorithm [14] for  $U$  and  $X, Y$  in the field  $\mathbb{Z}_q$ , and sort the resulting values. We obtain a list of sorted values that we can use to generate the collisions. Namely, consider some fixed value  $z$ . Assume that there are  $x$  substrings of  $X$  and  $y$  substrings of  $Y$  of length  $\ell$  such that the fingerprint of their projection is equal to  $z$ . The value  $z$  then gives  $xy$  collisions, and we can generate each one of them in constant time. This explains how to choose the subset  $C'$  in  $\mathcal{O}(nL \log n)$  time.

To draw a collision from  $C_\ell^\mathcal{H}$  uniformly at random, we could simply compute the total number of collisions across all functions  $h \in \mathcal{H}$ , draw a number in  $[1, |C_\ell^\mathcal{H}|]$ , and generate the corresponding collision. However, this would require to generate the collisions twice. Instead, we use the weighted reservoir sampling algorithm [23]. We divide all collisions into subsets according to the values of fingerprints. We assume that the weighted reservoir sampling algorithm receives the fingerprint values one-by-one, as well as the number of corresponding collisions. At all times, the algorithm maintains a “reservoir” containing one fingerprint value and a random collision corresponding to this value. When a new value  $z$  with  $xy$  collisions arrives, the algorithm replaces the value in the reservoir with  $z$  and a random collision with some probability. Note that to select a random collision it suffices to choose a pair from  $[1, x] \times [1, y]$  uniformly at random. It is guaranteed that if for a value  $z$  we have  $xy$  collisions, the algorithm will select  $z$  with probability  $xy/|C_\ell^\mathcal{H}|$ . Consequently, after processing all values, the reservoir will contain a collision chosen from  $C_\ell^\mathcal{H}$  uniformly at random.

**Lemma 5.10.** *Algorithm 5.3 uses  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n + \sigma n \log^2 n)$  time and  $\mathcal{O}(n \log n)$  space.*

*Proof.* Computing the sketches (Line 4) takes  $\mathcal{O}(\sigma n \log^2 n)$  time and  $\mathcal{O}(n \log n)$  space. Computing the collisions and choosing the collisions to test takes  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$  time and  $\mathcal{O}(n)$  space in total. Testing  $\min\{4nL, |C_\ell^\mathcal{H}|\}$  collisions (Line 5) takes  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log n)$  time and constant space. Computing the Hamming distance for a random collision (Line 8) takes  $\mathcal{O}(\ell) = \mathcal{O}(n)$  time and constant space.  $\square$

**Lemma 5.11.** *Let  $S_1$  and  $S_2$  be two length- $\ell$  substrings of  $X$  and  $Y$ , with  $d_H(S_1, S_2) \leq k$ . If  $L = \Theta(n^{1/(1+\varepsilon)})$  is large enough, then, with probability at least  $3/4$ , there exists a function  $h \in \mathcal{H}$  such that  $h(S_1) = h(S_2)$ .*

*Proof.* Consider a function  $h = (\pi_{a_1}, \dots, \pi_{a_m})$  drawn from  $\Pi^m$  uniformly at random. The probability of  $h(S_1) = h(S_2)$  is at least  $p_1^m$ . Due to  $p_1 \leq 1$ , we have

$$p_1^m = p_1^{\lceil \log_{p_2} \frac{1}{n} \rceil} \geq p_1^{1 + \log_{p_2} \frac{1}{n}} = p_1 \cdot n^{-\frac{\log p_1}{\log p_2}}.$$

Moreover,  $p_1 = 1 - \frac{k}{\ell}$  and  $(1 + \varepsilon)k < \ell$  yield  $p_1 > 1 - \frac{1}{1+\varepsilon} = \frac{\varepsilon}{1+\varepsilon}$ , whereas Bernoulli's inequality implies  $p_2 = 1 - (1 + \varepsilon)\frac{k}{\ell} \leq (1 - \frac{k}{\ell})^{1+\varepsilon} = p_1^{1+\varepsilon}$ , i.e.,  $\log p_2 \leq (1 + \varepsilon) \log p_1$ . Therefore,

$$p_1^m \geq p_1 \cdot n^{-\frac{\log p_1}{\log p_2}} \geq \frac{\varepsilon}{1+\varepsilon} \cdot n^{-\frac{1}{1+\varepsilon}}.$$

Hence, we can choose the constant in  $L = |\mathcal{H}|$  so that the claim of the lemma holds.  $\square$

**Lemma 5.12.** *If  $|\mathcal{C}_\ell^{\mathcal{H}}| > 4nL$  and  $(S_1, S_2, h)$  is a uniformly random element of  $\mathcal{C}_\ell^{\mathcal{H}}$ , then  $\Pr[d_H(S_1, S_2) \geq (1 + \varepsilon)k] \leq \frac{1}{2}$ .*

*Proof.* Consider length- $\ell$  substrings  $S_1, S_2$  of  $X, Y$ , respectively, such that  $d_H(S_1, S_2) \geq (1 + \varepsilon)k$ , and a hash function  $h$ . Let us bound the probability of  $(S_1, S_2, h) \in \mathcal{C}_\ell^{\mathcal{H}}$ . There two possible cases: either  $h(S_1) \neq h(S_2)$  but  $\varphi(h(S_1)) = \varphi(h(S_2))$ , or  $h(S_1) = h(S_2)$ . The probability of the first event is bounded by the collision probability of Karp–Rabin fingerprints, which is at most  $1/n$ . Let us now bound the probability of the second event. Since  $d_H(S_1, S_2) \geq (1 + \varepsilon)k$ , we have  $\Pr[h(S_1) = h(S_2)] \leq p_2^m \leq 1/n$ , where the last inequality follows from the definition of  $m$ . Therefore, the probability that for some function  $h \in \mathcal{H}$  we have  $\varphi(h(S_1)) = \varphi(h(S_2))$  is at most  $2/n$ .

In total, we have  $n^2|\mathcal{H}|$  possible triples  $(S_1, S_2, h)$  so by linearity of expectation, we conclude that the expected number of such triples is at most  $\frac{2}{n}n^2L = 2nL$ . Therefore, the probability to hit a triple  $(S_1, S_2, h)$  such that  $d_H(S_1, S_2) \geq (1 + \varepsilon)k$  when drawing from  $\mathcal{C}_\ell^{\mathcal{H}}$  uniformly at random is at most  $2nL/|\mathcal{C}_\ell^{\mathcal{H}}| \leq 2nL/4nL = 1/2$ .  $\square$

Below, we combine the previous results to prove that, with constant probability, Algorithm 5.3 correctly solves the decision variant of the LCS with Approximately  $k$  Mismatches problem. Note that we can reduce the error probability to an arbitrarily small constant  $\delta > 0$ : it suffices to repeat the algorithm a constant number of times.

**Corollary 5.13.** *With non-zero constant probability, Algorithm 5.3 solves the decision variant of LCS with Approximately  $k$  Mismatches correctly.*

*Proof.* Suppose first that  $\ell \leq \text{LCS}_k(X, Y)$ , which means that there are two length- $\ell$  substrings  $S_1, S_2$  of  $X, Y$  such that  $d_H(S_1, S_2) \leq k$ . By Lemma 5.11, with probability at least  $3/4$ , there exists a function  $h \in \mathcal{H}$  such that  $h(S_1) = h(S_2)$ . In other words,  $(S_1, S_2, h) \in \mathcal{C}_\ell^{\mathcal{H}}$  with probability at least  $\frac{3}{4}$ . If  $|\mathcal{C}_\ell^{\mathcal{H}}| < 4nL$ , we will find this triple and it will pass the test with probability at least  $1 - n^{-6}$ . If  $|\mathcal{C}_\ell^{\mathcal{H}}| \geq 4nL$ , then by Lemma 5.12 the Hamming distance between  $S_1, S_2$ , where  $(S_1, S_2, h)$  was drawn from  $\mathcal{C}_\ell^{\mathcal{H}}$  uniformly at random, is at most  $(1 + \varepsilon)k$  with probability  $\geq 1/2$ , and therefore this pair will pass the test with probability  $\geq 1/2$ . It follows that in this case the algorithm outputs YES with constant probability.

Suppose now that  $\ell > \text{LCS}_{(1+\varepsilon)k}(X, Y)$ . In this case, the Hamming distance between any pair of length- $\ell$  substrings of  $X$  and  $Y$  is at least  $(1 + \varepsilon)k$ , so none of them will ever pass the second test and none of them will pass the first test with constant probability.  $\square$

We now improve the space of the algorithm to linear. Note that the only reason why we needed  $\mathcal{O}(n \log n)$  space is that we precompute and store the sketches for the Hamming distance. Below we explain how to overcome this technicality.

First, we do not precompute the sketches. Second, we process the collisions in  $C'$  in batches of size  $n$ . Consider one of the batches,  $\mathcal{B}$ . For each collision  $(S_1, S_2, h) \in \mathcal{B}$  we must compute  $\|\text{sk}_\varepsilon(S_1) - \text{sk}_\varepsilon(S_2)\|^2$ . We initialize a counter for every collision, setting it to zero initially. The number of rounds in the algorithm will be equal to the length of the sketches, and, in round  $i$ , the counter for a collision  $(S_1, S_2, h) \in \mathcal{B}$  will contain the squared  $L_2$  distance between the length- $i$  prefixes of  $\text{sk}_\varepsilon(S_1)$  and  $\text{sk}_\varepsilon(S_2)$ . In more detail, let  $\mathcal{S}$  be the set of all substrings of  $X, Y$  that participate in the collisions in  $\mathcal{B}$ . Recall that all these substrings have length  $\ell$ . At round  $i$ , we compute the  $i$ -th coordinate of the sketches of the substrings in  $\mathcal{S}$ . By definition, the  $i$ -th coordinate is the dot product of the  $i$ -th row of  $c \cdot M$ , where  $c$  and  $M$  are as in Corollary 5.4, and a substring encoded using  $\mu$ . Hence, we can compute the coordinate using the FFT algorithm [14] in  $\mathcal{O}(\sigma n \log n)$  time and  $\mathcal{O}(n)$  space. When we have the coordinate  $i$  computed, we update the counters for the collisions and repeat.

At any time, the algorithm uses  $\mathcal{O}(n)$  space. Compared to the time consumption proven in Lemma 5.10, the algorithm spends an additional  $\mathcal{O}(\sigma n^{1+1/(1+\varepsilon)} \log^2 n)$  time for computing the coordinates of the sketches. Therefore, in total the algorithm uses  $\mathcal{O}(\sigma n^{1+1/(1+\varepsilon)} \log^2 n) = \mathcal{O}(n^{1+1/(1+\varepsilon)} \log^2 n)$  time and  $\mathcal{O}(n)$  space. For constant-size alphabets, this completes the proof of Theorem 5.9. For alphabets of arbitrary size, we replace the sketches from Section 2 with the sketches defined in [294] to achieve the desired complexity. We note that we could use the sketches [294] for small-size alphabets as well, but their lengths hide a large constant.

## 4 Experiments

We now present results of experimental evaluation of the second solution presented in Theorem 5.1.

*Methodology and test environment.* The baselines and our solution are written in C++11 and compiled with optimizations using gcc 7.4.0. The experimental results were generated on an Intel Xeon E5-2630 CPU using 128 GiB RAM. To ensure the reproducibility of our results, our complete experimental setup, including data files, is available at [https://github.com/fnareoh/LCS\\_Approx\\_k\\_mis](https://github.com/fnareoh/LCS_Approx_k_mis).

*Baseline.* The only other solution to the LCS with Approximately  $k$  Mismatches problem was presented in [294], however, it has a worse complexity and is likely to be unpractical because it uses a very complex class of hash functions. We therefore chose to compare our algorithm against algorithms for the LCS with  $k$  Mismatches problem. To the best of our knowledge, none of the existing algorithms has been implemented. We implemented the solution to LCS with  $k$  Mismatches by Flouri et al., which we refer to as FGKU [204]. (The other algorithms seem to be too complex to be efficient in practice.) The main idea of the algorithm of Flouri et al. is that if we know that the longest common substring with  $k$  mismatches is obtained by a substring of  $X$  that starts at a position  $p$  and a substring of  $Y$  that starts at a position  $p + i$ , then we can find it by scanning  $X$  and  $Y[i, |Y|]$  in linear time; see Algorithm 5.4 for details.

*Details of implementation.* We made several adjustments to the theoretical algorithm we described. First, we use the fact that  $A = \text{LCS}(X, Y) + k \leq \text{LCS}_k(X, Y) \leq B = (k + 1) \cdot \text{LCS}(X, Y) + k$  to bound the interval in the Twenty Questions game. We also treated the number of questions in the Twenty Questions game and  $L$ , the size of the set of hash functions  $\mathcal{H}$ , as parameters that trade time for accuracy, and put the number of questions to  $2 \log(B - A)$  in the Twenty Questions game and  $L = n^{1/(1+\varepsilon)}/16$ . In Line 6 of Algorithm 5.3, we used sketches to estimate the Hamming distance. In practice, we computed the Hamming distance via character-

---

**Algorithm 5.4** FGKU algorithm
 

---

```

1:  $n \leftarrow |X|, m \leftarrow |Y|$ 
2:  $l \leftarrow 0, r_1 \leftarrow 0, r_2 \leftarrow 0$ 
3: for  $d \leftarrow -m + 1$  to  $n - 1$  do
4:    $i \leftarrow \max(-d, 0) + d, j \leftarrow \max(-d, 0)$ 
5:    $Q \leftarrow \emptyset, s \leftarrow 0, p \leftarrow 0$ 
6:   while  $p \leq \min(n - i, m - j) - 1$  do
7:     if  $X[i + p] \neq Y[j + p]$  then
8:       if  $|Q| = k$  then
9:          $s \leftarrow \min Q + 1$ 
10:        DEQUEUE( $Q$ )
11:        ENQUEUE( $Q, p$ )
12:      $p \leftarrow p + 1$ 
13:     if  $p - s > l$  then
14:        $l \leftarrow p - s, r_1 \leftarrow i + s, r_2 \leftarrow j + s$ 
    
```

---

by-character comparison when  $\ell$  is small compared to  $k$  and via kangaroo jumps otherwise [31]. Also, when the length  $\ell$  in Algorithm 5.3 is smaller than  $2 \log n$ , we compute the hash values of the  $\ell$ -length substrings of  $S_1$  and  $S_2$  naively, instead of using the FFT algorithm [14].

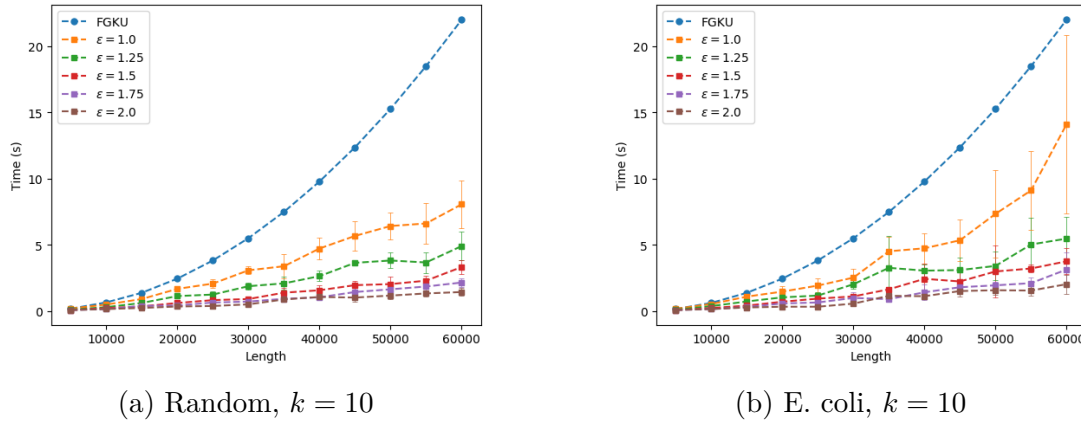


Figure 5.1: Comparison of the FGKU algorithm versus our algorithm for  $k = 10$  and different values of  $\varepsilon$ . Large standard deviation for length 60000 is caused by an outlier with very long longest common substring with  $k$  mismatches.

*Data sets and results.* We considered  $k \in \{10, 25, 50\}$  and  $\varepsilon \in \{1.0, 1.25, 1.5, 1.75, 2.0\}$ . We tested the algorithms on pairs of random strings (each character is selected independently and uniformly from a four-character alphabet  $\{A, T, G, C\}$ ) and on pairs of strings extracted at random from the E. coli genome. The lengths of the strings in each pair are equal and vary from 0 to 60000 with a step of 5000. All timings reported are averaged over ten runs. Figures 5.1- 5.3 show the results for  $k = 10, 25, 50$ . We note that for  $\varepsilon = 1$  and  $k = 10, 25$ , the standard deviation of the running time on the E. coli data set is quite large, which is probably caused by our choice of the method to compute the Hamming distance between substrings, but for all other parameter combinations it is within the standard range. We can see that the time decreases when  $\varepsilon$  grows, which is coherent with the theoretical complexity.

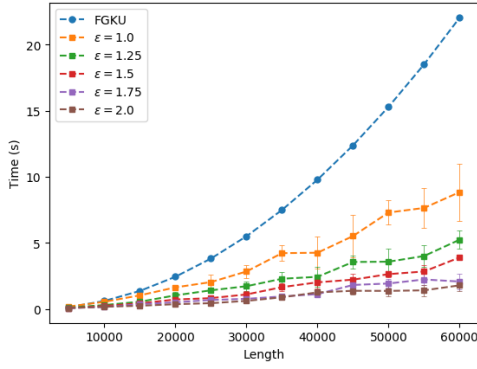
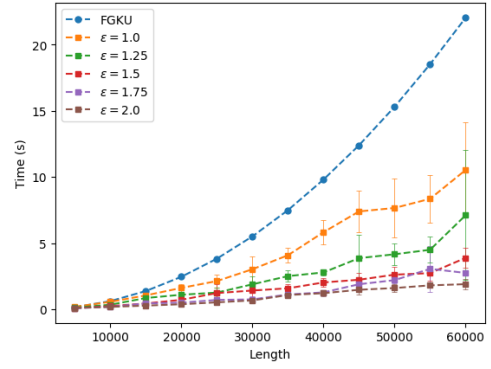
(a) Random,  $k = 25$ (b) E. coli,  $k = 25$ 

Figure 5.2: Comparison of the FGKU algorithm versus our algorithm for  $k = 25$  and different values of  $\varepsilon$ .

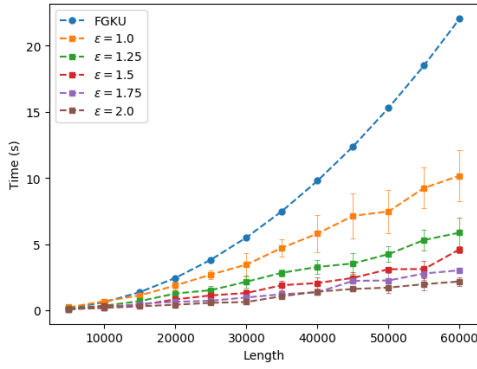
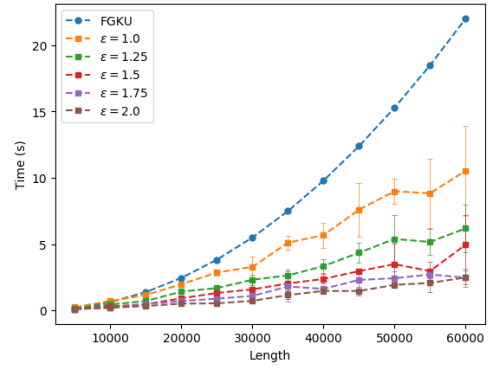
(a) Random,  $k = 50$ (b) E. coli,  $k = 50$ 

Figure 5.3: Comparison of the FGKU algorithm versus our algorithm for  $k = 50$  and different values of  $\varepsilon$ .

As for the accuracy, note that our algorithm cannot return a pair of strings at Hamming distance more than  $(1 + \varepsilon)k$ , and so the only risk is returning strings which are too short. Consequently, we measured the accuracy of our implementation by the ratio of the length  $\text{LCS}_{\tilde{k}}(X, Y)$  returned by our algorithm divided by  $\text{LCS}_k(X, Y)$  computed by the dynamic programming. We estimate  $r_{\min}(\varepsilon, k) = \min_{X, Y} (\text{LCS}_{\tilde{k}}(X, Y) / \text{LCS}_k(X, Y))$  and  $r_{\max}(\varepsilon, k) = \max_{X, Y} (\text{LCS}_{\tilde{k}}(X, Y) / \text{LCS}_k(X, Y))$  by computing  $\text{LCS}_{\tilde{k}}$  and  $\text{LCS}_k$  for 10 pairs of strings for each length from 5000 to 60000 with step of 5000, as well as the error rate, i.e. the percentage of experiments where  $\text{LCS}_{\tilde{k}}(X, Y)$  is shorter than  $\text{LCS}_k(X, Y)$  (see Table 5.1). Not surprisingly,  $r_{\min}$  and  $r_{\max}$  grow as  $k$  and  $\varepsilon$  grow, while the error rate drops. Even though there is no theoretical upper bound on  $r_{\max}$ , the latter is at most 2.24 at all times. We also note that even in the cases when the error rate is non-negligible,  $\text{LCS}_{\tilde{k}} \geq 0.86 \cdot \text{LCS}_k$ , in other words, our algorithm returns a reasonable approximation of  $\text{LCS}_k$ .



	Random						E. coli					
	$k = 10$		$k = 25$		$k = 50$		$k = 10$		$k = 25$		$k = 50$	
$\varepsilon = 1.0$	0.95	1.41	1.12	1.46	1.27	1.54	0.89	1.34	0.94	1.48	0.97	1.59
	err = 3%		err = 0%		err = 0%		err = 33%		err = 13%		err = 3%	
$\varepsilon = 1.25$	0.97	1.47	1.15	1.63	1.44	1.78	0.88	1.48	0.98	1.56	0.99	1.73
	err = 1%		err = 0%		err = 0%		err = 28%		err = 5%		err = 3%	
$\varepsilon = 1.5$	1.05	1.57	1.37	1.76	1.55	1.91	0.88	1.45	0.96	1.67	0.99	1.89
	err = 0%		err = 0%		err = 0%		err = 17%		err = 3%		err = 3%	
$\varepsilon = 1.75$	1.02	1.69	1.46	1.86	1.72	2.12	0.88	1.58	0.95	1.84	1.02	2.15
	err = 0%		err = 0%		err = 0%		err = 17%		err = 2%		err = 0%	
$\varepsilon = 2.0$	1.10	1.72	1.59	2.00	1.89	2.24	0.91	1.77	1.01	2.10	1.00	2.19
	err = 0%		err = 0%		err = 0%		err = 9%		err = 0%		err = 1%	

Table 5.1: Accuracy of the LCS with Approximately  $k$  Mismatches algorithm. For each  $k$  and  $\varepsilon$ , we show  $r_{\min}(\varepsilon, k)$ ,  $r_{\max}(\varepsilon, k)$ , as well as the error rate.

## Publication

This chapter corresponds to the extended version of the following publication: Garance Gourdel, Anne Driemel, Pierre Peterlongo, and Tatiana Starikovskaya, “Pattern Matching Under DTW Distance”, in: *String Processing and Information Retrieval - 29<sup>th</sup> International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, ed. by Diego Arroyuelo and Barbara Poblete, vol. 13617, Lecture Notes in Computer Science, Springer, 2022, pp. 315–330, DOI: 10.1007/978-3-031-20643-6\_23.

In this work, we consider the problem of pattern matching under the dynamic time warping (DTW) distance motivated by potential applications in the analysis of biological data produced by the third generation sequencing. To measure the DTW distance between two strings, one must “warp” them, that is, double some letters in the strings to obtain two equal-lengths strings, and then sum the distances between the letters in the corresponding positions. When the distances between letters are integers, we show that for a pattern  $P$  with  $m$  runs and a text  $T$  with  $n$  runs:

1. There is an  $\mathcal{O}(m+n)$ -time algorithm that computes all locations where the DTW distance from  $P$  to  $T$  is at most 1;
2. There is an  $\mathcal{O}(kmn)$ -time algorithm that computes all locations where the DTW distance from  $P$  to  $T$  is at most  $k$ .

As a corollary of the second result, we also derive an approximation algorithm for general metrics on the alphabet.

## 1 Introduction

Introduced more than forty years ago [18], the dynamic time warping (DTW) distance has become an essential tool in the time series analysis and its applications due to its ability to preserve the signal despite speed variation in compared sequences. To measure the DTW distance between two discrete temporal sequences, one must “warp” them, that is, replace some data items in the sequences with multiple copies of themselves to obtain two equal-lengths sequences, and then sum the distances between the data items in the corresponding positions.

The DTW distance has been extensively studied for parameterized curves — sequences where the data items are points in a multidimensional space — specifically, in the context of locality sensitive hashing and nearest neighbor search [240, 258]. In this work, we focus on a somewhat simpler, but surprisingly much less studied setting when the data items are elements of a finite set, the alphabet. Following traditions, we call such sequences *strings*.

The classical textbook dynamic programming algorithm computes the DTW distance between two  $N$ -length strings in  $\mathcal{O}(N^2)$  time and space. Unfortunately, unless the Strong Exponential Time Hypothesis is false, there is no algorithm with strongly subquadratical time even for ternary alphabets [191, 198, 295]. On the other hand, very recently Gold and Sharir [271]

showed the first weakly subquadratic time algorithm (to be more precise, the time complexity of the algorithm is  $\mathcal{O}(N^2 \log \log \log N / \log \log N)$ ). Kuszmaul [295] gave a  $\mathcal{O}(kN)$ -time algorithm that computes the value of the distance between the strings if it is bounded by  $k$ , assuming that the distance between any two distinct letters of the alphabet is at least one, and used it to derive a subquadratic-time approximation algorithm for the general case. Finally, it is known that binary strings admit much faster algorithms: Abboud, Backurs, and Vassilevska Williams [191] showed an  $\mathcal{O}(N^{1.87})$ -time algorithm followed by a linear-time algorithm by Kuszmaul [341].

The problem of computing the DTW distance has also been studied in the sparse and run-length compressed settings, as well as in the low distance regime. In the sparse setting, we assume that most letters of the string are zeros. Hwang and Gelfand [246] gave an  $\mathcal{O}((s+t)N)$ -time algorithm, where  $s$  and  $t$  denote the number of non-zero letters in each of the two strings. On sparse binary strings, the distance can be computed in  $\mathcal{O}(s+t)$  time [293, 225]. Froese et al. [285] suggested an algorithm with running time  $\mathcal{O}(mN + nM)$ , where  $M, N$  are the length of the strings, and  $m, n$  are the sizes of their run length encodings. If  $n \in \mathcal{O}(\sqrt{N})$  and  $m \in \mathcal{O}(\sqrt{M})$ , their algorithm runs in time  $\mathcal{O}(nm \cdot (n + m))$ . For binary strings, the DTW distance can be computed in  $\mathcal{O}(nm)$  time [202].

Nishi et al. [320] considered the question of computing the DTW distance in the dynamic setting when the strings can be edited, and Sakai and Inenaga [324] showed a reduction from the problem of computing the DTW distance to the problem of computing the longest increasing subsequence, which allowed them to give polynomial-time algorithms for a series of DTW-related problems.

In this work, we focus on the pattern matching variant of the problem: Given a pattern  $P$  and a text  $T$ , one must output the smallest DTW distance between  $P$  and a suffix of  $T[1..r]$  for every position  $r$  of the text.

Our interest to this problem sparks from its potential applications in Third Generation Sequencing (TGS) data comparisons. TGS has changed the genomic landscape as it allows to sequence reads of few dozens of thousand of letters where previous sequencing techniques were limited to few hundred letters [304]. However, TGS suffers from a high error rate (from  $\approx 1$  to 10% depending on the used techniques) mainly due to the fact that the DNA sequences are read and thus sequenced at an uneven speed. The uneven sequencing speed has a major impact in the sequencing quality of DNA regions composed of two or more equal consecutive letters. Those regions, called *homopolymers*, are hardly correctly sequenced as, due to the uneven sequencing speed, their size cannot be precisely determined [338]. In particular, a common post-sequencing task consists in aligning the obtained reads to a reference genome. This enables for instance to predict alternative splicing and gene expression [218] or to detect structural variations [297]. All known aligners use the edit distance, most likely, due to the availability of software tools for the latter (see [274] and references therein). However, we find that the nature of TGS errors is much better described by the DTW distance, which we confirm experimentally in Section 6.

**Our contribution.** As a baseline, the problem of pattern matching under the DTW distance can be solved using dynamic programming in time  $\mathcal{O}(MN)$ , where  $M$  is the length of the pattern and  $N$  of the text (Equation 6.1).

In this work, we aim to show more efficient algorithms for the low-distance regime on run-length compressible data, which is arguably the most interesting setting for the TGS data processing. Formally, in the  $k$ -DTW *problem* we are given an integer  $k > 0$ , a pattern  $P$  and a text  $T$ , and must find all positions  $r$  of the text such that the smallest DTW distance between the pattern  $P$  and a suffix of  $T[1..r]$  does not exceed  $k$ . One might hope that the DTW distance is close enough to the edit distance and thus is amenable to the techniques developed for the latter, such as [61, 37]. In the full version, we show that this is indeed the case for  $k = 1$ :

**Lemma 6.1.** *Given run-length encodings of a pattern  $P$  and of a text  $T$  over an alphabet  $\Sigma$  and a distance  $d : \Sigma \times \Sigma \rightarrow \mathbb{Z}^+$ , the 1-DTW problem can be solved in  $\mathcal{O}(m + n)$  time, where  $m$  is the number of runs in  $P$  and  $n$  is the number of runs in  $T$ . The output is given in a compressed form, with a possibility to retrieve each position in constant time.*

Unfortunately, extending the approach of [61, 37] to higher values of  $k$  seems to be impossible as it is heavily based on the fact that in the edit distance dynamic programming matrix the distances are non-decreasing on every diagonal, which is not the case for the DTW distance (see Fig. 6.1).

In Section 4 we develop a different approach. Interestingly, we show that the value of any cell of the bottom row and the right column of a block of the dynamic programming table (i.e. a subtable formed by a run in the pattern and a run in the text) can be computed in constant time given a constant-time oracle access to the left column and the top row. Combining this with a compact representation of the  $k$ -bounded values, we obtain the following result:

**Theorem 6.2.** *Given run-length encodings of a pattern  $P$  and of a text  $T$  over an alphabet  $\Sigma$  and a distance  $d : \Sigma \times \Sigma \rightarrow \mathbb{Z}^+$ , the  $k$ -DTW problem can be solved in  $\mathcal{O}(kmn)$  time, where  $m$  is the number of runs in  $P$  and  $n$  is the number of runs in  $T$ . The output is given in a compressed form, with a possibility to retrieve each position in constant time.*

We note that while our algorithm can be significantly faster than the baseline, its worst-case time complexity is cubic. We leave it as an open question whether there exists an  $\mathcal{O}(k \cdot (m + n))$ -time algorithm. Finally, in Section 5 we use Theorem 6.2 to derive an approximation algorithm for the general variant of pattern matching under the DTW distance.

		G	G	T	T	T	T	C	T	T	A	T	T	T	T	G	G	T	G	A	T	A
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
A	$\infty$	1	1	1	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	0	1	0
A	$\infty$	2	2	2	2	2	2	2	2	2	0	1	2	2	2	2	2	2	2	0	1	0
T	$\infty$	3	3	2	2	2	2	3	2	2	1	0	0	0	0	1	2	2	3	1	0	1
T	$\infty$	4	4	2	2	2	2	3	2	2	2	0	0	0	0	1	2	2	3	2	0	1
A	$\infty$	5	5	3	3	3	3	3	3	3	2	1	1	1	1	1	2	3	3	2	1	0
T	$\infty$	6	6	3	3	3	3	4	3	3	3	1	1	1	1	2	2	2	3	3	1	1

Figure 6.1: Consider  $P = AATTAT$  and  $T = GGT TTTCTTATTTTG GTGATA$ . A cell  $(i, j)$  contains the smallest DTW distance between  $P[1..i]$  and  $T[1..j]$ , where the distance between two letters equals one if they are distinct and zero otherwise. A non-monotone diagonal of the table is shown in red.

## 2 Preliminaries

We assume a polynomial-size alphabet  $\Sigma$  with  $\sigma$  letters. A *string*  $X$  is a sequence of letters. If the sequence has length zero, it is called the *empty string*. Otherwise, we assume that the letters in  $X$  are numbered from 1 to  $n = |X|$  and denote the  $i$ -th letter by  $X[i]$ . We define  $X[i..j]$  to be equal to  $X[i] \dots X[j]$  which we call a *substring* of  $X$  if  $i \leq j$  and to the empty string otherwise. If  $j = n$ , we call a substring  $X[i..j]$  a *suffix* of  $X$ .

**Definition 6.3** (Run, Run-length encoding). *A run of a string  $X$  is a maximal substring  $X[i..j]$  such that  $X[i] = X[i + 1] = \dots = X[j]$ . The run-length encoding of a string  $X$ ,  $RLE(X)$  is a sequence obtained from  $X$  by replacing each run with a tuple consisting of the letter forming the run and the length of the run. For example,  $RLE(aabbbc) = (a, 2)(b, 3)(c, 1)$ .*

Let  $d : \Sigma \times \Sigma \rightarrow \mathbb{R}^+$  be a distance function such that for any letters  $a, b \in \Sigma$ ,  $a \neq b$ , we have  $d(a, a) = 0$  and  $d(a, b) > 0$ . The dynamic time warping distance  $\text{DTW}_d(X, Y)$  between strings  $X, Y \in \Sigma^*$  is defined as follows. If both strings are empty,  $\text{DTW}_d(X, Y) = 0$ . If one of the strings is empty, and the other is not, then  $\text{DTW}_d(X, Y) = \infty$ . Otherwise, let  $X = X[1]X[2] \dots X[r]$  and  $Y = Y[1]Y[2] \dots Y[q]$ . Consider an  $r \times q$  grid graph such that each vertex  $(i, j)$  has (at most) three outgoing edges: one going to  $(i + 1, j)$  (if it exists), one to  $(i + 1, j + 1)$  (if it exists), and one to  $(i, j + 1)$  (if it exists). A path  $\pi$  in the graph starting at  $(1, 1)$  and ending at  $(r, q)$  is called a *warping path*, and its *cost* is defined to be  $\sum_{(i,j) \in \pi} d(X[i], Y[j])$ . Finally,  $\text{DTW}_d(X, Y)$  is defined to be the minimum cost of a warping path for  $X, Y$ . Below we omit  $d$  if it is clear from the context.

Let  $M = |P|$ ,  $N = |T|$ , and  $D$  be an  $(M + 1) \times (N + 1)$  table where the rows are indexed from 0 to  $M$ , and the columns from 0 to  $N$  such that:

1. For all  $j \in [0, N]$ ,  $D[0, j] = 0$ ;
2. For all  $i \in [1, M]$ ,  $D[i, 0] = +\infty$ ;
3. For all  $i \in [1, M]$  and  $j \in [1, N]$ ,  $D[i, j]$  equals the smallest DTW distance between  $P[1 \dots i]$  and a suffix of  $T[1 \dots j]$ .

(See Fig. 6.1.) To solve the pattern matching problem under the DTW distance, it suffices to compute the table  $D$ , which can be done in  $\mathcal{O}(MN)$  time via a dynamic programming algorithm, using the following recursion for all  $1 \leq i \leq M, 1 \leq j \leq N$ :

$$D[i, j] = \min\{D[i - 1, j - 1], D[i - 1, j], D[i, j - 1]\} + d(P[i], T[j]) \quad (6.1)$$

In the subsequent sections, we develop more efficient solutions for the low-distance regime on run-length compressible data. We will be processing the table  $D$  by blocks, defined as follows: A subtable  $D[i_p \dots j_p, i_t \dots j_t]$  is called a *block* if  $P[i_p \dots j_p]$  is a run in  $P$  or  $i_p = j_p = 0$ , and  $T[i_t \dots j_t]$  is a run in  $T$  or  $i_t = j_t = 0$ . For  $i_p, i_t > 0$ , a block  $D[i_p \dots j_p, i_t \dots j_t]$  is called *homogeneous* if  $P[i_p] = T[i_t]$ . (For example, a block  $D[3 \dots 4][3 \dots 6]$  in Fig. 6.1 is homogeneous.) A block such that all cells in it contain a value  $q$ , for some fixed integer  $q$ , is called a *q-block*. (For example, a block  $D[5 \dots 5][11 \dots 14]$  in Fig. 6.1 is a 1-block.) The *border* of a block is the set of the cells contained in its top and bottom rows, as well as first and last columns. Consider a cell  $(a, b)$  in  $B$ . We say that a block  $B'$  is the *top neighbor* of  $B$  if it contains  $(a - 1, b)$ , the *left neighbor* if it contains  $(a, b - 1)$ , and the *diagonal neighbor* if it contains  $(a - 1, b - 1)$ .

**Lemma 6.4.** *Consider a block  $B = D[i_p \dots j_p, i_t \dots j_t]$  and cell  $(a, b)$  in it. If  $i_p \leq a < j_p$ , then  $D[a, b] \leq D[a + 1, b]$  and if  $i_t \leq b < j_t$ , then  $D[a, b] \leq D[a, b + 1]$ .*

*Proof.* Let us first give an equivalent statement of the lemma: if  $(a, b)$  and  $(a + 1, b)$  are in the same block, then  $D[a, b] \leq D[a + 1, b]$ , and if  $(a, b)$  and  $(a, b + 1)$  are in the same block, then  $D[a, b] \leq D[a, b + 1]$ .

We show the lemma by induction on  $a + b$ . The base of the induction are the cells such that  $a = 0$  or  $b = 0$ , and for them the statement holds by the definition of  $D$ . Consider now a cell  $(a, b)$ , where  $a, b \geq 1$ . Assume that the induction assumption holds for all cells  $(x, y)$  such that  $x + y < a + b$ . By Equation 6.1, we have:

$$\begin{aligned} D[a, b] &= \min\{D[a - 1, b - 1], D[a - 1, b], D[a, b - 1]\} + d \\ D[a + 1, b] &= \min\{D[a, b - 1], D[a, b], D[a + 1, b - 1]\} + d \\ D[a, b + 1] &= \min\{D[a - 1, b], D[a - 1, b + 1], D[a, b]\} + d \end{aligned}$$

Assume that  $(a, b)$  and  $(a + 1, b)$  are in the same block. We have  $D[a, b] \leq D[a, b - 1] + d$  and trivially  $D[a, b] \leq D[a, b] + d$ . By the induction assumption,  $D[a, b - 1] \leq D[a + 1, b - 1]$  (the cells  $(a, b - 1)$  and  $(a + 1, b - 1)$  must belong to the same block). Therefore,

$$\begin{aligned} D[a + 1, b] &= \min\{D[a, b - 1], D[a, b], D[a + 1, b - 1]\} + d \\ &= \min\{D[a, b - 1] + d, D[a, b] + d, D[a + 1, b - 1] + d\} \\ &\geq \min\{D[a, b], D[a, b], D[a, b - 1] + d\} \\ &\geq \min\{D[a, b], D[a, b], D[a, b]\} = D[a, b]. \end{aligned}$$

Assume now that  $(a, b)$  and  $(a, b + 1)$  are in the same block. We have  $D[a, b] \leq D[a - 1, b] + d$ . Furthermore, as  $(a - 1, b)$  and  $(a - 1, b + 1)$  are in the same block, we have  $D[a - 1, b] \leq D[a - 1, b + 1]$  by the induction assumption. Therefore,

$$\begin{aligned} D[a, b + 1] &= \min\{D[a - 1, b], D[a - 1, b + 1], D[a, b]\} + d \\ &= \min\{D[a - 1, b] + d, D[a - 1, b + 1] + d, D[a, b] + d\} \\ &\geq \min\{D[a - 1, b] + d, D[a - 1, b] + d, D[a, b]\} \\ &\geq \min\{D[a, b], D[a, b], D[a, b]\} = D[a, b]. \end{aligned}$$

This concludes the proof of the lemma.  $\square$

By Equation 6.1, inside a homogeneous block each value is equal to the minimum of its neighbors. Therefore, the values in a row or in a column cannot increase and we have the following corollary:

**Corollary 6.5.** *Each homogeneous block is a  $q$ -block for some value  $q$ .*

### 3 Linear Algorithm for $k = 1$

In this section, we show Lemma 6.1 that for a pattern  $P$  with  $m$  runs and text  $T$  with  $n$  runs gives an  $\mathcal{O}(m + n)$ -time algorithm.

**Definition 6.6** (RLE-diagonals). *We say that a sequence of blocks forms an RLE-diagonal if the blocks are formed by runs  $i, i + 1, \dots, j$  of  $P$  and  $i + \delta, i + 1 + \delta, \dots, j + \delta$  of  $T$ , for some integers  $i, j, \delta$ .*

**Definition 6.7** (Streak). *A  $q$ -streak is a maximal subsequence of an RLE-diagonal containing sequential homogeneous  $q$ -blocks.*

**Observation 6.8.** *If  $D[i, j] = 0$ , then it belongs to a 0-streak. Furthermore, each 0-streak necessarily starts in the first row of  $D$ .*

*Proof.* By definition, there must be a path from the first row of  $D$  to  $D[i, j]$  containing 0-values only. For every 0-value  $D[i', j']$  we must have  $P[i'] = T[j']$ , and therefore every such value must belong to a homogeneous 0-block. Furthermore, two homogeneous blocks can only be neighbours diagonally, else it would contradict the maximality of the runs. The claim follows.  $\square$

**Observation 6.9.** *If  $D[i, j] = 1$ , then  $D[i, j]$  belongs to a 1-streak or neighbours a block in a 0-streak.*

*Proof.* If  $P[i] = T[j]$ , we are in a homogeneous block and  $D[i, j]$  belongs to a 1-streak, and we are done. Otherwise, we have  $P[i] \neq T[j]$  and there is a path  $(i_1, j_1), (i_2, j_2), \dots, (i_q, j_q)$  such that  $i_1 = 1$ ,  $(i_q, j_q) = (i, j)$ , and  $D[i_q, j_q] = \sum_{q'=1}^q d(P[i_{q'}], T[j_{q'}])$ . It follows that for all  $1 \leq q' \leq q - 1$ ,  $d(P[i_{q'}], T[j_{q'}]) = 0$ , and therefore  $D[i_{q'}, j_{q'}]$  must belong to a 0-streak by Observation 6.8.  $\square$

**Lemma 6.1.** *Given run-length encodings of a pattern  $P$  and of a text  $T$  over an alphabet  $\Sigma$  and a distance  $d : \Sigma \times \Sigma \rightarrow \mathbb{Z}^+$ , the 1-DTW problem can be solved in  $\mathcal{O}(m + n)$  time, where  $m$  is the number of runs in  $P$  and  $n$  is the number of runs in  $T$ . The output is given in a compressed form, with a possibility to retrieve each position in constant time.*

*Proof.* For a string  $S$ , define  $\overline{RLE}(S)$  to be a string such that  $\overline{RLE}(S)[i]$  contains the letter forming the  $i$ -th run of  $S$ . We preprocess  $P' = \overline{RLE}(P)$  and  $T' = \overline{RLE}(T)$  in  $\mathcal{O}(m + n)$  time and space to maintain longest common suffix queries in constant time [105]. The input of a longest common suffix query are two positions  $i, j$  of  $P'$  and  $T'$  respectively, and the output is the largest  $\ell$  such that  $P'[i - \ell \dots i] = T'[j - \ell \dots j]$ .

Let  $B_i$ ,  $1 \leq i \leq n$ , be the block of  $D$  formed by the  $m$ -th run in  $P$  and the  $i$ -th run in  $T$ . Using one longest common suffix query for each block  $B_i$ , we find the maximal streak containing it. If this streak reaches the first row of  $D$ , it is a 0-streak and we can fill the last row of  $B_i$  with zeros.

We must now decide which entries in the  $M$ -th row of  $D$  must be filled with one. Consider an entry  $D[M, \ell] \neq 0$  that belongs to a block  $B_i$ .

If  $B_i$  is contained in a streak of length at least one, then for  $D[M, \ell]$  to be equal to one, it must be a 1-streak. Consider the first block in the maximal streak containing  $B_i$ , and let  $c$  be the cell in its top left corner. Because  $c$  can not be equal to zero, it suffices to check whether the value in  $c$  equals one. Consider a path realizing the value of  $c$ . It goes either through the left neighbour  $\ell$  of  $c$ , the top neighbour  $t$  of  $c$ , or the diagonal neighbour  $d$  of  $c$ . Furthermore, the value in  $c$  equals the minimum of the values in  $\ell, d, t$ , and therefore, at least one of these values equals one, and neither of them can belong to a 1-streak. By Observation 6.9, such a cell must neighbour a block in a zero-streak. For each block neighbouring the cells  $\ell, d, t$ , we use one longest common suffix query to decide whether they are contained in a 0-streak. If they are, then the value in  $c$ , and consequently all the values in  $B_i$ , including the values in the last row of  $B_i$ , are equal to one, and we fill them in appropriately.

Suppose now that  $B_i$  does not belong to a streak. For  $D[M, \ell]$  to be equal to one, it must neighbour a block in a 0-streak. Therefore, there can be only one such cell in  $B_i$ , the one in the left bottom corner, and we can decide whether the value in it equals to one in constant time similar to above.  $\square$

## 4 Main Result: $\mathcal{O}(kmn)$ -time Algorithm

In this section, we show Theorem 6.2 that for a pattern  $P$  with  $m$  runs and a text  $T$  with  $n$  runs gives an  $\mathcal{O}(kmn)$ -time algorithm. We start with the following lemma which is a keystone to our result:

**Lemma 6.10.** *For a block  $D[i_p \dots j_p, i_t \dots j_t]$  let  $h = j_p - i_p$ ,  $w = j_t - i_t$ , and  $d = d(P[i_p], T[i_t])$ . We have for every  $i_p < x \leq j_p$ :*

$$D[x, j_t] = \begin{cases} D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d & \text{if } x - i_p \leq w; \\ D[x - w, i_t] + w \cdot d & \text{otherwise.} \end{cases} \quad (6.2)$$

For every  $i_t < y \leq j_t$ :

$$D[j_p, y] = \begin{cases} D[j_p - (y - i_t), i_t] + (y - i_t) \cdot d & \text{if } y - i_t \leq h; \\ D[i_p, y - h] + h \cdot d & \text{otherwise.} \end{cases} \quad (6.3)$$

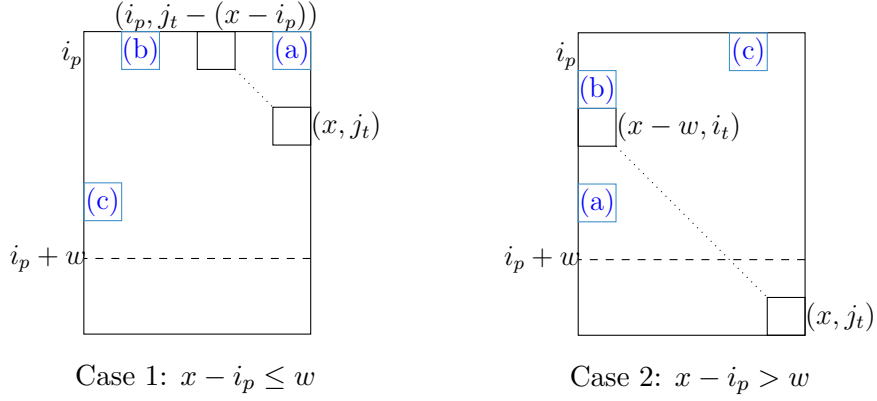


Figure 6.2: Cases of Lemma 6.10. Possible locations of the cell  $(a, b)$  are shown in blue.

*Proof.* For a homogeneous block, we have  $d = 0$ , and by Corollary 6.5 all the values in such a block are equal, hence the claim of the lemma is trivially true.

Assume now  $d > 0$ . Consider  $x$ ,  $i_p < x \leq j_p$ , and let us show Eq. 6.2, Eq. 6.3 can be shown analogously. Let  $\pi$  be a warping path realizing  $D[x, j_t]$ . Let  $(a, b)$  be the first node of  $\pi$  belonging to the block. We have  $a \in [i_p, j_p]$  and  $b \in [i_t, j_t]$  and either  $a = i_p$  or  $b = i_t$ . The number of edges of  $\pi$  in the block from  $(a, b)$  to  $(x, j_t)$  must be minimal, else there would be a shorter path, thus it is equal to  $\max\{x - a, j_t - b\}$  and  $D[x, j_t] = D[a, b] + \max\{x - a, j_t - b\} \cdot d$ .  
Case 1:  $x - i_p \leq w$ . Consider a cell  $(i_p, j_t - (x - i_p))$ . There is a path from  $(i_p, j_t - (x - i_p))$  to  $(x, j_t)$  that takes  $x - i_p$  diagonal steps inside the block, and therefore  $D[x, j_t] \leq D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d$ . We now show that  $D[x, j_t] \geq D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d$ , which implies the claim of the lemma.

- (a) If  $a = i_p$  and  $b \geq j_t - (x - i_p)$ , then  $\max\{x - i_p, j_t - b\} = x - i_p$ . We have  $D[x, j_t] = D[i_p, b] + (x - i_p) \cdot d \geq D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d$  (Lemma 6.4).
- (b) If  $a = i_p$  and  $b < j_t - (x - i_p)$ , then  $\max\{x - i_p, j_t - b\} = j_t - b$ . As there is a path from  $(a, b) = (i_p, b)$  to  $(i_p, j_t - (x - i_p))$  of length  $(j_t - (x - i_p) - b)$ , we have  $D[i_p, j_t - (x - i_p)] \leq D[i_p, b] + (j_t - (x - i_p) - b) \cdot d$ . Consequently,

$$\begin{aligned}
 D[x, j_t] &= D[i_p, b] + (j_t - b) \cdot d \\
 &\geq D[i_p, j_t - (x - i_p)] - (j_t - (x - i_p) - b) \cdot d + (j_t - b) \cdot d \text{ (Eq. 6.1)} \\
 &= D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d
 \end{aligned}$$

- (c) If  $b = i_t$ , then  $i_p \leq a$  and  $\max\{x - a, j_t - b\} \leq \max\{x - i_p, w\} = w$ . As there is a path from  $(i_p, i_t)$  to  $(i_p, j_t - (x - i_p))$  of length  $(j_t - (x - i_p) - i_t)$ , we have  $D[i_p, j_t - (x - i_p)] \leq D[i_p, i_t] + (j_t - (x - i_p) - i_t) \cdot d$ . Therefore,

$$\begin{aligned}
 D[x, j_t] &= D[a, i_t] + w \cdot d \geq D[i_p, i_t] + w \cdot d \text{ (Lemma 6.4)} \\
 &\geq D[i_p, j_t - (x - i_p)] - (j_t - (x - i_p) - i_t) \cdot d + w \cdot d \\
 &= D[i_p, j_t - (x - i_p)] + (x - i_p) \cdot d
 \end{aligned}$$

Case 2:  $x - i_p > w$ . Consider a cell  $(x - w, i_t)$ . There is a path from  $(x - w, i_t)$  to  $(x, j_t)$  that takes  $w$  diagonal steps inside the block, and therefore  $D[x, j_t] \leq D[x - w, i_t] + w \cdot d$ . We now show that  $D[x, j_t] \geq D[x - w, i_t] + w \cdot d$ , which implies the claim of the lemma.

- (a) If  $b = i_t$  and  $a \geq x - w$ , then  $\max\{x - a, j_t - b\} = \max\{x - a, w\} = w$  and we have  $D[x, j_t] = D[a, i_t] + w \cdot d \geq D[x - w, i_t] + w \cdot d$  (Lemma 6.4).



- (b) If  $b = i_t$  and  $a < x - w$ , then  $\max\{x - a, j_t - b\} = \max\{x - a, w\} = x - a$ . As there is a path from  $(a, i_t)$  to  $(x - w, i_t)$  of length  $(x - w - a)$ , we have  $D[x - w, i_t] \leq D[a, i_t] + (x - w - a) \cdot d$  by definition. Therefore,

$$\begin{aligned} D[x, j_t] &= D[a, i_t] + (x - a) \cdot d \\ &\geq D[x - w, i_t] - (x - w - a) \cdot d + (x - a) \cdot d \\ &= D[x - w, i_t] + w \cdot d \end{aligned}$$

- (c) If  $a = i_p$ ,  $b \geq i_t$  and thus  $\max\{x - a, j_t - b\} \leq \max\{x - i_p, w\} = x - i_p$ . Additionally, as there is a path from  $(i_p, i_t)$  to  $(x - w, i_t)$  of length  $(x - w - i_p)$  we have  $D[x - w, i_t] \leq D[i_p, i_t] + (x - w - i_p) \cdot d$ . Consequently,

$$\begin{aligned} D[x, j_t] &= D[i_p, b] + (x - i_p) \cdot d \geq D[i_p, i_t] + (x - i_p) \cdot d \text{ (Lemma 6.4)} \\ &\geq D[x - w, i_t] - (x - w - i_p) \cdot d + (x - i_p) \cdot d \\ &= D[x - w, i_t] + w \cdot d \end{aligned}$$

□

We say that a cell in a border of a block is *interesting* if its value is at most  $k$ . To solve the  $k$ -DTW problem it suffices to compute the values of all interesting cells in the last row of  $D$ . Consider a block  $B = D[i_p \dots j_p, i_t \dots j_t]$  and recall that the values in it are non-decreasing top to down and left to right (Lemma 6.4). We can consider the following compact representation of its interesting cells. For an integer  $\ell$ , define  $q_{\text{top}}^\ell \in [i_t, j_t]$  to be the last position such that  $D[i_p, q_{\text{top}}^\ell] \leq \ell$ , and  $q_{\text{bot}}^\ell \in [i_t, j_t]$  the last position such that  $D[j_p, q_{\text{bot}}^\ell] \leq \ell$ . If a value is not defined, we set it equal to  $i_t - 1$ . Analogously, define  $q_{\text{left}}^\ell \in [i_p, j_p]$  to be the last position such that  $D[q_{\text{left}}^\ell, i_t] \leq \ell$ , and  $q_{\text{right}}^\ell \in [i_p, j_p]$  the last position such that  $D[q_{\text{right}}^\ell, j_t] \leq \ell$ . If a value is not defined, we set it equal to  $i_p - 1$ . Positions  $q_{\text{top}}^0, \dots, q_{\text{top}}^k$  uniquely describe the interesting border cells in the top row of  $B$ ,  $q_{\text{bot}}^0, \dots, q_{\text{bot}}^k$  in the bottom row,  $q_{\text{left}}^0, \dots, q_{\text{left}}^k$  in the leftmost column,  $q_{\text{right}}^0, \dots, q_{\text{right}}^k$  in the rightmost column.

**Lemma 6.11.** *The compact representations of the interesting border cells in the top row and the leftmost column of a block  $B$  can be computed in  $\mathcal{O}(k)$  time given the compact representation of the interesting border cells in its neighbors.*

*Proof.* We explain how to compute the representation for the leftmost column of  $B$ , the representation for the top row is computed analogously. Let  $d = d(P[i_p], T[i_t])$ . If  $d = 0$  (the block is homogeneous), by Corollary 6.5 the block is a  $q$ -block for some value  $q$  which can be computed in  $\mathcal{O}(1)$  time by Equation 6.1 if it is interesting (and otherwise we have a certificate that the value is not interesting). We can then derive the values  $q_{\text{left}}^\ell$ ,  $\ell = 0, 1, \dots, k$  in  $\mathcal{O}(k)$  time.

Assume now  $d > 0$ . We start by computing  $D[i_p, i_t]$  using Equation 6.1. We note that if  $D[i_p, i_t] \leq k$ , then we know the values of its neighbors realizing it and therefore can compute it, otherwise we can certify that  $D[i_p, i_t] > k$ . Assume  $D[i_p, i_t] = v$ , which implies that  $q_{\text{left}}^0, \dots, q_{\text{left}}^{\min\{k, v\}-1}$  equal  $i_p - 1$ . We must now compute  $q_{\text{left}}^{\min\{k, v\}}, \dots, q_{\text{left}}^k$ . Consider a cell  $(q, i_t)$  of the block with  $q > i_p$ . The second to the last cell in the warping path that realizes  $D[q, i_t] = \ell$  is one of the cells  $(q - 1, i_t)$ ,  $(q - 1, i_t - 1)$  or  $(q, i_t - 1)$ , and the value of the path up to there must be  $\ell - d$ . Note that all the three cells belong either to the leftmost column of  $B$ , or the rightmost column of its left neighbor. Consequently, for all  $\min\{k, v\} < \ell \leq k$ , we have  $q_{\text{left}}^\ell = \min\{\max\{q_{\text{left}}^{\ell-d}, r_{\text{right}}^{\ell-d}\} + 1, j_t\}$ , and the positions  $q_{\text{left}}^0, \dots, q_{\text{left}}^k$  can be computed in  $\mathcal{O}(k)$  time. □

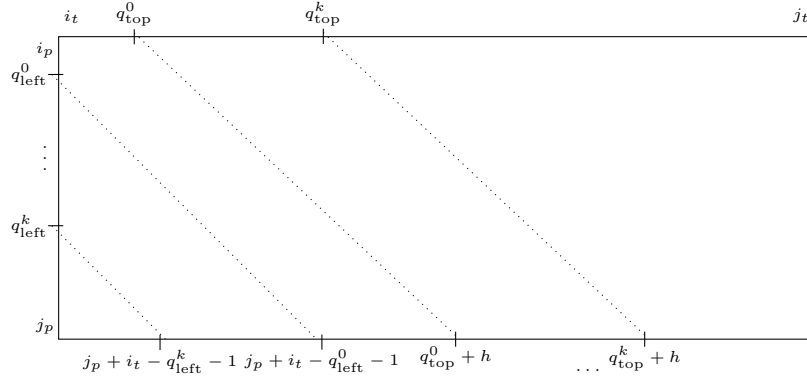


Figure 6.3: Compressed representation of interesting border cells.

**Lemma 6.12.** *The compact representations of the interesting border cells in the bottom row and the rightmost column of a block  $B$  can be computed in  $\mathcal{O}(k)$  time given the compact representation of the interesting border cells in its leftmost column and the top row.*

*Proof.* We explain how to compute the representation for the bottom row, the representation for the rightmost column is computed analogously.

Eq. 6.3 and the compact representations of the leftmost column and the top row of  $B$  partition the bottom row of  $B$  into  $\mathcal{O}(k)$  intervals (some intervals can be empty), and in each interval the values are described either as a constant or as a linear function. (See Fig. 6.3.) Formally, let  $h = j_p - i_p$ . By Eq. 6.3, for  $y \in [i_t, j_p + i_t - q_{\text{left}}^k - 1] \cap [i_t, j_t]$  we have  $D[j_p][y] > k$ . For  $y \in [j_p + i_t - q_{\text{left}}^k, j_p + i_t - q_{\text{left}}^{k-1} - 1] \cap [i_t, j_t]$ ,  $\ell = k, k-1, \dots, 1$ , we have  $D[j_p][y] = \ell + (y - i_t) \cdot d$ . For  $y \in [j_p + i_t - q_{\text{left}}^0, j_p + i_t - i_p] \cap [i_t, j_t]$  we have  $D[j_p][y] = (y - i_t) \cdot d$ . For  $y \in [i_t + h, q_{\text{top}}^0 + h - 1] \cap [i_t, j_t]$  we have  $D[j_p][y] = h \cdot d$ . For  $y \in [q_{\text{top}}^0 + h, q_{\text{top}}^{k-1} + h - 1] \cap [i_t, j_t]$ ,  $\ell = 0, 1, \dots, k-1$ , we have  $D[j_p][y] = \ell + h \cdot d$ . Finally, for  $y \in [q_{\text{top}}^k + h, j_t]$ , there is  $D[j_p][y] > k$  again.

By Lemma 6.4, the values in the bottom row are non-decreasing. We scan the intervals from left to right to compute the values  $q_{\text{bot}}^0, \dots, q_{\text{bot}}^k$  in  $\mathcal{O}(k)$  time. In more detail, let  $q_{\text{bot}}^\ell$  be the last computed value, and  $[i, j]$  be the next interval. We set  $q_{\text{bot}}^{\ell+1} = q_{\text{bot}}^\ell$ . If the values in the interval are constant and larger than  $\ell + 1$ , we continue to computing  $q_{\text{bot}}^{\ell+2}$ . If the values are increasing linearly, we find the position of the last value smaller or equal to  $\ell + 1$ , set  $q_{\text{bot}}^{\ell+1}$  equal to this position, and continue to computing  $q_{\text{bot}}^{\ell+2}$ . Finally, if the values in the interval are constant and equal to  $\ell + 1$ , we update  $q_{\text{bot}}^{\ell+1} = j$  and continue to the next interval. As soon as  $q_{\text{bot}}^k$  is computed, we stop the computation.  $\square$

Since there are  $\mathcal{O}(mn)$  blocks in total, Lemmas 6.11 and 6.12 immediately imply Theorem 6.2.

## 5 Approximation Algorithm

In this section, we show an approximation algorithm for computing the smallest DTW distance between a pattern  $P$  and a substring of a text  $T$ . We assume that the DTW distance is defined over a metric on the alphabet  $\Sigma$ . Kuszmaul [295] showed that the problem of computing the smallest DTW distance over an arbitrary metric can be reduced to the problem of computing the smallest distance over a so-called well-separated tree metric:

**Definition 6.13** (Well-separated tree metric). *Consider a rooted tree  $\tau$  with positive weights on the edges whose leaves form an alphabet  $\Sigma$ . The tree  $\tau$  specifies a metric  $\mu_\tau$  on  $\Sigma$ : The distance*

between two leaves  $a, b \in \Sigma$  is defined as the maximum weight of an edge in the shortest path from  $a$  to  $b$ . The metric  $\mu_\tau$  is a well-separated tree metric if the weights of the edges are not increasing in every root-to-leaf path. The depth of  $\mu_\tau$  is defined to be the depth of  $\tau$ .

Below we show that Theorem 6.2 implies the following result for well-separated tree metrics:

**Lemma 6.14.** *Given run-length encodings of a pattern  $P$  with  $m$  runs and a text  $T$  with  $n$  runs over an alphabet  $\Sigma$ . Assume that the DTW distance is specified by a well-separated tree metric  $\mu_\tau$  on  $\Sigma$  with depth  $h$ , and suppose that the ratio between the largest and the smallest non-zero distances between the letters of  $\Sigma$  is at most exponential in  $L = \max\{|P|, |T|\}$ . For any  $0 < \epsilon < 1$ , there is an  $\mathcal{O}(L^{1-\epsilon} \cdot hmn \log L)$ -time algorithm that computes  $\mathcal{O}(L^\epsilon)$ -approximation of the smallest DTW distance between  $P$  and a substring of  $T$ .*

By plugging the lemma into the framework of [295], we obtain:

**Theorem 6.15.** *approx Given run-length encodings of a pattern  $P$  with  $m$  runs and of a text  $T$  with  $n$  runs over an alphabet  $\Sigma$ . Assume that the DTW distance is specified by a metric  $\mu$  on  $\Sigma$ , and suppose that the ratio between the largest and the smallest non-zero distances between the letters of  $\Sigma$  is at most exponential in  $L = \max\{|P|, |T|\}$ . For any  $0 < \epsilon < 1$ , there is a  $\mathcal{O}(L^{1-\epsilon} \cdot mn \log^3 L)$ -time algorithm that computes  $\mathcal{O}(L^\epsilon)$ -approximation of the smallest DTW distance between  $P$  and a substring of  $T$  correctly with high probability<sup>1</sup>.*

*Proof.* The proof follows the lines of the full version [296] of [295]. Any metric  $\mu$  can be embedded in  $\mathcal{O}(\sigma^2)$  time into a well-separated tree metric  $\mu_\tau$  of depth  $\mathcal{O}(\log \sigma)$  with expected distortion  $\mathcal{O}(\log \sigma)$  (see [89] and [148, Theorem 2.4]). Furthermore, the ratio between the smallest distance and the largest distance grows at most polynomially. Formally, for any two letters  $a, b$  we have  $\mu(a, b) \leq \mu_\tau(a, b)$  and  $\mathbb{E}(\mu_\tau(a, b)) \leq \mathcal{O}(\log \sigma) \cdot d(a, b)$ . Therefore, we have:

$$\text{DTW}_\mu(X, Y) \leq \text{DTW}_{\mu_\tau}(X, Y) \quad (6.4)$$

$$\mathbb{E}(\text{DTW}_{\mu_\tau}(X, Y)) \leq \mathcal{O}(\log \sigma) \cdot \text{DTW}_\mu(X, Y) \quad (6.5)$$

Let  $\delta = \min_{S - \text{ substr. of } T} \text{DTW}_\mu(P, S)$  and  $\delta_\tau = \min_{S - \text{ substr. of } T} \text{DTW}_{\mu_\tau}(P, S)$ . Assume that  $\delta$  is realised on a substring  $X$ , and  $\delta_\tau$  on a substring  $X_\tau$ . By Eq. 6.4, we then obtain:

$$\delta = \text{DTW}_\mu(P, X) \leq \text{DTW}_{\mu_\tau}(P, X_\tau) \leq \delta_\tau$$

And Eq. 6.5 gives the following:

$$\mathbb{E}(\delta_\tau) \leq \mathbb{E}(\text{DTW}_{\mu_\tau}(P, X)) \leq \mathcal{O}(\log \sigma) \cdot \text{DTW}_\mu(P, X) = \mathcal{O}(\log \sigma) \cdot \delta$$

We apply the embedding  $\log L$  times independently to obtain well-separated tree metrics  $\mu_\tau^i$ ,  $i = 1, 2, \dots, \log L$ . From above and by Chernoff bounds,

$$\min_i \min_{S - \text{ substring of } T} \text{DTW}_{\mu_\tau^i}^i(P, S)$$

gives an  $\mathcal{O}(\log \sigma) = \mathcal{O}(\log L)$  approximation of  $\delta$  with high probability and can be computed in time  $\mathcal{O}(L^{1-\epsilon} \cdot mn \log^3 L)$  by Lemma 6.14, concluding the proof of the theorem.  $\square$

<sup>1</sup>The preprocessing time  $\mathcal{O}(|\Sigma|^2 \log L)$  that is required to embed  $\mu$  into a well-separated metric is not accounted for in the runtime of the algorithm.

We now show Lemma 6.14. Compared to [295], the main technical challenge is that our  $k$ -DTW algorithm (Theorem 6.2) assumes an integer-valued distance function on the alphabet. We overcome this by developing an intermediary 2-approximation algorithm for real-valued distances (see the two claims below).

**Proof of Lemma 6.14.** For brevity, let  $\delta$  be the smallest  $\text{DTW}_{\mu_\tau}$  distance between  $P$  and a substring of  $T$ .

**Claim 6.16.** *Let  $0 < \varepsilon < 1$ . Assume that for all  $a, b \in \Sigma$ ,  $a \neq b$ , there is  $\mu_\tau(a, b) \geq \gamma$  and that the value of  $\mu_\tau(a, b)$  can be evaluated in  $\mathcal{O}(t)$  time. There is an  $\mathcal{O}(L^{1-\varepsilon}tmn)$ -time algorithm which either computes a 2-approximation of  $\delta$  or concludes that it is larger than  $\gamma \cdot L^{1-\varepsilon}$ .*

*Proof.* Define a new distance function  $\mu'_\tau(a, b) = \lceil \mu_\tau(a, b) / \gamma \rceil$ . For all  $a, b \in \Sigma$ ,  $a \neq b$ , we have  $\mu_\tau(a, b) \leq \gamma \cdot \mu'_\tau(a, b) \leq \mu_\tau(a, b) + \gamma \leq 2\mu_\tau(a, b)$ . Consequently, for all strings  $X, Y$  we have  $\text{DTW}_{\mu_\tau}(X, Y) \leq \gamma \cdot \text{DTW}_{\mu'_\tau}(X, Y) \leq 2\text{DTW}_{\mu_\tau}(X, Y)$ . Let  $\delta' = \min_{S \text{ substring of } T} \min\{2k + 1, \text{DTW}_{\mu'_\tau}(P, S)\}$  for  $k = L^{1-\varepsilon}$ . By Theorem 6.2, it can be computed in  $\mathcal{O}(L^{1-\varepsilon}tmn)$  time. If  $\delta' = 2L^{1-\varepsilon} + 1$ , we conclude that  $\delta \geq \gamma \cdot L^{1-\varepsilon}$ , and otherwise, output  $\gamma\delta'$ .  $\square$

W.l.o.g., the minimum non-zero distance between two distinct letters of  $\Sigma$  is 1 and the largest distance is some value  $M$ , which is at most exponential in  $L$ . We run the algorithm above for  $\gamma = 1$ , which either computes a 2-approximation of  $\delta$  which we can output immediately, or concludes that  $\delta \geq L^{1-\varepsilon}$ . Below we assume that  $\delta \geq L^{1-\varepsilon}$ .

**Definition 6.17** ( $r$ -simplification). *For a string  $X \in \Sigma^*$  and  $r \geq 1$ , the  $r$ -simplification  $s_r(X)$  is constructed by replacing each letter  $a$  of  $X$  with its highest ancestor  $a'$  in  $\tau$  that can be reached from  $a$  using only edges of weight  $\leq r/4$ .*

**Fact 6.18** (Corollary of [295, Lemma 4.6], see also [281]). *For all  $X, Y \in \Sigma^{\leq L}$ , the following properties hold:*

1.  $\text{DTW}_{\mu_\tau}(s_r(X), s_r(Y)) \leq \text{DTW}_{\mu_\tau}(X, Y)$ .
2. If  $\text{DTW}_{\mu_\tau}(X, Y) > Lr$ , then  $\text{DTW}_{\mu_\tau}(s_r(X), s_r(Y)) > Lr/2$ .

Fix  $r \geq 1$  and  $0 < \varepsilon < 1$ . In the  $(L^\varepsilon, r)$ -DTW gap pattern matching problem, we must output 0 if the smallest DTW distance between  $P$  and a substring of  $T$  is at most  $L^{1-\varepsilon}r/4$  and 1 if it is at least  $Lr$ , otherwise we can output either 0 or 1.

**Claim 6.19.** *The  $(L^\varepsilon, r)$ -DTW gap pattern matching problem can be solved in  $\mathcal{O}(L^{1-\varepsilon} \cdot hmn)$  time.*

*Proof.* Let  $\delta_r$  be the smallest  $\text{DTW}_{\mu_\tau}$  distance between  $s_r(P)$  and a substring of  $s_r(T)$ . If  $L^{1-\varepsilon} > L/2$ , then  $L = \mathcal{O}(1)$  and we can compute  $\delta$  exactly in  $\mathcal{O}(1)$  time by Equation 6.1. Otherwise, we run the 2-approximation algorithm for  $\gamma = r/4$ , which takes  $\mathcal{O}(L^{1-\varepsilon} \cdot hmn)$  time (we can evaluate the distance between two letters in  $\mathcal{O}(h)$  time). If the algorithm concludes that  $\delta_r > L^{1-\varepsilon}r/4$ , then  $\delta > L^{1-\varepsilon}r/4$  by Fact 6.18, and we can output 1. Otherwise, the algorithm outputs a 2-approximation  $\delta'_r$  of  $\delta_r$ , i.e.  $\delta_r \leq \delta'_r \leq 2\delta_r$ . If  $\delta'_r \leq L^{1-\varepsilon}r \leq Lr/2$ , then we have  $\delta_r \leq Lr/2$ . Therefore,  $\delta \leq Lr$  by Fact 6.18 and we can output 0. Otherwise,  $\delta \geq \delta_r \geq \delta'_r/2 > L^{1-\varepsilon}r/2 > L^{1-\varepsilon}r/4$ , and we can output 1.  $\square$

Consider the  $(L^\varepsilon/2, 2^i)$ -DTW gap pattern matching problem for  $0 \leq i \leq \lceil \log ML \rceil$ . If the  $(L^\varepsilon/2, 2^0)$ -DTW gap pattern matching problem returns 0, then we know that  $\delta \leq L$ , and can return  $L^{1-\varepsilon}$  as a  $L^\varepsilon$ -approximation for  $\delta$ . Therefore, it suffices to consider the case where the  $(L^\varepsilon/2, 2^0)$ -DTW gap pattern matching problem returns 1. We can assume, without computing it,

that the  $(L^\varepsilon/2, 2^{\lceil \log ML \rceil})$ -DTW gap pattern matching returns 0 as  $\delta \leq ML$ . Consequently, there must exist  $i^*$  such that  $(L^\varepsilon/2, 2^{i^*-1})$ -DTW gap pattern matching returns 1 and  $(L^\varepsilon/2, 2^{i^*})$ -DTW returns 0. We can find  $i^*$  by a binary search which takes  $\mathcal{O}(L^{1-\varepsilon}hmn \log \log ML) = \mathcal{O}(L^{1-\varepsilon}hmn \log L)$  time. We have  $\delta \geq 2^{i^*-1}L^{1-\varepsilon}/4$  and  $\delta \leq 2^{i^*}L$ , and therefore can return  $2^{i^*-1}L^{1-\varepsilon}/4$  as a  $\mathcal{O}(L^\varepsilon)$ -approximation of  $\delta$ .

## 6 Experiments

This section provides evidence of the advantage of the DTW distance over the edit distance when processing the third generation sequencing (TGS) data. Our experiment compares how the two distances are affected by biological mutation as opposed to sequencing errors, including homopolymer length errors.

We first simulate two genomes,  $G$  and  $G'$ , which can be considered as strings on the alphabet  $\Sigma = \{A, C, G, T\}$ . The genome  $G$  is a substring of the E.coli genome (strain SQ110, NCBI Reference Sequence: NZ\_CP011322.1) of length 10000 (positions 100000 to 110000, excluded). The genome  $G'$  is obtained from  $G$  by simulating biological mutations, where the probabilities are chosen according to [128]. The algorithm initializes  $G'$  as the empty string, and  $\text{pos} = 1$ . While  $\text{pos} \leq |G|$  it executes the following:

1. With probability 0.01, simulate a substitution: chose uniformly at random  $a \in \Sigma$ ,  $a \neq G[\text{pos}]$ . Set  $G' = G'a$  and  $\text{pos} = \text{pos} + 1$ .
2. Else, with probability 0.0005 simulate an insertion or a deletion of a substring of length  $x$ , where  $x$  is chosen uniformly at random from an interval  $[1, \text{max\_len\_ID}]$ , where  $\text{max\_len\_ID}$  is fixed to 10 in the experiments:
  - (a) With probability 0.5, set  $\text{pos} = \text{pos} + x + 1$  (deletion);
  - (b) With probability 0.5, choose a string  $X \in \Sigma^x$  uniformly at random, set  $G' = G'X$  and  $\text{pos} = \text{pos} + 1$  (insertion).
3. Else, set  $G' = G'G[\text{pos}]$  and  $\text{pos} = \text{pos} + 1$ .

To simulate reads, we extract substrings of  $G'$  and add sequencing errors:

1. For each read, extract a substring  $R$  of length 500 at a random position of  $G'$ . As  $G'$  originates from  $G$ , we know the theoretical distance from  $R$  to  $G$ , which we call the “*biological diversity*”. The biological diversity is computed as the sum of the number of letter substitutions, letter insertions, and letter deletions that were applied to the original substring from  $G$  to obtain  $R$ .
2. Add sequencing errors by executing the following for each position  $i$  of  $R$ :
  - (a) With probability 0.001, substitute  $R[i]$  with a letter  $a \in \Sigma$ ,  $a \neq R[i]$ . The letter  $a$  is chosen uniformly at random.
  - (b) If  $R[i] = R[i-1]$ , insert with a probability  $p_{\text{hom}}$  a third occurrence of the same letter to simulate a homopolymer error.

Fig. 6.4 shows the difference between the biological diversity and the smallest edit and DTW distances between a generated read and a substring of  $G$  depending on  $p_{\text{hom}}$ . It can be seen that the DTW distance gives a good estimation of the biological diversity, whereas, as expected, the edit distance is heavily affected by homopolymer errors. To ensure reproducibility of our results, our complete experimental setup is available at <https://github.com/fnareoh/DTW>.

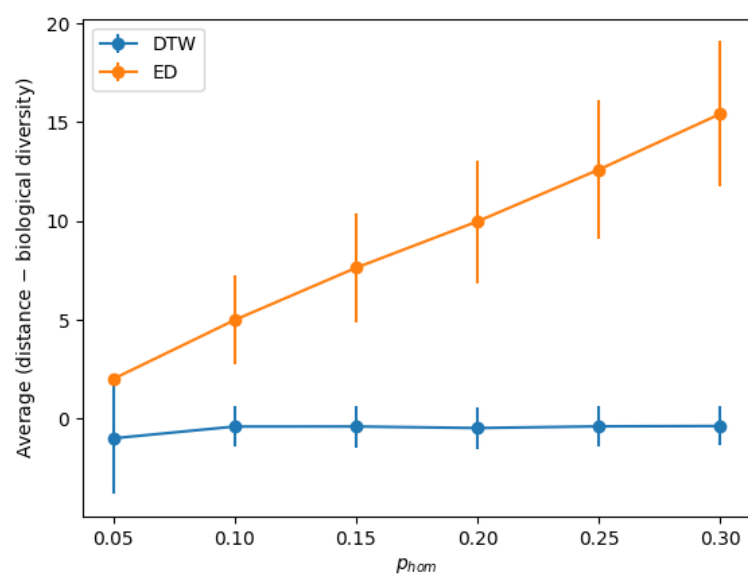


Figure 6.4: Edit and DTW distances offset by the biological diversity as a function of  $p_{hom}$ . Each point is averaged over 600 reads ( $\times 30$  coverage).



## Chapter 7

# Compressing and Indexing Aligned Readset

### Publication

This chapter corresponds to the extended version of the following publication: Travis Gagie, Garance Gourdel, and Giovanni Manzini, “Compressing and Indexing Aligned Readsets”, in: *21<sup>st</sup> International Workshop on Algorithms in Bioinformatics, (WABI 2021), August 2-4, 2021, Virtual Conference*, ed. by Alessandra Carbone and Mohammed El-Kebir, vol. 201, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 13:1–13:21, DOI: 10.4230/LIPIcs.WABI.2021.13.

Compressed full-text indexes are one of the main success stories of bioinformatics data structures but even they struggle to handle some DNA readsets. This may seem surprising since, at least when dealing with short reads from the same individual, the readset will be highly repetitive and, thus, highly compressible. If we are not careful, however, this advantage can be more than offset by two disadvantages: first, since most base pairs are included in at least ten reads each, the uncompressed readset is likely to be at least an order of magnitude larger than the individual’s uncompressed genome; second, these indexes usually pay some space overhead for each string they store, and the total overhead can be substantial when dealing with millions of reads.

The most successful compressed full-text indexes for readsets so far are based on the Extended Burrows-Wheeler Transform (EBWT) and use a sorting heuristic to try to reduce the space overhead per read, but they still treat the reads as separate strings and thus may not take full advantage of the readset’s structure. For example, if we have already assembled an individual’s genome from the readset, then we can usually use it to compress the readset well: e.g., we store the gap-coded list of reads’ starting positions; we store the list of their lengths, which is often highly compressible; and we store information about the sequencing errors, which are rare with short reads. There is nowhere, however, where we can plug an assembled genome into the EBWT.

In this paper we show how to use one or more assembled or partially assembled genome as the basis for a compressed full-text index of its readset. Specifically, we build a labelled tree by taking the assembled genome as a trunk and grafting onto it the reads that align to it, at the starting positions of their alignments. Next, we compute the eXtended Burrows-Wheeler Transform (XBWT) of the resulting labelled tree and build a compressed full-text index on that. Although this index can occasionally return false positives, it is usually much more compact than the alternatives. Following the established practice for datasets with many repetitions, we compare different full-text indices by looking at the number of runs in the transformed strings. For a human Chr19 readset our preliminary experiments show that eliminating separator characters from the EBWT reduces the number of runs by 19%, from 220 million to 178 million, and using the XBWT reduces it by a further 15%, to 150 million.

**Acknowledgement.** Many thanks to Jarno Alanko and Uwe Baier for their XBWT-construction software, and to Diego Díaz, Richard Durbin, Filippo Geraci, Giuseppe Italiano, Ben Langmead, Gonzalo Navarro, Pierre Peterlongo, Nicola Prezza, Giovanna Rosone, Jared Simpson, Jouni Sirén and Jan Studený for helpful discussions.



# 1 Introduction

The FM-index [99] is an important data structure in both combinatorial pattern matching and bioinformatics. Its most important application so far has been in standard short-read aligners — Bowtie [134, 164] and BWA [135] have together over 70 thousand citations and are used every day in clinics and research labs worldwide — but it has myriad other uses and more are still being discovered. Just within computational genomics, FM-indexes have been generalized from single strings to collections of strings for tools such as BEETL [159], RopeBWT [185] and Spring [256], to de Bruijn graphs for tools such as BOSS [156], VARI [248] and Rainbowfish [234], and to graphs for tools such as vg [268]. Recent breakthroughs [313] mean we can now scale FM-indexes to massive but highly repetitive pan-genomic datasets for a new generation of tools [319].

As genomic datasets grow exponentially (from the Human Genome Project to the 1000 Genomes Project and the 100K Genomes Project) and standards for sequencing coverage increase (from less than 10x a few years ago to 30x and 50x now and over 100x for some applications), an obvious question is whether and how the recent breakthroughs in FM-indexing of repetitive datasets can be turned into comparable advances in indexing readsets, so more researchers can efficiently mine them for biomedical insights. For example, extrapolating from previous experiments [319], it should be possible to index both haplotypes from 2705 individuals in less than 100 GB of RAM. In contrast, the readset from the final phase of the 1000 Genomes Project consisted of reads from 2705 individuals and was released as a 464 GB Burrows-Wheeler Transform (BWT) [239], which is beyond the resources of most labs to process. This almost five-fold increase (from 100 to 464 GB) seems reasonable, given the range of lengths and the error rate of short-read sequencing technologies, but those reads were trimmed and error-corrected before their BWT was computed, making that increase harder to justify and thus a target for improvement. Although experimenting with that particular readset is beyond the scope of this paper, since it occupies 87 TB uncompressed, we expect the insights and techniques we develop here will eventually be useful in software able to handle efficiently inputs of that scale.

Recent results on FM-indexing repetitive datasets [313] have shown that the index performance depends on the number of runs in the transformed sequence, where a run is a maximal non-empty unary substring. For example, if the BWT of a dataset of (uncompressed) size  $n$  has  $r$  runs, we can design an FM-index of size  $O(r \log \log n)$  supporting the count and locate operations in optimal linear time. Hence, if a BWT variant produces a transformed string with a smaller number of runs, the resulting index will be smaller and equally fast. The naïve approach to FM-indexing readsets is to concatenate the reads with copies of a separator character between them, and FM-index the resulting single string. However, computing the BWT of such a long string is a challenge and each separator character causes several runs in that BWT. The most competitive indexes for readsets are based on Mantaci et al.’s [117] Extended Burrows Wheeler Transform, which is also easier to build for readsets. The first index for readsets based on the EBWT was BEETL [159], followed by RopeBWT [185]; recently the EBWT has been used also by the Spring compressor [256] specialized for FASTQ reads. BEETL and RopeBWT use explicit separator characters but such characters could be replaced by bitvectors marking positions at the ends of reads.

BEETL and RopeBWT use a heuristic to reduce the number of runs in the EBWT: they conceptually put the separator characters at the ends of reads into the co-lexicographic order (lexicographic order on the reverse string, also referred to as reverse lexicographic order) of the reads, so that the final characters or reads with similar suffixes are grouped together in the EBWT. This often works surprisingly well but in the worst case it cannot make up for the lack of context for sorting those characters into their places in the EBWT. Our proposal in this paper is to graft the reads onto their assembled genome, or a reference genome to which they align

well, and index the resulting labelled tree with Ferragina et al.’s [130] XBWT. To this end we assume that we know how the reads align to the assembled/reference genome: this is not an unreasonable assumption since alignment is the initial step of any readset analysis.

In order to implement our idea we have to overcome a significant hurdle: as the coverage increases so does the amount of raw data produced by a single NGS experiment. Although the high coverage implies that the data is highly compressible, the actual compression process, ie the construction and the compression of the XBWT, must be done partially in externally memory since the input will be usually much larger than the available RAM. Another contribution of the paper is therefore the adaptation of the prefix-free parsing (PFP) technique [280] to the construction of the XBWT. PFP has been proposed for the construction of BWTs of collections of similar genomes: the initial parsing phase is able to compress the input maintaining enough information to compute the BWT working on the compressed representation. In this paper we adapt PFP to readsets, taking care also of the “grafting” of the single reads to the reference/assembled genome. Given a pattern  $P$ , our index could answer  $count(P)$  and  $locate(P)$  queries which report respectively the number of positions where  $P$  occurs and the list of positions where  $P$  occurs. The main drawback to our index, apart from taking one or more assembled or partially assembled genomes as a base, is that it can return a false-positive in the  $count$  operation when an occurrence of a pattern starts in the trunk of an alignment tree and ends in a branch. In other words, the index can report a match that is not completely contained within a read but would be if we padded the read on the left with enough characters copied from just before where it aligns. In a locate operation false-positives could be identified, but this operation is much slower. Even this is not entirely bad, however, and it is conceivable this bug could sometimes be a feature. The analysis of those false positive and the size of the bit vectors marking the end of reads is left as future work.

The rest of the paper is organized as follows. In Section 2 we first describe the BWT and FM-indexes, then the EBWT and XBWT and the concept of Wheeler graph that unifies them. In Section 3 we introduce our idea for indexing aligned readsets with the XBWT and we prove some theoretical results supporting it. In Section 4 we describe how we adapt PFP to indexing readsets, which allows us to experiment with larger files than would otherwise be possible with reasonable resources. In Section 5 we present our experimental results showing that applying the XBWT to index readsets works well in practice as well as in theory. Finally, we outline in Section 6 how our study of storing reads with the XBWT may improve the space usage of the hybrid index [178, 260, 205].

## 2 Concepts

For a better understanding of the problem context, we give a succinct description of the second generation sequencing technique. Most publicly available readsets are from Illumina sequencers [163] which rely on sequencing by synthesis. For this process, millions or billions of single-stranded snippets of DNA called templates are deposited onto a slide and amplified into clusters of clones. In each sequencing cycle we learn one base of each template: we add DNA polymerase and specially terminated bases; the polymerase attaches a terminated base to each strand, complementary to the next base in the strand; we shine a light on the slide and the terminated bases glow various colours; we take a photo and note the colour of each cluster; and finally, we treat the slide to remove the terminators. Sometimes, however, one of the added bases is not correctly terminated, so the polymerase attaches first it and then another base to a strand in some cluster; that strand is then out of step with the rest of the cluster, and the cluster will have a mix of colours in the photos for subsequent sequencing cycles. As we go through more and more sequencing cycles, more strands tend to fall out of step, resulting in

less reliable results. (For further discussion we refer the reader to, e.g., Langmead’s lecture on this topic [209].) This tendency means sequencing by synthesis has an asymmetric error profile, with errors more likely towards the ends of the reads. It follows that sequencing errors tend to be near the end of the reads: our index is designed to take advantage of this feature (see Theorem 7.2).

## 2.1 BWT and FM-index

The Burrows-Wheeler Transform (BWT) [48] of a string  $S$  is a permutation of the characters in  $S$  into the lexicographic order of the suffixes that immediately follow them, considering  $S$  to be cyclic. For example, as shown on the left in Figure 7.1, the BWT of `GATTAGATACAT$` is `TTTCGGAA$AATA`, assuming `$` is a special end-of-string symbol lexicographically smaller than all other characters. Because the BWT groups together characters that precede similar suffixes, it tends to convert global repetitiveness into local homogeneity: e.g., for any string  $\alpha$ , the BWT of  $\alpha^t$  consists of  $|\alpha|$  unary substrings of length  $t$  each; even the BWT in our example has length 13 but consists of only 8 maximal unary substrings (called runs). This property led Burrows and Wheeler to propose the BWT as a pre-processing step for data compression and Seward [53] used it as the basis for the popular `bzip2` compression program.

The BWT is also the basis for the FM-index [99], one of the first and most popular compressed indexes, which is essentially a rank data structure over the BWT combined with a suffix-array sample. The FM-index is an important data structure in combinatorial pattern matching and bioinformatics, and is itself the basis for popular tools such as Bowtie [134, 164] and BWA [135] that align DNA reads to reference genomes. We refer the reader to Navarro’s [227] and Mäkinen et al.’s [211] textbooks for detailed discussions of how FM-indexes are implemented and used for read alignment.

## 2.2 EBWT

Although alignment against one or more reference genomes remains a key task in bioinformatics, there is growing interest in compressed indexing of sets of reads [239, 340]. The FM-index plays a central role here too: Mantaci et al. [117] generalized the BWT to the Extended BWT (EBWT), which applies to collections of strings, and then Cox et al. [167, 158, 181] used an FM-index built on the EBWT in their index BEETL for readsets. The same construction was also used in subsequent indexes for readsets, such as RopeBWT [185] and Spring [256].

The EBWT of a collection of strings is a permutation of the characters in those strings into the lexicographic order of the suffixes that immediately follow them, considering each string to be cyclic. For example, as shown on the right in Figure 7.1, the EBWT of `GATTAGATACAT$`, `TTAGAG$`, `TAGATA$`, `GATAC$` and `ATACAT$` is `TCAAATTGTTTTTCGG$GAAAA$ATAAAT$A$`. When we see the BWT and EBWT as permutations of characters, the BWT of a single string has a single cycle, whereas the EBWT of a collection of strings has a cycle for each string. This means it is easier to build the EBWT and update it when a string is added or deleted, than to build and update the BWT of the concatenation of the collection with the strings separated by copies of a special character. We refer the reader to Egidi et al.’s [284, 312] and Díaz-Domínguez and Navarro’s [332] recent papers for descriptions of efficient construction and updating algorithms.

Despite its benefits, the EBWT sometimes does not take full advantage of its input’s compressibility. In our example, as Figure 7.1 shows, even though all the strings in the collection are substrings of `GATTAGATACAT$` with copies of `$` appended to them, their EBWT has more than twice as many runs as its BWT. As a heuristic for reducing the number of runs, and thus reducing BEETL’s space usage, Cox et al. suggested considering the lexicographic order of the copies

	<i>F</i>	<i>L</i>		<i>F</i>	<i>L</i>		<i>F</i>	<i>L</i>
0	\$GATTAGATACAT		0	\$ATACAT		16	ATTAS	G
1	ACAT\$GATTAGAT		1	\$GATA	C	17	C\$GAT	A
2	AGATACAT\$GATT		2	\$GATT	A	18	CAT\$ATA	
3	AT\$GATTAGATAC		3	\$TAGATA		19	GA\$TT	A
4	ATACAT\$GATTAG		4	\$TTAG	A	20	GATA\$TA	
5	ATTAGATACAT\$G		5	A\$GAT	T	21	GATAC	\$
6	CAT\$GATTAGATA		6	A\$TAGAT		22	GATTA	\$
7	GATACAT\$GATTA		7	A\$TTA	G	23	T\$ATACA	
8	GATTAGATACAT\$		8	AC\$GA	T	24	TA\$GA	T
9	T\$GATTAGATACA		9	ACAT\$AT		25	TA\$TAGA	
10	TACAT\$GATTAGA		10	AGAT\$T	T	26	TAC\$G	A
11	TAGATACAT\$GAT		11	AGATA\$T		27	TACAT\$A	
12	TTAGATACAT\$GA		12	AT\$ATAC		28	TAGA\$	T
			13	ATA\$TAG		29	TAGATA\$	
			14	ATAC\$	G	30	TTA\$G	A
			15	ATACAT\$		31	TTAGA	\$

Figure 7.1: The matrices whose rows are the lexicographically sorted rotations of GATTAGATACAT\$ (left) and of GATTAS\$, TTAGAS\$, TAGATAS\$, GATAC\$ and ATACAT\$ (right). The BWT and EBWT are TTTTCGGAA\$AATA and TCAAATTGTTTTTCGG\$GAAAA\$SATAAAT\$AS with 8 and 19 runs, respectively.

of \$ to be the strings' co-lexicographic order. This does not help in cases such as our example, however, for which the EBWT still has 19 runs even with that ordering. Bentley et al. [309] recently gave a linear-time algorithm to find the ordering of the copies of \$ that minimizes the number of runs, but it has not been implemented and it is unclear whether it is practical for large readsets.

Another way to potentially reduce the number of runs is to remove the copies of \$ entirely, and store an auxiliary ternary vector marking which characters in the EBWT are the first and last characters in the strings. If there are  $t$  strings in the collection with total length  $n$ , then storing this vector takes  $O(t \log(n/t) + t)$  bits (even if some of the strings are empty or consist of only one character). As shown in Figure 7.2, the EBWT becomes TTTTTTGTCTGGGAACAAAAAATTAAAA, with only 10 runs. The idea of replacing \$'s with an auxiliary vector is relatively new since it originates from seeing the EBWT as a special case of Wheeler graphs [242] which are described in the next section.

## 2.3 Wheeler Graphs and XBWT

Wheeler graphs were introduced by Gagie, Manzini and Sirén [242] as a unifying framework for several extensions of the BWT, including the EBWT, Ferragina et al.'s [130] eXtended BWT (XBWT) for labelled trees, Bowe, et al.'s. [156] index (BOSS) for de Bruijn graphs, and Sirén et al.'s [188] Generalized Compressed Suffix Array (GCSA) for variation graphs. A directed edge-labelled graph is a Wheeler graph if there exists a total order on the vertices such that

- vertices with in-degree 0 are earliest in the order;
- if  $(u, v)$  is labelled  $a$  and  $(u', v')$  is labelled  $b$  with  $a < b$ , then  $v < v'$ ;

		<i>F</i>	<i>L</i>			<i>F</i>	<i>L</i>
0	0	ACATAT		14	+	GATA	C
1	0	ACGA	T	15	0	GATATA	
2	0	AGATAT		16	0	GATT	A
3	–	AGAT	T	17	+	GATT	A
4	0	AGAT	T	18	0	TACATA	
5	+	ATACAT		19	0	TACG	A
6	0	ATAC	G	20	+	TAGATA	
7	–	ATAGAT		21	0	TAGA	T
8	0	ATATAC		22	0	TAGA	T
9	0	ATATAG		23	–	TATACA	
10	0	ATTA	G	24	0	TATAGA	
11	–	ATTA	G	25	0	TTAG	A
12	0	CATATA		26	+	TTAG	A
13	–	CGAT	A				

Figure 7.2: The matrix whose rows are the lexicographically sorted rotations of **GATTA**, **TTAGA**, **TAGATA**, **GATAC** and **ATACAT**. The EBWT is **TTTTTTGTCGGAACAAAAATTAAAA** with 10 runs.

- if  $(u, v)$  and  $(u', v')$  are both labelled  $a$  and  $u < u'$  then  $v \leq v'$ .

Figure 7.3 shows an example of a Wheeler graph with a valid order on the vertices. The ordering is obtained by lexicographically sorting the strings spelling the labels in the upward path from each vertex to the root where the ties are broken deterministically (following an arbitrary order on the branches). For example, vertex 0 has upward path  $\varepsilon$ , vertex 3 has upward path **AG**, vertex 30 has upward path **TAG** and so on. Notice that for directed acyclic graphs such as trees, such order on the vertices can be computed quickly with an adaptation of the doubling algorithm [11].

Once we have a valid order, the standard representation of a Wheeler graph is defined considering the vertices in that order and listing the labels on the outgoing edges of each vertex. In addition, for each vertex we represent its out-degree and in-degree in unary thus obtaining two additional binary arrays. For example, for the graph in Figure 7.3 the first five vertices have outgoing edges labelled **GG T T T TT**, so the label array starts with **GGTTTTT**... and the out-degree bit-array starts with **001010101001**.... This simple representation, combined with **rank** and **select** primitives, supports efficient search and navigation operations on Wheeler graphs. We refer the reader to Prezza’s [344] recent survey for a discussion of Wheeler graphs and related results.

Note that the graph in Figure 7.3 is a labelled tree: indeed its Wheeler Graph representation is equivalent to the output of the XBWT [130] applied to the same tree (details in the full paper). For clarity of presentation in the following we will still refer to the EBWT and XBWT even if they are both special cases of Wheeler graphs.

### 3 Our Contribution

Figure 7.3 can be seen as a representation of a “genome” **GATTAGATACAT** and of five “reads” **GATTA**, **TTAGA**, **TAGATA**, **GATAC** and **ATACAT** extracted, without errors, from it. Starting with the vertex with rank 28, corresponding to the last symbol of the “genome”, and navigating the tree

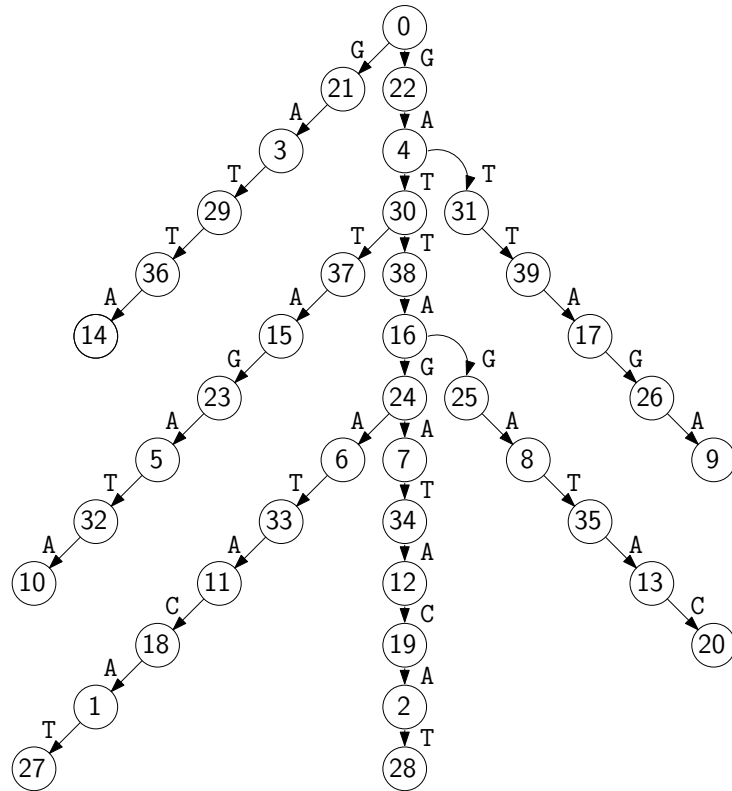


Figure 7.3: A directed, edge-labelled tree whose vertices are labelled to show it is a Wheeler graph. The XBWT is GGTTTTTTTTTTCCCGGGGAAAAAAAAATTTAAAAAAAA with 7 runs.

we are able to recover all the individual strings. Notice however, that the XBWT has only 7 runs while the BWT of the “genome” and the EBWT of the “reads” in Figure 7.1 have 8 and 19 runs, respectively. The EBWT without \$ of the reads alone in Figure 7.2 has 10 runs. We refer the reader to Giuliani et al.’s [337, 291] recent papers for a discussion of the impact of the \$ and of the direction of the string on the number of runs in the BWT. The following theorem shows that the example in Figure 7.3 is not a coincidence: if the “reads” have no errors and they are appended to the reference in the proper positions, then the XBWT has the same number of runs as the BWT of the *reverse* of the “genome”.

**Theorem 7.1.** *Suppose we sample substrings from a string and we form a labelled tree by grafting (appending) the substrings in the same position they were sampled so that all edge labels at the same depth are equal. Then the XBWT of the tree has the same number of runs as the BWT of the reverse of the string.*

*Proof.* Consider the tree shown in Figure 7.3. The tree satisfies the hypothesis of Theorem 7.1 since it was obtained by sampling some substrings from GATTAGATACAT and then grafting them onto it such that all the edge labels at the same depth are equal (so a horizontal line always hits edges with only the same label). Clearly, all the labels at the same depth not only are equal, but they have the same upward-path label, which is the prefix preceding the corresponding character in the string. Since the XBWT is built by sorting labels according to the string spelled by their upward path, we see that each symbol of the original string will be adjacent to all reads symbols at the same horizontal level, and that all such symbols are identical. Finally, observe that also

in the BWT of the reverse of the string symbols are sorted according to the prefix preceding them; hence the XBWT can be obtained by replacing each symbol in the BWT, except the \$, by a run of the same symbol and the thesis follows.  $\square$

Figure 7.3 and Theorem 7.1 suggest a new way to compress and index readsets: graft the reads onto a fully or partially assembled genome, or a reference genome if need be, and store the XBWT of the resulting tree. We note that, although assembly-free indexing is a more general problem, indexing assembled reads is still of practical interest [239]. Many readsets have coverage of 30x or even 50x, which makes them extremely large but should also make run-length compression practical on the XBWTs. If we want to index readsets from several individuals, we can simply graft the reads onto the appropriate assembled genomes and compute the XBWT of the forest, which is also a Wheeler graph.

Theorem 7.1, provides an extremely good estimate of the number of runs of the XBWT, but it holds under the unrealistic assumption that the reads have no errors. However, we can take advantage of the fact that sequencing by synthesis has an asymmetric error profile: errors are much more likely at the end of a read than at the beginning. The following result shows that errors at the end of the reads have a limited impact to the overall number of runs in the XBWT.

**Theorem 7.2.** *In the hypothesis of Theorem 7.1 suppose that the sampled substrings may differ from the reference string and that the average distance from first difference (insertion, deletion, or substitution) to the end of the substring is  $\delta$ . Then, with respect to Theorem 7.1 the XBWT of the tree will have at most  $2\delta$  additional runs per substring.*

*Proof.* Consider a single substring of length  $\ell$  in which the distance between the first difference and the end of the substring is  $d$  (we assume  $d = 0$  if there are no differences). Reasoning as in the proof of Theorem 7.1, we see that the first  $\ell - d$  symbols of the substring will end up in the same run as the corresponding symbol of the reference string (the one at the same depth in the tree). Each of the other  $d$  symbols will, in the worst case, end in the middle of a run of a different symbol thus creating two additional runs. Summing this additional runs over all substrings we get a total number of additional runs upper bounded by  $2\delta$  runs per substring.  $\square$

To guarantee that most of the errors are at the end of the reads, we propose to build two trees: one for the assembled genome and one for its reverse complement. Having two trees means we do not have to reverse and complement half the reads before grafting them onto a single tree: the reversal of the string would be problematic in view of Theorem 7.2 since it would move an error from the end of the read to its front. We can build two trees with a small additional cost since the alignment algorithm will tell us whether each read aligns to the reference or to its reverse complement.

Assuming our scheme guarantees an improvement in compression we want to be sure the resulting index is also efficient. Prezza [343] recently showed how to generalize Gagie, Navarro and Prezza’s [313] results about fast locating from run-length compressed BWTs to run-length compressed XBWTs, at the cost of storing the trees’ shapes, which takes a linear number of bits. For trees with far more internal vertices than leaves, however, it is relatively easy to support fast locating in small space, as a corollary of the following theorem.

**Theorem 7.3.** *Let  $G$  be a Wheeler graph and  $r$  be the number of runs in a Burrows-Wheeler Transform of  $G$ , and suppose  $G$  can be decomposed into  $v$  edge-disjoint directed paths whose internal vertices each have in- and out-degree exactly 1. We can store  $G$  in  $O(r + v)$  space such that later, given a pattern  $P$ , in  $O(|P| \log \log |G|)$  time we can count the vertices of  $G$  reachable by directed paths labelled  $P$ , and then report those vertices in  $O(\log \log |G|)$  time per vertex.*

**Corollary 7.4.** *Let  $T$  be a labelled tree on  $n$  vertices obtained by grafting reads onto their assembled genome as described. Let  $r$  be the number of runs in the XBWT and let  $t$  be the number of reads. We can store  $T$  in  $O(r + t)$  words of space such that later, given a pattern  $P$ , in  $O((|P| + k) \log \log n)$  time we can report all the  $k$  vertices reachable by paths labelled  $P$ .*

We sketch a proof of Theorem 7.3 in 7, although we omit the details because, at least when dealing with short reads, it may be more practical just to descend until we reach a branching node (in which case the pattern is in the assembled genome, not in a read) or a leaf. We have not yet considered carefully whether Nishimoto and Tabei’s [321] faster locating can be applied to improve Theorem 7.3 or Corollary 7.4.

Before we concentrate on optimizations we should consider two basic questions: are our XBWTs for readsets significantly smaller than their EBWTs in practice and, if so, how can we build them efficiently? Theorem 7.2 offers some guarantees of compression, but to test how our idea works in practice in Section 5 we build the XBWT and EBWT for a real, high-coverage readset and see how the numbers of runs in them compare. In Section 4 instead we face the problem of the efficient construction of XBWTs for large datasets.

## 4 XBWT via Prefix Free Parsing

The problem of building the XBWT for a set of reads as described in Section 3 is non trivial because the input typically consists in tens of gigabytes of data and we cannot make use of the available algorithms [303, 307] which are designed to work in RAM. However, the fact that reads are copies (possibly with errors), of portions of a relatively small reference suggests that the overall amount of information content is relatively small. Therefore we decided to compute the XBWT using the technique of Prefix Free Parsing (PFP) that has been successfully utilized for computing the BWT for large collections of genomes from individuals of the same species. Our implementation was done in C++ and is available on [github.com/fnareoh/Big\\_XBWT](https://github.com/fnareoh/Big_XBWT). Note that our algorithm does not take as input a labelled tree, but rather a reference genome and a set of reads aligned to that genome (in the format of a `.bam` file); the alignment implicitly defines a labeled tree as described in Section 3.

In the PFP construction of the BWT the input is parsed into overlapping phrases using context-triggered piecewise hashing [280]. If the input contains many repetitions, the use of context-triggered hashing ensures that the parsing will contain a relatively small number of distinct phrases. The actual construction of the BWT is done using only the dictionary of distinct phrase and the parse (which describes how the dictionary phrases can be used to reconstruct the input). For repetitive datasets the dictionary and the parse fit in RAM even when the original input does not. Unmodified, however, PFP does not work well on readsets since the phrases generated at the beginning and end of each read will likely be unique. As a result, the dictionary will be quite large and the algorithm inefficient. To prevent this, we extend the reads forward and backward so they begin and end with complete phrases. The extension is done using the symbols in the reference immediately before and after the position where the read aligns, so that the phrases are likely to be not unique (if the read has no errors the phrases will be exactly the same generated when parsing the reference). Although this technique maintains the dictionary small, the tricky part is to exclude these extensions when computing the actual XBWT.

Summing up, our implementation is divided in three main phases. In the first phase we partition the reference and the reads into phrases; the set of distinct phrases is called the *dictionary* and the way phrases form the reference and the reads is called the *parse*. We use the extension trick mentioned before, and, if the reference and the reads are similar, the dictionary will be relatively small. In the second phase we compute the XBWT of the parse. Since



phrases are relatively large, the number of symbols in the parse is much smaller than in the original input, so the parse fits in RAM and the computation can be done using a doubling algorithm [11]. Finally, in the third phase we recover the XBWT of the input from the XBWT of the parse. The details of the three phases are given below.

## 4.1 Construction of the Dictionary and the Parse

We start by scanning the reference as in the PFP BWT construction algorithm. The algorithm takes as input parameters a window size  $w$ , and a modulo  $m$ . We slide a window of length  $w$  over the text, at each step computing the Karp-Rabin fingerprint [35] of the window. We define a terminating windows as a window with Karp-Rabin fingerprint equal to zero modulo  $m$ . Terminating windows decompose the text into overlapping phrases: each phrase is a minimal substring that begins and ends with a terminating window. Note that each terminating window is a suffix of the current phrase and the prefix of the next phrase so consecutive phrases have a size- $w$  overlap. Note that defining phrases using terminating windows ensures that no phrase is a prefix (or a suffix) of another phrase, hence the name “prefix free parsing”.

In addition to keeping track of window fingerprints, we also maintain a different hash  $h(p_i)$  of the current phrase  $p_i$ . For simplicity in the following we assume distinct phrases always have distinct hashes, if not we detect it and crash. At the end of this scanning phase, the reference has been parsed into the (overlapping) phrases  $p_1, p_2, \dots, p_z$ . We build a vector  $S[1, z]$  storing for each phrase  $p_i$  its starting position  $s_i$  in the reference and its hash  $h(p_i)$ . We also build as we go the dictionary that associate to each hash value  $h(p_i)$  the corresponding phrase  $p_i$  (stored as a simple string) and  $occ(p_i)$  the number of occurrences of that phrase. We will later also need the length of each phrase but we don’t store it explicitly, just deduce it from the string stored in the dictionary.

After parsing the reference, we process the reads one by one. From the file of aligned reads, we obtain both the read  $r$  as a string and the position  $l$  where the read aligns to the reference. We binary search in  $S$  for the rightmost phrase  $p_s$  that starts before position  $l$  and for the leftmost phrase  $p_e$  that ends after position  $l + |r| - 1$ . Let  $p'_s$  (resp.  $p'_e$ ) denote the prefix (resp. suffix) of  $p_s$  (resp.  $p_e$ ) ending (resp. starting) immediately before (resp. after) position  $l$  (resp.  $l + |r| - 1$ ). We define the extended read  $r_{ext} = p'_s \cdot r \cdot p'_e$  where  $\cdot$  here denotes string concatenation. We slide a window onto  $r_{ext}$ , decomposing it into phrases, as we did for the reference. Since  $r_{ext}$  starts and ends with a terminating window the phrases we add while parsing  $r_{ext}$  still form a prefix-free parsing. However, as we do not want to index the whole  $r_{ext}$  in the final XBWT, for each read we keep track and store to disk the starting and ending position of  $r$  in  $r_{ext}$ .

When processing the reads we continue adding the hashes of the phrases to the end parse, using a special value as separator between reads. If we parse a new phrase, we add it to the dictionary. However, as previously pointed out, the phrases coming from the extended reads are likely to be equal to phrases in the reference so we expect the dictionary not to grow significantly (the dictionary would not grow at all if all the reads were substrings of the reference). From the starting and ending position of the original read in the extended read we deduce for each phrase what characters are part of the original read (the reads without extensions) and we store a starting and ending position for each phrase.

Once all the reads have been processed, we sort the phrases in the dictionary in reverse lexicographic order and we output a new parse where each hash of phrase is replaced by its reverse lexicographic rank, the separator symbol is replaced by the number of phrases plus one. To summarize, at the end of this phase we have produced the following output files:

1. `file.dict`: the dictionary in co-lexicographic order;

2. `file.occ`: the frequency of each phrases;
3. `file.parse`: the parse with each phrase represented by its co-lexicographic rank;
4. `file.limits`: the starting and ending position of the original input (reads without extension) in each phrase.

## 4.2 XBWT of the Parse

The main goal of this phase is to construct the XBWT of the parse, using the co-lexicographic rank as meta-characters. To this end we load the parse on RAM, reconstruct its tree structure, and compute the XBWT of this tree via a doubling algorithm [11]. Then, rather than storing the XBWT as is, we construct an inverted list as this structure will be more appropriate for the next phase. For each phrase  $p_i$  we store the list of XBWT positions where  $p_i$  appears. The size of the inverted list for  $p_i$  is equal to its frequency; since frequencies were computed in the first phase, we can output the inverted list as a plain concatenation of positions.

In this phase we also permute the limits (the starting and ending position in the original input) of each phrase according to their order in the XBWT. This way, in the next phase, with the inverted list, we can easily access the limit of any given phrase in the parse. In this phase, we also compute and write to disk for every phrase, the list of phrases (with multiplicities) that immediately follow in the parse. This list will be used to index the characters that precede a full word. However because we only want to index the characters that are in the original input, we only add it after checking the limits. Finally, because we are not storing special characters to mark the end of a read or of the reference (as they would break runs), we construct a bit vector marking such positions and we permute it according to the XBWT order. To summarize, at the end of this phase we have produced the following output files:

1. `file.dict`: the dictionary of the reversed phrases (from the first phase).
2. `file.occ`: the frequency of each phrases (from the first phase).
3. `file.ilist`: the inverted list of the parse.
4. `file.xbwt_limits`: the limits of the phrases in XBWT order.
5. `file.xbwt_end`: markers of the phrases where a read or reference ends in XBWT order.
6. `file.full_children`: for every word, the list of words that follows it.

## 4.3 Building the Final XBWT

This is the final phase where we compute the XBWT of the reference and of the readset. We start by sorting lexicographically the suffixes of the strings in the dictionary  $D$ . At this stage the dictionary  $D$  contains the phrases reversed, so this is equivalent to sort in reverse lexicographic order the prefixes of all phrases. We ignore the suffixes of length  $\leq w$  as they correspond to the terminating window which also belongs to the previous phrase. The sorting is done by the gSACAK algorithm [247] which computes the SA and LCP array for the set of dictionary phrases. We scan the sorted elements of  $D$ , for  $s$  a proper suffix, there are two cases, all the elements in  $D$  which have  $s$  as a proper suffix have the same preceding character, in this case we add it the correct number of times using the frequency of each phrase. In the other case, we use a heap to merge the inverted list writing the appropriate characters accordingly. Here when writing a character we first check that the suffix length is between the limits and only write it

to file if it does. We also check if the character to be added is the last of its sequence (read or reference), if so output a 1 to signal the end of a sequence, else 0. When finding a suffix  $s'$  that corresponds to an entire phrase, we use the children file to output the character at the start of the following phrase. At the end of this phase we have written to disk a file with the XBWT of the reference and readset as well as a bit vector marking which positions are the last character of a read or a genome. To summarize, in this phase we use `file.dict`, `file.occ`, `file.ilist`, `file.full_children` and `file.xbwt_limits`; all other files can be discarded. We output the XBWT in plain text as `file.bwt` and `file.is_end` is the compressed bit vector marking the end of reads.

## 5 Experiments

In this section we present a first experimental evaluation of our XBWT-based approach for compressing a set of aligned reads and we compare it with the known methods based on the EBWT. We compare ourselves to the EBWT and not other compression tools for aligned readset as our long-term goal is to create an index and not just compression. Recall that our implementation and experimental pipeline is available on [github.com/fnareoh/Big\\_XBWT](https://github.com/fnareoh/Big_XBWT). For simplicity we compare the numbers of runs produced by the different algorithms. The actual compression depends on the algorithm used for encoding the run lengths: preliminary experiments with the  $\gamma$  encoder show that the number of runs is a good proxy for measuring the actual compression. An accurate comparison of the time efficiency is left as a future work: we only compared the number of runs produced by our XBWT with the number of runs produced by the EBWT and some of its variants. Note that our implementation computes the XBWT of the reference genome and the readset (as described in the previous section), while the EBWT and its variants were applied only to the readset. We computed all EBWT variants using ropeBWT2 [185]; in addition to plain EBWT we also tested 2 heuristics that reorder the reads to reduce the number of runs in the EBWT: Spring [256] and reverse lexicographic order (RLO) [158], the latter obtained using the option `-s` in ropeBWT2. Since our XBWT implementation does not use the `$` symbol, for a fair comparison we measured the number of runs with and without the `$` for EBWT, Spring+EBWT and RLO+EBWT (therefore ignoring for all algorithms the extra cost of implicitly encoding the ending position of each string). In our tests, we used the following readsets:

- **E.coli** and **S.aureus** from the single-cell dataset [149], the references used are those linked on the single-cell website<sup>1,2</sup>.
- **R.sphaeroides** We have HiSeq and MiSeq sequencing, raw and trimmed versions of the reads from the GAGE-B dataset [173]. The reference used is the longest contig assembled by MSRCA v1.8.3 [174] as it was the most accurate assembler according to the Gage-b companion paper [173]. We only considered the longest contig because our implementation doesn't handle forests of trees yet.
- **Human Chromosome 19** We used as a reference Chromosome 19 from the CHM1 human assembly [189] and one of the HiSeq 2000 readsets<sup>3</sup> used to compute that assembly, considering only the reads that aligned with the reference.

---

<sup>1</sup>[https://www.ncbi.nlm.nih.gov/nuccore/NC\\_000913](https://www.ncbi.nlm.nih.gov/nuccore/NC_000913)

<sup>2</sup><https://www.ncbi.nlm.nih.gov/nuccore/87125858>

<sup>3</sup>[https://www.ncbi.nlm.nih.gov/sra/SRX966833\[accn\]](https://www.ncbi.nlm.nih.gov/sra/SRX966833[accn])

Dataset	Number of reads	Read length	Coverage	Avg. dist. from the first sequencing err. to the end	Prop. of reads without seq. error	Error rate
E.coli [149]	14139182	100	304×	13	57.30%	0.01%
S.aureus [149]	26654420	100	927×	7	88.79%	0.01%
Human Chr19 [189]	34167479	100	57×	15	71.62%	0.01%
R.sphaeroides [173]						
HiSeq raw	166820	101	46×	27	31.34%	0.04%
HiSeq trimmed	134207	up to 101	37×	6	83.26%	0.01%
MiSeq raw	23102	251	24×	122	0.25%	0.15%
MiSeq trimmed	20046	up to 251	20×	29	63.55%	0.03%

Table 7.1: Statistics on each dataset used in the experiments. Those statistics were computed only on the reads that aligned forward to the reference. We call sequencing error (or simply error) any difference between the genome and the reads. The coverage is simply defined as the total number of base-pairs in the reads compared to the number of base-pairs in the reference. The average distance between the first sequencing error and the end of the read and the end is computed considering that for error less read this distance is 0. Note that this parameter is exactly  $\delta$  in Theorem 7.2.

None of those readsets are aligned, so we used bwa mem [135] to align them to the chosen reference. In this preliminary experiments we discarded the reads that bwa aligned with the reverse-complement of the reference genome. As mentioned in Section 3 our final prototype will build an XBWT of the tree with the reference and of the tree of the reversed-complemented reference. In Table 7.1, we present statistics on the readsets we used: those statistics were computed only on the reads that aligned forward to the reference.

Preliminary experiments, not reported here, show that removing the \$ in the EBWT (all variants) reduces the number of runs between 2.7% and 29.2%. Consequently, we focus our analysis on the comparison of Plain (no read reordering) EBWT (without dollars), SPRING+EBWT (without dollars), RLO+EBWT, with and without \$ and XBWT.

The results of this comparison are reported in Figures 7.4a and 7.4b. They show that in general the plain EBWT performs worse followed by the SPRING reordering, RLO ordering with dollars then RLO ordering without dollars and finally XBWT performs best. XBWT yields a smaller number of runs than RLO+EBWT (with or without \$) on all datasets, although the number is comparable on some datasets this is still a significant improvement considering that RLO+EBWT already has far less run than the EBWT baseline. On the Chr19 dataset, using RLO+EBWT-no-\$ over plain BWT-no-\$ (not reported in Figure 7.4a) reduced the number of runs by 49%; using the XBWT reduced the number of runs by an additional 16%. On S.aureus and E.coli the reduction between RLO+EBWT-no-\$ and XBWT is of only 3% and 8% respectively.

The R.sphaeroides datasets are especially interesting as they involve two NGS technologies that generate reads of different lengths, different coverages, and with different error profiles. We can first notice that our method brings greater benefits on the HiSeq sequencing which has smaller reads with less errors that are located towards the end of the string. This is an experimental validation of the statement of Theorem 7.2. We can also observe the effect of trimming the reads on the number of runs. On the HiSeq sequencing, trimming reduces the coverage only from 46x to 37x but yields a reduction in the number of XBWT runs by 86%. Note that, as a result, on HiSeq trimmed, the number of XBWT runs is less than half the

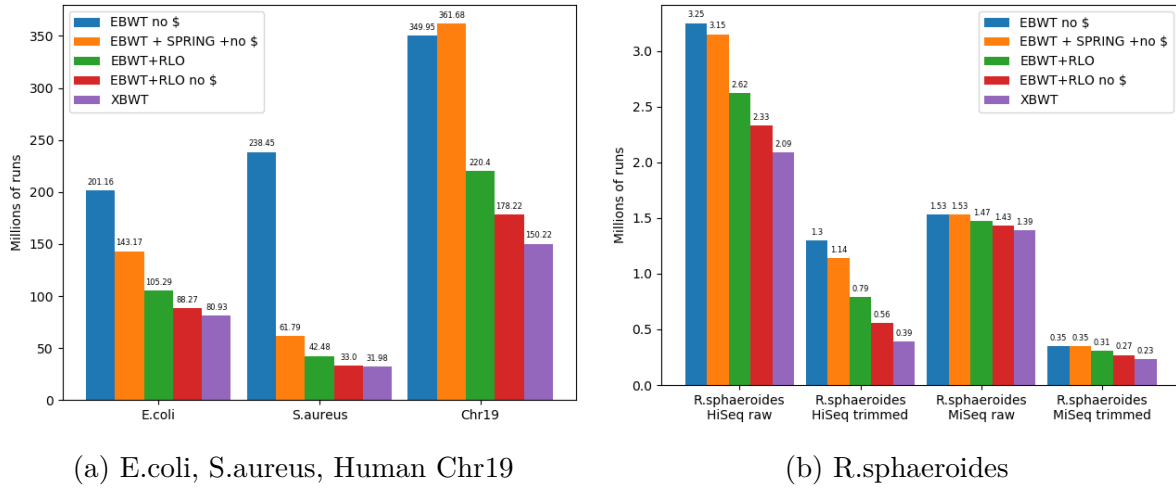


Figure 7.4: Comparison of run-lengths compression by RLO+EBWT with and without \$ and XBWT on various species (7.4a) and on two sequencing of *R.sphaeroides* (HiSeq and MiSeq) and for reads both raw and trimmed (7.4b).

number of runs in plain RLO+EBWT.

## 6 Application to the JST

From a certain angle, Figure 7.3 is reminiscent of Figure 7.5, from Rahn, Weese and Reinert's [187] paper on their Journaled String Tree (JST). This raises the question of whether the XBWT and JST can be used to improve the space usage of the hybrid index [178, 205, 260] and eventually the PanVC [278] pan-genomic read aligner, which is based on the hybrid index.

Figure 7.5 shows a JST supporting search for patterns of length up to 4 in four aligned sequences: the reference

$$r = \text{TAGCGTAGCAGCTATGAGGAGGACCGAGTT}$$

and three others,

$$\begin{aligned} s^1 &= \text{TAGCGTAGCAGCGAGGAGCGACCGAGTT}, \\ s^2 &= \text{TAGCGTGGCAGCGAGGAGCACCGAGTT}, \\ s^3 &= \text{TAGCGTGGCAGCTATGAGGAGCACCGAGTT}. \end{aligned}$$

The straight branch of the tree running along the bottom of the figure is labelled with  $r$ , and the other branches indicate places where the other sequences differ from  $r$ . The other branches end just before a window of size 4 sliding over their sequences matches an aligned window of size 4 sliding over  $r$ . For example, the first branch ends at position 9 because a sliding window of length 4 over positions 7 to 10 of sequences  $s^2$  and  $s^3$  (that is, containing the characters in columns 7 to 10 and the rows for  $s^2$  and  $s^3$  in the alignment shown at the top right in the figure), matches a sliding window of length 4 over positions 7 to 10 in  $r$  (that is, containing the characters in columns 7 to 10 and the row for  $r$  in the alignment).

Suppose we are looking for the pattern  $p = \text{AGCG}$ : considering the circle at the left as the root,  $p$  occurs 3 times as a substring (marked in orange) of root-to-leaf paths, and we can find those occurrences using a depth-first traversal of the tree. Since the sequences are similar, such

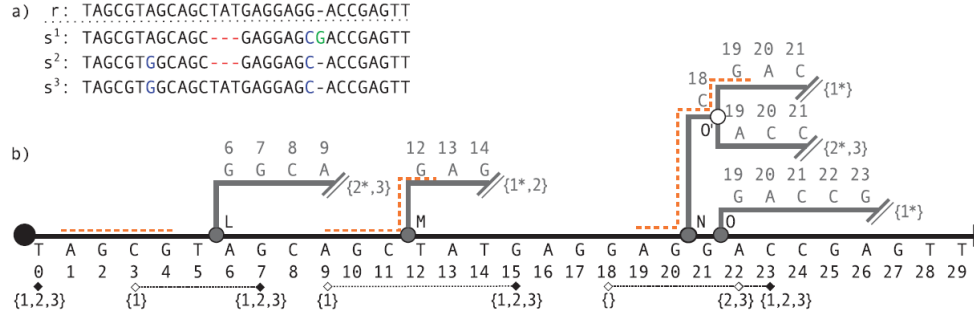


Figure 7.5: An illustration of a JST [187]

a traversal is faster than running a sliding window over each sequence separately. If we find an occurrence of  $p$  in the tree that ends at a node not in the branch for  $r$ , then we have found occurrences in each of the sequences labelling the leaves in that node's subtree. If we find an occurrence of  $p$  in the branch for  $r$ , then we have found occurrences in  $r$  and possibly other sequences. Unfortunately this case is not illustrated in the figure, but if we were looking for **G**TAG then the occurrence at position 4 in  $r$  would have a corresponding occurrence in  $s^1$  but not in  $s^2$  or  $s^3$ ; this is shown by the dashed line between 3 and 7, with  $\{1\}$  at the left end indicating that  $s^1$  matches  $r$  between 3 and 7 and  $\{1, 2, 3\}$  indicating that  $s^1, s^2$  and  $s^3$  all match  $r$  from 7 onward (until the next such interval starts at 9).

The hybrid index is conceptually similar to the JST, but the former is an index and the latter performs pattern matching by scanning the tree sequentially. To build the hybrid index supporting search for patterns of length up to 4 in  $r, s^1, s^2, s^3$ , we first build a string kernel consisting of  $r$  and substrings from  $s^1, s^2, s^3$  that contain all the characters within distance 3 of variations from  $r$ , all separated by copies of a special symbol  $\$$ :

TAGCGTAGCAGCTATGAGGAGGACCGAGTT\$CGTGGCA\$AGCGAG\$GAGCGACC\$GAGCACC.

Any substring of length at most 4 of the the four sequences  $r, s^1, s^2, s^3$  is a substring of the string kernel, and any substring of length at most 4 of the string kernel that does not include a copy of  $\$$  is a substring of at least one of those sequences. We then build an FM-index for the string kernel, with auxiliary data structure that allow us to quickly map occurrences of a pattern in the string kernel to occurrences in the sequences.

It seems interesting that the string kernel for the four sequences in Figure 7.5 has more characters than the JST: on top of  $r$ , the string kernel has a substring **\$**CGTGGCA and the JST has a branch labelled **GGCA**; the string kernel has **\$**AGCGAG and the JST has **GAG**; the string kernel has **\$**GAGCGACC and the JST has **CGAC** and **GACCG** (a tie in this one case); the string kernel has **\$**GAGCACC and the JST has **CACC** (with the first **C** shared with the branch ending **CGAC**). This difference is because the string kernel stores copies of the characters both before and after variation sites, whereas the JST stores copies only of the characters after them. If we build an index using the XBWT of the JST, therefore, it may be smaller than the hybrid index while having the same basic functionality. We leave exploring this possibility as future work.

## 7 Proof Sketch for Theorem 7.3

Let  $G$  be a Wheeler graph with the vertices sorted according to the permutation  $\pi$ . A Burrows-Wheeler Transform (BWT) of  $G$  according to  $\pi$  is a permutation of  $G$ 's edge labels such that, for any pair of edges  $e = (u, v)$  and  $e' = (u', v')$  labelled  $a$  and  $a'$  respectively, if  $u < u'$  then  $a$  precedes  $a'$  in that permutation. For convenience, we assume that the labels of each vertex's out-edges appear in the order in  $\pi$  of their destinations. Notice there may be many BWTs for  $G$  because it may have many permutations  $\pi$  satisfying the Wheeler graph conditions.

Let  $B$  be a BWT of  $G$  according to  $\pi$ . By the definition of a Wheeler graph, for any pattern  $P$  over the alphabet of edge labels, the vertices reachable by directed paths labelled  $P$  form an interval in  $\pi$ . Moreover, if we store a rank data structure for  $B$  and partial sum data structures for the frequencies of the distinct edge labels and the vertices' in- and out-degrees, then given  $P$  we can find its interval in  $O(|P| \log \log |G|)$  time. Let  $r$  is the number of runs (i.e., maximal non-empty unary substrings) in  $B$  and suppose  $G$  can be decomposed into  $v$  edge-disjoint directed paths whose internal vertices each have in- and out-degree exactly 1. Then these data structures take a total of  $O(r + v)$  space, measured in words.

Let  $D$  be a such decomposition of  $G$  and  $n$  be the number of vertices in  $G$ , and assume the vertices are assigned numeric identifiers from 0 to  $n-1$  such that if  $(u, v)$  is an edge and neither  $u$  nor  $v$  is an endpoint of a path in  $D$ , and  $u$  has identifier  $i$ , then  $v$  has identifier  $i+1$ . Notice these identifiers are not necessarily the vertices' ranks in  $\pi$ . For convenience, we assume that even though  $G$  is a multigraph, the number of edges is polynomial in  $n$ , so  $\log \log |G| = O(\log \log n)$ . We show how, still using  $O(r + v)$  space, after we have found the interval for  $P$  we can then report the vertices in it using  $O(\log \log n)$  time for each one.

We first prove a generalization of Bannai, Gagie and I's version [308] of Policriti and Prezza's Toehold Lemma [276], that lets us report the last vertex in the interval for  $P$ . We then define a generalization of Kärkkäinen, Manzini and Puglisi's  $\phi$  function [133], that maps each vertex's identifier to the identifier of its predecessor in  $\pi$ . Finally, we give a generalization of a key lemma behind Gagie, Navarro and Prezza's  $r$ -index [313], that lets us compute our generalized  $\phi$  function with  $O(r+v)$ -space data structures. Combined, these three results yield a generalized  $r$ -index for Wheeler graphs.

### 7.1 Generalized Toehold Lemma

For any pattern  $P[0..m-1]$ , the interval for the empty suffix  $P[m..m-1]$  of  $P$  is all of  $\pi$ , because every vertex is reachable by an empty path. Assume we have found the interval  $\pi[s_{i+1}, e_{i+1}]$  for  $P[i+1..m-1]$  and now we want to find the interval  $\pi[s_i, e_i]$  for  $P[i..m-1]$ . With the partial sum data structure for the vertices' out-degrees, in  $O(\log \log n)$  time we can find the interval in  $B$  containing the labels of the edges leaving the vertices in  $\pi[s_{i+1}, e_{i+1}]$ .

By the definition of a Wheeler graph, the edges labelled with the first and last occurrences of  $P[i]$  in that interval in  $B$ , lead to the first and last vertices in the interval  $\pi[s_i, e_i]$  for  $P[i..m-1]$ . Using the partial sum data structures for the frequencies of the distinct edge labels and the vertices in-degrees, in  $O(\log \log n)$  time we can find the ranks  $s_i$  and  $e_i$  in  $\pi$  of those first and last vertices in  $\pi[s_i, e_i]$ . It follows that in  $O(\log \log n)$  time we can find  $\pi[s_i, e_i]$  from  $\pi[s_{i+1}, e_{i+1}]$ ; therefore, by induction, we can find the interval for  $P$  in  $O(|P| \log \log n)$  time. We can count the vertices in that interval in the same asymptotic time by simply returning the size of the interval.

To be able to find the identifier of the last vertex in the interval for  $P$ , for each edge  $(u, v)$  we store  $u$ 's and  $v$ 's identifiers if any of the following conditions hold:

- $(u, v)$ 's label  $a$  is the last label in a run in  $B$ ;
- either  $u$  or  $v$  is an endpoint of a path in  $D$ ;

- the vertex that follows  $u$  in  $\pi$  has out-degree 0.

We store a select data structure for  $B$ , a bitvector marking the labels  $a$  in  $B$  for whose edges  $(u, v)$  we have  $u$ 's and  $v$ 's identifiers stored, and a hash table mapping the position in  $B$  of each marked label  $a$  to the identifiers of its edge's endpoints. This again takes a total of  $O(r + v)$  space.

By querying the rank data structure, the select data structure, the bitvector and the hash table in that order, we can find the identifier of the vertex reached by the edge labelled by the last copy of  $P[m - 1]$  in  $B$ . By the definition of a Wheeler graph, this is the last vertex in the interval  $\pi[s_{m-1}, e_{m-1}]$  for  $P[m - 1]$ . Assume we have found the interval  $\pi[s_{i+1}, e_{i+1}]$  for  $P[i + 1..m - 1]$  and the identifier of the last vertex  $u$  in that interval, and now we want to find the interval  $\pi[s_i, e_i]$  for  $P[i..m - 1]$  and the identifier of the last vertex  $v$  in that interval. We can find  $\pi[s_i, e_i]$  as described above, so we need only say how to find  $v$ 's identifier.

With the partial sum data structure on the vertices' out-degree and the rank data structure, in  $O(\log \log n)$  time we can check whether  $u$  has an outgoing edge labelled  $P[i]$ . If it does then, of all its out-edges labelled  $P[i]$ , the one whose label appears last in  $B$  goes to  $v$ . By our assumption of how the vertices are assigned their identifiers, if neither  $u$  nor  $v$  are endpoints of a path in  $D$ , then  $v$ 's identifier is  $u$ 's identifier plus 1. If either  $u$  or  $v$  is an endpoint of a path in  $D$ , then we have  $v$ 's identifier stored and we can use the hash table to find it from the position in  $B$  of the last label  $P[i]$  on one of  $u$ 's out-edges, again in  $O(\log \log n)$  time.

If  $u$  does not have an outgoing edge labelled  $P[i]$  then we can use the rank data structure to find the last copy of  $P[i]$  in  $B$  that labels an edge leaving a vertex in  $\pi[s_{i+1}, e_{i+1}]$ . By the definition of a Wheeler graph, this edge  $(u', v)$  goes to  $v$ . Unlike in a BWT of a string, however, its label may not be the end of a run in  $B$ :  $u$  could have out-degree 0,  $u'$  could immediately precede  $u$  in  $\pi$  and the last of its outgoing edges' labels in  $B$  could be a copy of  $P[i]$ , and the first label in  $B$  of an outgoing edge of the successor of  $u$  in  $\pi$  could also be a copy of  $P[i]$ . This is why we store  $v$ 's identifier if the vertex that follows  $u'$  in  $\pi$  has out-degree 0. If  $(u', v)$ 's label is the end of a run in  $B$ , of course, then we also have  $v$ 's identifier stored. In both cases we use  $O(\log \log n)$  time, so from the interval  $\pi[s_{i+1}, e_{i+1}]$  for  $P[i + 1..m - 1]$  and the identifier of the last vertex  $u$  in that interval, in  $O(\log \log n)$  time we can compute the interval  $\pi[s_i, e_i]$  for  $P[i..m - 1]$  and the identifier of the last vertex  $v$  in that interval. Therefore, by induction, in  $O(|P| \log \log n)$  time we can find the interval for  $P$  and the identifier of the last vertex in that interval.

**Lemma 7.5.** *We can store  $G$  in  $O(r + v)$  space such that in  $O(|P| \log \log n)$  time we can find the interval for  $P$  and identifier of the last vertex in that interval.*

## 7.2 Generalized $\phi$

For a string  $S$ , the function  $\phi$  takes a position  $i$  in  $S$  and returns the starting position of the suffix of  $S$  that immediately precedes  $S[i..|S| - 1]$  in the lexicographic order of the suffixes. In other words,  $\phi$  takes the value in some cell of suffix array of  $S$  and returns the value in the preceding cell. Given a pattern  $P$ , if we can find the interval of the suffix array containing the starting positions of occurrences of  $P$  in  $S$ , and the entry in the last cell in that interval, then by iteratively applying  $\phi$  we can report the starting positions of all the occurrences of  $P$ . This is the idea behind the  $r$ -index for strings, which uses a lemma saying it takes only space proportional to the number of runs in the BWT of  $S$  to store data structures that let us evaluate  $\phi$  in  $O(\log \log |S|)$  time.

We generalize  $\phi$  to Wheeler graphs by redefining it such that it takes the identifier of some vertex  $u$  in  $G$  and returns the identifier of the vertex that immediately precedes  $u$  in  $\pi$ . (For our



purposes here, it is not important how  $\phi$  behaves when given the identifier of the first vertex in  $\pi$ .) Given a pattern  $P$ , if we can find the interval in  $\pi$  containing the vertices in  $G$  reachable by directed paths labelled  $P$ , and the identifier of the last vertex in that interval, then by iteratively applying  $\phi$  we can report the identifiers of all those vertices.

Let  $J$  be the set that contains  $u$ 's identifier if and only if any of the following conditions hold:

- $u$  has out-degree not exactly 1;
- $u$  has a single outgoing edge  $(u, v)$  but  $v$  has in-degree not exactly 1;
- the predecessor  $u'$  of  $u$  in  $\pi$  has out-degree not exactly 1;
- $u'$  has a single outgoing edge  $(u', v')$  but  $v'$  has in-degree not exactly 1;
- the edges  $(u, v)$  and  $(u', v')$  have different labels.

We store a successor data structure for  $J$  and, if  $u$ 's identifier is in  $J$ , then we store with it as satellite data the identifier of  $u$ 's predecessor  $u'$  in  $\pi$ . Notice  $u$ 's identifier is in  $J$  only if at least one of  $u$  or  $u'$  or  $v$  or  $v'$  is the endpoint of a path in  $D$ , or the label of  $(u', v')$  is the the last in a run in  $B$  and the label of  $(u, v)$  is the first in the next run. It follows that we can use  $O(r + v)$  space for the successor data structure and have it support queries in  $O(\log \log n)$  time.

Suppose we know the identifier of some vertex  $u$  with identifier  $i$  that is immediately preceded by  $u'$  in  $\pi$  with identifier  $i'$ . If  $u \in J$  then we have  $i'$  stored as satellite data with  $\succ(i) = i$ . If  $u \notin J$ , then  $u$  has a single outgoing edge  $(u, v)$  and  $u'$  has a single outgoing edge  $(u', v')$  with the same label, say  $a$ , and  $v$  and  $v'$  each have in-degree exactly 1. By our assumption on how the identifiers are assigned, the identifiers of  $v$  and  $v'$  are  $i + 1$  and  $i' + 1$  and, by the definition of a Wheeler graph,  $v$  is immediately preceded by  $v'$  in  $\pi$ . It follows that if  $i + \ell$  is the successor of  $i$  then it has stored with it as satellite data  $i' + \ell$ , and so we can compute  $\ell$  and then  $i'$  in  $O(\log \log n)$  time.

**Lemma 7.6.** *We can store  $G$  in  $O(r + v)$  space such that we can evaluate  $\phi$  in  $O(\log \log n)$  time.*

### 7.3 Discussion

Combining Lemmas 7.5 and 7.6, we generalize, we obtain Theorem 7.3. Since  $v = 1$  for a single string labelling a simple path or cycle, Theorem 7.3 gives the same  $O(r)$  space bound and  $O(|P| + k \log \log n)$  time bound we achieve with the  $r$ -index for strings, where  $k$  is the number of occurrences. Nishimoto and Tabei [321] recently improved the query time of the  $r$ -index for strings to  $O(P + k \log \log n)$  — or optimal  $O(P + k)$  for polylogarithmic alphabets — without changing the space bound, and we conjecture this is achievable also for  $r$ -indexes for Wheeler graphs.

# Conclusion

---

In this thesis, we presented the need for more general string queries than classical pattern matching, and the scalability challenges that come from the large productions and archivals of data. In Sections 2 and 3, we detailed the sketch-based approach common to all contributions. The use of sketches enabled processing and storing data efficiently to yield better time and space complexities. The sketching techniques we use are diverse and are not immediate to apply to other problems, but my personal takeaway is the importance of thinking in terms of the key characteristic of the input for a given query. I find it very helpful in algorithmic design both for theoretical studies and applied projects.

Let me recall briefly our contributions and the open questions they leave. In Chapter 1, we gave streaming algorithms for solving the regular expression pattern matching and membership problems. For a regular expression  $R$  of size  $m$  in a stream of size  $n$ , our algorithms run in  $\mathcal{O}(d^3 \text{polylog } n)$  space and  $\mathcal{O}(nd^5 \text{polylog } n)$  time per character where  $d$  is the number of  $|$  and  $*$  symbols in  $R$ . Is it possible to achieve  $\text{poly}(d, \log n)$  space and time per character?

Chapters 2 and 3 studied gapped consecutive occurrences, where given a text  $T$ , patterns  $P_1$  and  $P_2$  and an interval  $[a, b]$ , the task is to find in  $T$  all consecutive occurrence of  $P_1$  and  $P_2$  separated by a distance  $d \in [a, b]$ . In both chapters, the text  $T$  is given as a straight-line program (a grammar generating a single string) of size  $g$ , but Chapter 2 focuses on indexing, while Chapter 3 studies pattern matching. In Chapter 2, we gave two indexes: an index for reporting consecutive occurrences without constraints on the distance, using  $\mathcal{O}(g^2 \log^4 |T|)$  space, and an index for reporting consecutive occurrences at distance within  $[0, b]$  taking  $\mathcal{O}(g^5 \log^5(|T|))$  space. The query time is  $\tilde{\mathcal{O}}(|P_1| + |P_2| + \text{occ})$  for both indexes. Can we improve the space complexity of our indexes, in particular, is space  $\tilde{\mathcal{O}}(g)$  achievable with the same time complexity for consecutive matching (without distance constraints)? Is there an efficient index for the general case where we search for consecutive occurrences separated by a distance in an interval  $[a, b]$ ?

In Chapter 3 we addressed pattern matching where the pattern and the text are processed simultaneously. We showed how to report all consecutive occurrences in  $\mathcal{O}(g + |P_1| + |P_2| + \text{occ})$  optimal time and how we can filter the consecutive occurrences to report those at distance in  $[a, b]$  in the same complexity.

The last contribution of Part I (Chapter 4), was a deterministic algorithm for reporting runs in  $\mathcal{O}(n \log \sigma)$  time for unordered alphabet (where the only operation allowed on characters is equality testing) and we showed a matching lower bound for deterministic algorithms. It remains open whether the lower bound of  $\Omega(n \log \sigma)$  comparisons for square testing holds for randomized algorithms.

Chapter 5 focused on the **LCS with Approximately  $k$  Mismatches** problem, where for a constant  $\varepsilon > 0$  and given two strings  $X$  and  $Y$  and an integer  $k$ , we must return a substring of  $X$  of length at least  $\text{LCS}_k(X, Y)$  that occurs in  $Y$  with at most  $(1 + \varepsilon) \cdot k$  mismatches. We provided two algorithms: one assuming a constant size alphabet running in  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  time and space, and one in  $\mathcal{O}(n^{1+1/(1+\varepsilon)} \log^3 n)$  time and linear space without constraints on the alphabet. We also confirmed the practicality of the second algorithm by an experimental evaluation. As future work, it would be interesting to implement our  $\mathcal{O}(n^{1+1/(1+2\varepsilon)+o(1)})$  time and space solution using an implementation of Approximate Nearest Neighbour data structure such as [377] and add it to the practical evaluation.

Next, we studied pattern matching for the Dynamic Time Warping (DTW) distance. In Chapter 6, for a run-length compressed pattern  $P$  with  $m$  runs and a run-length compressed text  $T$  with  $n$  runs and an integer  $k$ , we provided an  $\mathcal{O}(knm)$ -time algorithm that computes

---

all locations  $j$  where the DTW distance between  $P$  and a suffix of  $T[..j]$  is at most  $k$ . It remains open whether it is possible to improve to  $\mathcal{O}(k(n + m))$  time. Additionally, we detailed a possible practical application of DTW for third-generation sequencing alignment through minimal experiments. It would be interesting to investigate further, unfortunately, it seems difficult due to many tools using an alignment based on the edit distance under the hood.

Finally, in Chapter 7, we studied the applicative problem of read indexing. We proposed using the read's alignment to give additional context and to achieve better overall compression. Our index is based on the XBWT, and we provided a main memory-efficient construction using prefix-free parsing. We measured the improvement of our structure in terms of the number of runs in the XBWT as a first step, but it remains to implement and evaluate the full data structure in terms of space usage and query time.

From a more general point of view, there remains a lot to be done with sketches. In this thesis, we heavily used the Karp–Rabin fingerprints to improve the efficiency of our algorithms and this is likely extendable to many other problems. Another approach common to Chapters 2, 3 and 6 was working on compressed input (either a straight-line program or a run-length compressed string) and there again I believe there is more to be done. I find this direction especially interesting because of the practical perspective of being able to always work directly on the compressed data without the need to decompress. Several of the classical processing tasks have been implemented already for a straight-line program input, but, in practice, grammar compression requires large construction time and space, an order of magnitude more than to construct the Lempel–Ziv factorization (See column “Re-pair” of Figure 9 and 10 of [330]). So could we improve construction time for grammar compression or develop some algorithms working directly on the LZ factorization?

On the more practical side, I would personally be interested in the use of spaced seeds [96] in Bioinformatics. A space seed is a binary sequence that describes positions that are either relevant (marked by a one) or irrelevant (marked by a zero), then two strings match for the spaced seed if they match at every relevant position. They have been used for homology search [85], alignment [151], assembly [196], and metagenomics [197]. In all those applications they were reported to increase the sensitivity and specificity performances compared to the use of  $k$ -mers, but they cannot be hashed as efficiently as  $k$ -mers and their use incurred a greater computational cost. Recently, Petrucci et al. [322] proposed a technique that greatly speeds up space seed hashing. Consequently, I wonder if sketching techniques used in Bioinformatics (presented at the end of Section 2) could gain accuracy by being based on spaced seeds rather than  $k$ -mers.

# Bibliography

---

- [1] Alex Thue, “Über unendliche Zeichenreihen”, in: *Norske Vid. Selsk. Skr., I Mat.-Nat. Kl., Christiania* 7 (1906), pp. 1–22.
- [2] Axel Thue, *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen*, 1, Jacob Dybwad, 1912.
- [3] Stephen C. Kleene, *Representation of events in nerve nets and finite automata*, RAND Corporation, 1951.
- [4] Rene De La Briandais, “File searching using variable length keys”, in: *Papers presented at the March 3-5, 1959, western joint computer conference*, 1959, pp. 295–298.
- [5] Edward Fredkin, “Trie memory”, in: *Communications of the ACM* 3.9 (1960), pp. 490–499.
- [6] John Barkley Rosser and Lowell Schoenfeld, “Approximate formulas for some functions of prime numbers”, in: *Illinois Journal of Mathematics* 6.1 (1962), pp. 64–94, DOI: 10.1215/ijm/1255631807.
- [7] N. J. Fine and Herbert S. Wilf, “Uniqueness theorems for periodic functions”, in: *Proceedings of the American Mathematical Society*, vol. 16, 1965, pp. 109–114, DOI: 10.1090/S0002-9939-1965-0174934-9.
- [8] Vladimir I Levenshtein, “Binary codes capable of correcting deletions, insertions, and reversals”, in: *Soviet physics doklady*, vol. 10, 8, Soviet Union, 1966, pp. 707–710.
- [9] Ken Thompson, “Programming techniques: regular expression search algorithm”, in: *Communication of ACM* 11.6 (1968), pp. 419–422, ISSN: 0001-0782, DOI: 10.1145/363347.363387.
- [10] IUPAC-IUB Comm, “IUPAC-IUB Commission on Biochemical Nomenclature. Abbreviations and symbols for the description of the conformation of polypeptide chains. Tentative rules (1969)”, in: *Biochemistry* 9.18 (1970), pp. 3471–3479.
- [11] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg, “Rapid Identification of Repeated Patterns in Strings, Trees and Arrays”, in: *Proceedings of the Fourth Annual ACM Symposium on Theory of Computing*, 1972, pp. 125–136, DOI: 10.1145/800152.804905.
- [12] Peter Weiner, “Linear Pattern Matching Algorithms”, in: *14<sup>th</sup> Annual Symposium on Switching and Automata Theory (SWAT 1973)*, 1973, pp. 1–11.
- [13] Enrico Bombieri, *Le grand crible dans la théorie analytique des nombres*, fr, Astérisque 18, Société mathématique de France, 1974.
- [14] Michael J. Fischer and Michael S. Paterson, “String matching and other products”, in: *Complexity of Computation*, 1974, pp. 113–125.
- [15] Alfred V. Aho and Margaret J. Corasick, “Efficient string matching: an aid to bibliographic search”, in: *Communications of the ACM* 18.6 (1975), pp. 333–340, ISSN: 0001-0782, DOI: 10.1145/360825.360855.
- [16] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt, “Fast Pattern Matching in Strings”, in: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350, DOI: 10.1137/0206024.

- 
- [17] Jacob Ziv and Abraham Lempel, “A universal algorithm for sequential data compression”, in: *IEEE Transactions on information theory* 23.3 (1977), pp. 337–343.
- [18] Hiroaki Sakoe and Seibi Chiba, “Dynamic programming algorithm optimization for spoken word recognition”, in: *IEEE transactions on acoustics, speech, and signal processing* 26.1 (1978), pp. 43–49.
- [19] Dwight R. Bean, Andrzej Ehrenfeucht, and George F. McNulty, “Avoidable patterns in strings of symbols.”, in: *Pacific Journal of Mathematics* 85.2 (1979), pp. 261–294.
- [20] Antony Hewish, S Joceyln Bell, John DH Pilkington, Paul Frederick Scott, and Robin Ashley Collins, “Observation of a rapidly pulsating radio source”, in: *A Source Book in Astronomy and Astrophysics, 1900–1975*, Harvard University Press, 1979, pp. 498–504.
- [21] Jean Vuillemin, “A Unifying Look at Data Structures”, in: *Commun. ACM* 23.4 (1980), pp. 229–239.
- [22] Maxime Crochemore, “An Optimal Algorithm for Computing the Repetitions in a Word”, in: *Inf. Process. Lett.* 12.5 (1981), pp. 244–250, DOI: 10.1016/0020-0190(81)90024-7.
- [23] Min-Te Chao, “A general purpose unequal probability sampling plan”, in: *Biometrika* 69.3 (Dec. 1982), pp. 653–656, DOI: 10.2307/2336002.
- [24] Alberto Apostolico and Franco P. Preparata, “Optimal Off-Line Detection of Repetitions in a String”, in: *Theor. Comput. Sci.* 22 (1983), pp. 297–315, DOI: 10.1016/0304-3975(83)90109-3.
- [25] Zvi Galil and Joel Seiferas, “Time-space-optimal string matching”, in: *Journal of Computer and System Sciences* 26.3 (1983), pp. 280–294, ISSN: 0022-0000, DOI: 10.1016/0022-0000(83)90002-8.
- [26] William B. Johnson and Joram Lindenstrauss, “Extensions of Lipschitz mappings into a Hilbert space”, in: *Modern Analysis and Probability*, vol. 26, Contemporary Mathematics, 1984, pp. 189–206.
- [27] Michael G. Main and Richard J. Lorentz, “An  $\mathcal{O}(n \log n)$  Algorithm for Finding All Repetitions in a String”, in: *J. Algorithms* 5.3 (1984), pp. 422–432, DOI: 10.1016/0196-6774(84)90021-X.
- [28] Zvi Galil, “Open problems in stringology”, in: *Combinatorial Algorithms on Words*, Springer Berlin Heidelberg, 1985, pp. 1–8, ISBN: 978-3-642-82456-2.
- [29] Mamoru Maekawa, “A  $\sqrt{N}$  Algorithm for Mutual Exclusion in Decentralized Systems”, in: *ACM Trans. Comput. Syst.* 3.2 (1985), pp. 145–159.
- [30] Maxime Crochemore, “Transducers and Repetitions”, in: *Theor. Comput. Sci.* 45.1 (1986), pp. 63–86, DOI: 10.1016/0304-3975(86)90041-1.
- [31] Zvi Galil and Raffaele Giancarlo, “Improved String Matching with  $k$  Mismatches”, in: *SIGACT News* 17.4 (Mar. 1986), pp. 52–54, ISSN: 0163-5700, DOI: 10.1145/8307.8309.
- [32] Gad M Landau and Uzi Vishkin, “Efficient string matching with  $k$  mismatches”, in: *Theoretical Computer Science* 43 (1986), pp. 239–249.
- [33] Karl Abrahamson, “Generalized string matching”, in: *SIAM journal on Computing* 16.6 (1987), pp. 1039–1051.
- [34] Michael Gribskov, Andrew D McLachlan, and David Eisenberg, “Profile analysis: detection of distantly related proteins.”, in: *Proceedings of the National Academy of Sciences* 84.13 (1987), pp. 4355–4358, DOI: 10.1073/pnas.84.13.4355.

- 
- [35] R. M. Karp and M. O. Rabin, “Efficient randomized pattern-matching algorithms”, in: *IBM Journal of Research and Development* 31.2 (1987), pp. 249–260, DOI: 10.1147/rd.312.0249.
  - [36] Maxime Crochemore, “Constant-Space String-Matching”, in: *Proc. 8<sup>th</sup> FSTTCS*, 1988, pp. 80–87.
  - [37] Gad M. Landau and Uzi Vishkin, “Fast String Matching with k Differences”, in: *Journal of Computer and System Sciences* 37.1 (1988), pp. 63–78, DOI: 10.1016/0022-0000(88)90045-1.
  - [38] Simha Arom, “Time structure in the music of Central Africa: Periodicity, meter, rhythm and polyrhythmics”, in: *Leonardo* 22.1 (1989), pp. 91–99.
  - [39] Gad M Landau and Uzi Vishkin, “Fast parallel and serial approximate string matching”, in: *Journal of algorithms* 10.2 (1989), pp. 157–169.
  - [40] Harold N. Gabow, “Data Structures for Weighted Matching and Nearest Common Ancestors with Linking”, in: *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms, 22-24 January 1990, San Francisco, California, USA*, ed. by David S. Johnson, SIAM, 1990, pp. 434–443.
  - [41] Marcel Mongeau and David Sankoff, “Comparison of musical sequences”, in: *Computers and the Humanities* 24 (1990), pp. 161–175.
  - [42] Dany Breslauer, “Efficient String Algorithmics”, PhD thesis, Columbia University, 1992.
  - [43] Aditi Dhagat, Peter Gács, and Peter Winkler, “On Playing “Twenty Questions” with a Liar”, in: *SODA ’92*, 1992, pp. 16–22, ISBN: 0-89791-466-X.
  - [44] Eugene W. Myers, “A four Russians algorithm for regular expression pattern matching”, in: *Journal of the ACM* 39.2 (Apr. 1992), pp. 432–448, ISSN: 0004-5411, DOI: 10.1145/128749.128755.
  - [45] Brenda S Baker, “A theory of parameterized pattern matching: algorithms and applications”, in: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, 1993, pp. 71–80.
  - [46] Greg N Frederickson, “An Optimal Algorithm for Selection in a Min-Heap”, in: *Information and Computation* 104.2 (1993), pp. 197–214, ISSN: 0890-5401, DOI: 10.1006/inco.1993.1030.
  - [47] Udi Manber and Eugene W. Myers, “Suffix Arrays: A New Method for On-Line String Searches”, in: *SIAM J. Comput.* 22.5 (1993), pp. 935–948.
  - [48] Michael Burrows and David Wheeler, “A block-sorting lossless data compression algorithm”, in: *Digital SRC Research Report*, 1994.
  - [49] S. Rao Kosaraju, “Computation of Squares in a String (Preliminary Version)”, in: *Combinatorial Pattern Matching, 5<sup>th</sup> Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994, Proceedings*, ed. by Maxime Crochemore and Dan Gusfield, vol. 807, Lecture Notes in Computer Science, Springer, 1994, pp. 146–150, DOI: 10.1007/3-540-58094-8\_13.
  - [50] Julie D Thompson, Desmond G Higgins, and Toby J Gibson, “CLUSTAL W: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice”, in: *Nucleic acids research* 22.22 (1994), pp. 4673–4680.

- 
- [51] Alberto Apostolico and Dany Breslauer, “An Optimal  $\mathcal{O}(\log \log N)$ -Time Parallel Algorithm for Detecting All Squares in a String”, in: *SIAM J. Comput.* 25.6 (1996), pp. 1318–1331, DOI: 10.1137/S0097539793260404.
  - [52] Costas S Iliopoulos, Dennis WG Moore, and Kunsoo Park, “Covering a string”, in: *Algorithmica* 16.3 (1996), pp. 288–297.
  - [53] Julian Seward, “bzip2 and libbzip2”, in: *available at <http://www.bzip.org>* (1996).
  - [54] A.Z. Broder, “On the resemblance and containment of documents”, in: *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, 1997, pp. 21–29, DOI: 10.1109/SEQUEN.1997.666900.
  - [55] Richard Cole and Ramesh Hariharan, “Tighter Upper Bounds on the Exact Complexity of String Matching”, in: *SIAM J. Comput.* 26.3 (1997), pp. 803–856, DOI: 10.1137/S009753979324694X.
  - [56] Martin Farach, “Optimal Suffix Tree Construction with Large Alphabets”, in: *38<sup>th</sup> Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, IEEE Computer Society, 1997, pp. 137–143, DOI: 10.1109/SFCS.1997.646102.
  - [57] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*, Cambridge University Press, 1997, ISBN: 0-521-58519-8, DOI: 10.1017/cbo9780511574931.
  - [58] Jean-Paul Allouche and Jeffrey O. Shallit, “The Ubiquitous Prouhet-Thue-Morse Sequence”, in: *Sequences and their Applications - Proceedings of SETA 1998, Singapore, December 14-17, 1998*, ed. by Cunsheng Ding, Tor Helleseth, and Harald Niederreiter, Discrete Mathematics and Theoretical Computer Science, Springer, 1998, pp. 1–16, DOI: 10.1007/978-1-4471-0551-0\_1.
  - [59] Aviezri S. Fraenkel and Jamie Simpson, “How Many Squares Can a String Contain?”, in: *J. Comb. Theory, Ser. A* 82.1 (1998), pp. 112–120, DOI: 10.1006/jcta.1997.2843.
  - [60] Piotr Indyk, “Faster algorithms for string matching problems: matching the convolution bound”, in: *1998 IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998, p. 166, ISBN: 0818691727, DOI: 10.1109/SFCS.1998.743440.
  - [61] Gad M. Landau, Eugene W. Myers, and Jeanette P. Schmidt, “Incremental String Comparison”, in: *SIAM Journal on Computing* 27.2 (1998), pp. 557–582, DOI: 10.1137/S0097539794264810.
  - [62] Jean Berstel and Luc Boasson, “Partial Words and a Theorem of Fine and Wilf”, in: *Theor. Comput. Sci.* 218.1 (1999), pp. 135–141, DOI: 10.1016/S0304-3975(98)00255-2.
  - [63] Gerth Stølting Brodal, Rune B Lyngsø, Christian NS Pedersen, and Jens Stoye, “Finding maximal pairs with bounded gap”, in: *Combinatorial Pattern Matching: 10<sup>th</sup> Annual Symposium, CPM 99 Warwick University, UK, July 22–24, 1999 Proceedings 10*, Springer, 1999, pp. 134–149.
  - [64] Maria Gabriella Castelli, Filippo Mignosi, and Antonio Restivo, “Fine and Wilf’s Theorem for Three Periods and a Generalization of Sturmian Words”, in: *Theor. Comput. Sci.* 218.1 (1999), pp. 83–94, DOI: 10.1016/S0304-3975(98)00251-5.

- 
- [65] Roman M. Kolpakov and Gregory Kucherov, “Finding Maximal Repetitions in a Word in Linear Time”, in: *40<sup>th</sup> Annual Symposium on Foundations of Computer Science, FOCS ’99, 17-18 October, 1999, New York, NY, USA*, IEEE Computer Society, 1999, pp. 596–604, DOI: 10.1109/SFFCS.1999.814634.
- [66] Michael A Bender and Martin Farach-Colton, “The LCA problem revisited”, in: *LATIN 2000: Theoretical Informatics: 4th Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000 Proceedings 4*, Springer, 2000, pp. 88–94.
- [67] Michael A. Bender and Martin Farach-Colton, “The LCA Problem Revisited”, in: *LATIN 2000: Theoretical Informatics, 4<sup>th</sup> Latin American Symposium, Punta del Este, Uruguay, April 10-14, 2000, Proceedings*, ed. by Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, vol. 1776, Lecture Notes in Computer Science, Springer, 2000, pp. 88–94, DOI: 10.1007/10719839\_9.
- [68] Jacques Justin, “On a paper by Castelli, Mignosi, Restivo”, in: *RAIRO Theor. Informatics Appl.* 34.5 (2000), pp. 373–377, DOI: 10.1051/ita:2000122.
- [69] John C. Kieffer and En-Hui Yang, “Grammar-based codes: A new class of universal lossless source codes”, in: *IEEE Trans. Inf. Theory* 46.3 (2000), pp. 737–754.
- [70] Russell Impagliazzo and Ramamohan Paturi, “On the complexity of k-SAT”, in: *Journal of Computer and System Sciences* 62.2 (2001), pp. 367–375, ISSN: 0022-0000, DOI: 10.1006/jcss.2000.1727.
- [71] Russell Impagliazzo, Ramamohan Paturi, and Francis Zane, “Which Problems Have Strongly Exponential Complexity?”, in: *Journal of Computer and System Sciences* 63.4 (2001), pp. 512–530, DOI: 10.1006/jcss.2001.1774.
- [72] Quanzhong Li and Bongki Moon, “Indexing and querying XML data for regular path expressions”, in: *2001 International Conference on Very Large Data Bases (VLDB)*, 2001, pp. 361–370, ISBN: 1558608044.
- [73] Makoto Murata, “Extended path expressions of XML”, in: *2001 ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2001, pp. 126–137, ISBN: 1581133618, DOI: 10.1145/375551.375569.
- [74] Gonzalo Navarro, “A guided tour to approximate string matching”, in: *ACM computing surveys (CSUR)* 33.1 (2001), pp. 31–88.
- [75] Gonzalo Navarro and Mathieu Raffinot, “Fast and simple character classes and bounded gaps pattern matching, with application to protein searching”, in: *2001 International Conference on Computational Biology (RECOMB)*, 2001, pp. 231–240, ISBN: 1581133537, DOI: 10.1145/369133.369220.
- [76] Arseny M. Shur and Yulia V. Konovalova, “On the Periods of Partial Words”, in: *Mathematical Foundations of Computer Science 2001, 26<sup>th</sup> International Symposium, MFCS 2001 Mariánské Lázně, Czech Republic, August 27-31, 2001, Proceedings*, ed. by Jirí Sgall, Ales Pultr, and Petr Kolman, vol. 2136, Lecture Notes in Computer Science, Springer, 2001, pp. 657–665, DOI: 10.1007/3-540-44683-4\_57.
- [77] Jeong Seop Sim, Costas S. Iliopoulos, Kunsoo Park, and W.F. Smyth, “Approximate periods of strings”, in: *Theoretical Computer Science* 262.1 (2001), pp. 557–568, ISSN: 0304-3975, DOI: 10.1016/S0304-3975(00)00365-0.



- 
- [78] Ziv Bar-Yossef, TS Jayram, Ravi Kumar, D Sivakumar, and Luca Trevisan, “Counting distinct elements in a data stream”, in: *Randomization and Approximation Techniques in Computer Science: 6<sup>th</sup> International Workshop, RANDOM 2002 Cambridge, MA, USA, September 13–15, 2002 Proceedings* 5, Springer, 2002, pp. 1–10.
- [79] Francine Blanchet-Sadri and Robert A. Hegstrom, “Partial words and a theorem of Fine and Wilf revisited”, in: *Theor. Comput. Sci.* 270.1-2 (2002), pp. 401–419, DOI: 10.1016/S0304-3975(00)00407-2.
- [80] Moses Charikar, Eric Lehman, Ding Liu, Rina Panigrahy, Manoj Prabhakaran, April Rasala, Amit Sahai, and Abhi Shelat, “Approximating the smallest grammar: Kolmogorov complexity in natural models”, in: *Proc. 34<sup>th</sup> STOC*, 2002, pp. 792–801.
- [81] Richard Cole and Ramesh Hariharan, “Verifying candidate matches in sparse and wildcard matching”, in: *2002 ACM Symposium on Theory of Computing (STOC)*, 2002, pp. 592–601, ISBN: 1581134959, DOI: 10.1145/509907.509992.
- [82] Minos N. Garofalakis, Rajeev Rastogi, and Kyuseok Shim, “Mining sequential patterns with regular expression constraints”, in: *IEEE Transactions on Knowledge and Data Engineering (TKDE)* 14.3 (2002), pp. 530–552, DOI: 10.1109/TKDE.2002.1000341.
- [83] Yijie Han and Mikkel Thorup, “Integer Sorting in  $\mathcal{O}(n\sqrt{\log \log n})$  Expected Time and Linear Space”, in: *FOCS*, IEEE Computer Society, 2002, pp. 135–144.
- [84] Adam Kalai, “Efficient pattern-matching with don’t cares”, in: *2002 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 655–656, ISBN: 089871513X.
- [85] Bin Ma, John Tromp, and Ming Li, “PatternHunter: faster and more sensitive homology search”, in: *Bioinformatics* 18.3 (2002), pp. 440–445.
- [86] Shanmugavelayutham Muthukrishnan, “Efficient algorithms for document retrieval problems.”, in: *SODA*, vol. 2, 2002, pp. 657–666.
- [87] Dimitris Achlioptas, “Database-friendly random projections: Johnson-Lindenstrauss with binary coins”, in: *Journal of Computer and System Sciences* 66.4 (2003), pp. 671–687, ISSN: 0022-0000, DOI: 10.1016/S0022-0000(03)00025-4.
- [88] Sebastian Böcker, “Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt”, in: *Algorithms in Bioinformatics: Third International Workshop, WABI 2003, Budapest, Hungary, September 15-20, 2003. Proceedings* 3, Springer, 2003, pp. 476–497.
- [89] Jittat Fakcharoenphol, Satish Rao, and Kunal Talwar, “A Tight Bound on Approximating Arbitrary Metrics by Tree Metrics”, in: *STOC’03*, 2003, pp. 448–455, ISBN: 1581136749, DOI: 10.1145/780542.780608.
- [90] Roman M. Kolpakov, Ghizlane Bana, and Gregory Kucherov, “mreps: efficient and flexible detection of tandem repeats in DNA”, in: *Nucleic Acids Res.* 31.13 (2003), pp. 3672–3678, DOI: 10.1093/nar/gkg617.
- [91] Wojciech Rytter, “Application of Lempel-Ziv factorization to the approximation of grammar-based compression”, in: *Theor. Comput. Sci.* 302.1-3 (2003), pp. 211–222.
- [92] Robert Tijdeman and Luca Zamboni, “Fine and Wilf words for any periods”, in: *Indagationes Mathematicae* 14.1 (2003), pp. 135–147, ISSN: 0019-3577, DOI: 10.1016/S0019-3577(03)90076-0.
- [93] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena, “Primes is in P”, in: *Annals of Mathematics* 160.2 (2004), pp. 781–793, ISSN: 0003-486X, DOI: 10.4007/annals.2004.160.781.

- 
- [94] Christian Charras and Thierry Lecroq, *Handbook of exact string matching algorithms*, King's College Publications, 2004.
- [95] Revital Eres, Gad M Landau, and Laxmi Parida, "Permutation pattern discovery in biosequences", in: *Journal of Computational Biology* 11.6 (2004), pp. 1050–1060.
- [96] Ming Li, Bin Ma, Derek Kisman, and John Tromp, "PatternHunter II: Highly sensitive and fast homology search", in: *Journal of bioinformatics and computational biology* 2.03 (2004), pp. 417–439.
- [97] A M Shur and Yu V Gamzova, "Partial words and the interaction property of periods", in: *Izvestiya: Mathematics* 68.2 (2004), pp. 405–428, DOI: 10.1070/im2004v068n02abeh000480.
- [98] James D. Currie, "Pattern avoidance: themes and variations", in: *Theor. Comput. Sci.* 339.1 (2005), pp. 7–18.
- [99] Paolo Ferragina and Giovanni Manzini, "Indexing compressed text", in: *Journal of the ACM (JACM)* 52.4 (2005), pp. 552–581.
- [100] Leszek Gąsieniec, Roman M. Kolpakov, Igor Potapov, and Paul Sant, "Real-Time Traversal in Grammar-Based Compressed Files", in: *Proc. 15<sup>th</sup> DCC*, 2005, p. 458.
- [101] Alexandr Andoni and Piotr Indyk, "Efficient algorithms for substring near neighbor problem", in: *SODA'06*, 2006, pp. 1203–1212, DOI: 10.1145/1109557.1109690.
- [102] Philip Bille, "New algorithms for regular expression matching", in: *2006 International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 4051, Lecture Notes in Computer Science, Springer, 2006, pp. 643–654, DOI: 10.1007/11786986\_56.
- [103] Sorin Constantinescu and Lucian Ilie, "Fine and Wilf's theorem for abelian periods", in: *Bulletin of the EATCS* 89 (Jan. 2006), pp. 167–170.
- [104] Thomas M. Cover and Joy A. Thomas, *Elements of information theory*, 2<sup>nd</sup>, Wiley, 2006, ISBN: 978-0-471-24195-9, URL: <http://www.elementsofinformationtheory.com/>.
- [105] Johannes Fischer and Volker Heun, "Theoretical and Practical Improvements on the RMQ-Problem, with Applications to LCA and LCE", in: *CPM 2006*, 2006, pp. 36–48, DOI: 10.1007/11780441\_5.
- [106] Nicolas Hulo, Amos Bairoch, Virginie Bulliard, Lorenzo Cerutti, Edouard De Castro, Petra S Langendijk-Genevaux, Marco Pagni, and Christian JA Sigrist, "The PROSITE database", in: *Nucleic acids research* 34.suppl\_1 (2006), pp. D227–D230.
- [107] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan S. Turner, "Algorithms to accelerate multiple regular expressions matching for deep packet inspection", in: *2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, 2006, pp. 339–350, ISBN: 1595933085, DOI: 10.1145/1159913.1159952.
- [108] Pierre Peterlongo, Julien Allali, and Marie-France Sagot, "The gapped-factor tree", in: *Prague Stringology Conference 2006*, 2006, pp. 182–196.
- [109] Wojciech Rytter, "The Number of Runs in a String: Improved Analysis of the Linear Upper Bound", in: *STACS 2006, 23<sup>rd</sup> Annual Symposium on Theoretical Aspects of Computer Science, Marseille, France, February 23-25, 2006, Proceedings*, ed. by Bruno Durand and Wolfgang Thomas, vol. 3884, Lecture Notes in Computer Science, Springer, 2006, pp. 184–195, DOI: 10.1007/11672142\_14.

- 
- [110] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz, “Fast and memory-efficient regular expression matching for deep packet inspection”, in: *2006 Symposium on Architecture For Networking And Communications Systems*, 2006, pp. 93–102, DOI: 10.1145/1185347.1185360.
- [111] Peter Clifford and Raphael Clifford, “Simple deterministic wildcard matching”, in: *Information Processing Letters* 101.2 (2007), pp. 53–54, ISSN: 0020-0190, DOI: 10.1016/j.ip1.2006.08.002.
- [112] Andrew C Harvey, Thomas M Trimbur, and Herman K Van Dijk, “Trends and cycles in economic time series: A Bayesian approach”, in: *Journal of Econometrics* 140.2 (2007), pp. 618–649.
- [113] Lucian Ilie, “A note on the number of squares in a word”, in: *Theor. Comput. Sci.* 380.3 (2007), pp. 373–376, DOI: 10.1016/j.tcs.2007.03.025.
- [114] Theodore Johnson, Shan Muthu Muthukrishnan, and Irina Rozenbaum, “Monitoring regular expressions on out-of-order streams”, in: *2007 IEEE International Conference on Data Engineering (ICDE)*, 2007, pp. 1315–1319, DOI: 10.1109/ICDE.2007.369001.
- [115] Orgad Keller, Tsvi Kopelowitz, and Moshe Lewenstein, “Range non-overlapping indexing and successive list indexing”, in: *Algorithms and Data Structures: 10<sup>th</sup> International Workshop, WADS 2007, Halifax, Canada, August 15-17, 2007. Proceedings 10*, Springer, 2007, pp. 625–636.
- [116] Romans Lukashenko, Vita Graudina, and Janis Grundspenkis, “Computer-based plagiarism detection methods and tools: an overview”, in: *Proceedings of the 2007 international conference on Computer systems and technologies*, 2007, pp. 1–6.
- [117] S. Mantaci, A. Restivo, G. Rosone, and M. Sciortino, “An extension of the Burrows–Wheeler Transform”, in: *Theoretical Computer Science* 387.3 (2007), pp. 298–312, DOI: 10.1016/j.tcs.2007.07.014.
- [118] Alberto Apostolico and Raffaele Giancarlo, “Periodicity and repetitions in parameterized strings”, in: *Discret. Appl. Math.* 156.9 (2008), pp. 1389–1398, DOI: 10.1016/j.dam.2006.11.017.
- [119] Philip Bille and Martin Farach-Colton, “Fast and compact regular expression matching”, in: *Theoretical Computer Science* 409.3 (2008), pp. 486–496, ISSN: 0304-3975, DOI: 10.1016/j.tcs.2008.08.042.
- [120] Francine Blanchet-Sadri, Deepak Bal, and Gautam Sisodia, “Graph connectivity, partial words, and a theorem of Fine and Wilf”, in: *Inf. Comput.* 206.5 (2008), pp. 676–693, DOI: 10.1016/j.ic.2007.11.007.
- [121] Maxime Crochemore and Lucian Ilie, “Maximal repetitions in strings”, in: *J. Comput. Syst. Sci.* 74.5 (2008), pp. 796–807, DOI: 10.1016/j.jcss.2007.09.003.
- [122] Frantisek Franek and Qian Yang, “An asymptotic Lower Bound for the Maximal Number of Runs in a String”, in: *Int. J. Found. Comput. Sci.* 19.1 (2008), pp. 195–203, DOI: 10.1142/S0129054108005620.
- [123] Mathieu Giraud, “Not So Many Runs in Strings”, in: *Language and Automata Theory and Applications, Second International Conference, LATA 2008, Tarragona, Spain, March 13-19, 2008. Revised Papers*, ed. by Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, vol. 5196, Lecture Notes in Computer Science, Springer, 2008, pp. 232–239, DOI: 10.1007/978-3-540-88282-4\_22.

- 
- [124] Jin-Ju Hong and Gen-Huey Chen, “Efficient on-line repetition detection”, in: *Theor. Comput. Sci.* 407.1-3 (2008), pp. 554–563, DOI: 10.1016/j.tcs.2008.08.038.
  - [125] Wataru Matsubara, Kazuhiko Kusano, Akira Ishino, Hideo Bannai, and Ayumi Shinohara, “New Lower Bounds for the Maximum Number of Runs in a String”, in: *Proceedings of the Prague Stringology Conference 2008, Prague, Czech Republic, September 1-3, 2008*, ed. by Jan Holub and Jan Zdárek, Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, 2008, pp. 140–145.
  - [126] Simon J. Puglisi, Jamie Simpson, and William F. Smyth, “How many runs can a string contain?”, in: *Theor. Comput. Sci.* 401.1-3 (2008), pp. 165–171, DOI: 10.1016/j.tcs.2008.04.020.
  - [127] Philip Bille and Mikkel Thorup, “Faster regular expression matching”, in: *2009 International Colloquium on Automata, Languages, and Programming (ICALP)*, Springer Berlin Heidelberg, 2009, pp. 171–182, DOI: 10.1007/978-3-642-02927-1\_16.
  - [128] Jian-Qun Chen, Ying Wu, Haiwang Yang, Joy Bergelson, Martin Kreitman, and Dacheng Tian, “Variation in the Ratio of Nucleotide Substitution and Indel Rates across Genomes in Mammals and Bacteria”, in: *Molecular Biology and Evolution* 26.7 (Mar. 2009), pp. 1523–1531, ISSN: 0737-4038, DOI: 10.1093/molbev/msp063.
  - [129] Hagai Cohen and Ely Porat, “Range non-overlapping indexing”, in: *Algorithms and Computation: 20<sup>th</sup> International Symposium, ISAAC 2009, Honolulu, Hawaii, USA, December 16-18, 2009. Proceedings 20*, Springer, 2009, pp. 1044–1053.
  - [130] Paolo Ferragina, Fabrizio Luccio, Giovanni Manzini, and Senthilmurugan Muthukrishnan, “Compressing and indexing labeled trees, with applications”, in: *Journal of the ACM (JACM)* 57.1 (2009), pp. 1–33.
  - [131] Mathieu Giraud, “Asymptotic behavior of the numbers of runs and microruns”, in: *Inf. Comput.* 207.11 (2009), pp. 1221–1228, DOI: 10.1016/j.ic.2009.02.007.
  - [132] Costas S Iliopoulos and M Sohel Rahman, “Indexing factors with gaps”, in: *Algorithmica* 55.1 (2009), pp. 60–70.
  - [133] Juha Kärkkäinen, Giovanni Manzini, and Simon J Puglisi, “Permuted longest-common-prefix array”, in: *Annual Symposium on Combinatorial Pattern Matching*, Springer, 2009, pp. 181–192.
  - [134] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg, “Ultrafast and memory-efficient alignment of short DNA sequences to the human genome”, in: *Genome biology* 10.3 (2009), pp. 1–10.
  - [135] Heng Li and Richard Durbin, “Fast and accurate short read alignment with Burrows-Wheeler transform”, in: *Bioinformatics* 25.14 (2009), pp. 1754–1760, DOI: 10.1093/bioinformatics/btp324.
  - [136] Wataru Matsubara, Kazuhiko Kusano, Hideo Bannai, and Ayumi Shinohara, “A Series of Run-Rich Strings”, in: *Language and Automata Theory and Applications, Third International Conference, LATA 2009, Tarragona, Spain, April 2-8, 2009. Proceedings*, ed. by Adrian-Horia Dediu, Armand-Mihai Ionescu, and Carlos Martín-Vide, vol. 5457, Lecture Notes in Computer Science, Springer, 2009, pp. 578–587, DOI: 10.1007/978-3-642-00982-2\_49.
  - [137] Benny Porat and Ely Porat, “Exact and Approximate Pattern Matching in the Streaming Model”, in: *FOCS’09*, 2009, pp. 315–323, DOI: 10.1109/FOCS.2009.11.

- 
- [138] Amihood Amir, Estrella Eisenberg, Avivit Levy, Ely Porat, and Natalie Shapira, “Cycle Detection and Correction”, in: *Automata, Languages and Programming*, ed. by Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis, Springer Berlin Heidelberg, 2010, pp. 43–54, ISBN: 978-3-642-14165-2.
  - [139] Djamel Belazzougui, Paolo Boldi, Rasmus Pagh, and Sebastiano Vigna, “Fast Prefix Search in Little Space, with Applications”, in: *Proc. 18<sup>th</sup> ESA*, 2010, pp. 427–438.
  - [140] Philip Bille and Mikkel Thorup, “Regular expression matching with multi-strings and intervals”, in: *2010 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2010, pp. 1297–1308, DOI: 10.1137/1.9781611973075.104.
  - [141] Tahir Ejaz, “Abelian pattern matching in strings.”, PhD thesis, Dortmund University of Technology, 2010.
  - [142] Funda Ergün, Hossein Jowhari, and Mert Sağlam, “Periodicity in streams”, in: *2010 International Conference on Approximation Algorithms for Combinatorial Optimization (APPROX)*, 2010, pp. 545–559, DOI: 10.1007/978-3-642-15369-3\_41.
  - [143] Simone Faro and Thierry Lecroq, “The Exact String Matching Problem: a Comprehensive Experimental Evaluation”, in: *CoRR* abs/1012.2547 (2010), arXiv: 1012.2547.
  - [144] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter, “String Retrieval for Multi-pattern Queries”, in: *Proc. 17<sup>th</sup> SPIRE*, 2010, pp. 55–66.
  - [145] Daniel Lokshtanov and Jesper Nederlof, “Saving space by algebraization”, in: *ACM Symposium on Theory of Computing (STOC)*, 2010, pp. 321–330, DOI: 10.1145/1806689.1806735.
  - [146] Jamie Simpson, “Modified Padovan words and the maximum number of runs in a word”, in: *Australas. J Comb.* 46 (2010), pp. 129–146.
  - [147] Maxim Babenko and Tatiana Starikovskaya, “Computing the longest common substring with one mismatch”, in: *Problems of Information Transmission* 47.1 (2011), pp. 28–33, DOI: 10.1134/S0032946011010030.
  - [148] Nikhil Bansal, Niv Buchbinder, Aleksander Madry, and Joseph Naor, “A Polylogarithmic-Competitive Algorithm for the k-Server Problem”, in: *FOCS’11*, 2011, pp. 267–276, DOI: 10.1109/FOCS.2011.63.
  - [149] Hamidreza Chitsaz, Joyclyn Yee-Greenbaum, Glenn Tesler, Mary-Jane Lombardo, Christopher Dupont, Jonathan Badger, Mark Novotny, Douglas Rusch, Louise Fraser, Niall Gormley, Ole Schulz-Trieglaff, Geoffrey Smith, Dirk Evers, Pavel Pevzner, and Roger Lasken, “Efficient de novo assembly of single-cell bacterial genomes from short-read data sets”, in: *Nature biotechnology* 29 (Sept. 2011), pp. 915–21, DOI: 10.1038/nbt.1966.
  - [150] Maxime Crochemore, Lucian Ilie, and Liviu Tinta, “The "runs" conjecture”, in: *Theor. Comput. Sci.* 412.27 (2011), pp. 2931–2941, DOI: 10.1016/j.tcs.2010.06.019.
  - [151] Matei David, Misko Dzamba, Dan Lister, Lucian Ilie, and Michael Brudno, “SHRiMP2: sensitive yet practical short read mapping”, in: *Bioinformatics* 27.7 (2011), pp. 1011–1012.
  - [152] Moshe Lewenstein, “Indexing with gaps”, in: *String Processing and Information Retrieval: 18<sup>th</sup> International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings* 18, Springer, 2011, pp. 135–143.
  - [153] Daniel Tunkelang, *Retiring a great interview problem*, <https://thenoisychannel.com/2011/08/08/retiring-a-great-interview-problem/>, 2011.

- 
- [154] Amihood Amir and Avivit Levy, “Approximate Period Detection and Correction”, in: *String Processing and Information Retrieval - 19<sup>th</sup> International Symposium, SPIRE 2012, Cartagena de Indias, Colombia, October 21-25, 2012. Proceedings*, ed. by Liliana Calderón-Benavides, Cristina N. González-Caro, Edgar Chávez, and Nivio Ziviani, vol. 7608, Lecture Notes in Computer Science, Springer, 2012, pp. 1–15, DOI: 10.1007/978-3-642-34109-0\_1.
- [155] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and David Kofoed Wind, “String matching with variable length gaps”, in: *Theoretical Computer Science* 443 (2012), pp. 25–34.
- [156] Alexander Bowe, Taku Onodera, Kunihiro Sadakane, and Tetsuo Shibuya, “Succinct de Bruijn Graphs”, in: *Algorithms in Bioinformatics*, ed. by Ben Raphael and Jijun Tang, Springer, Springer Berlin Heidelberg, 2012, pp. 225–235.
- [157] Francisco Claude and Gonzalo Navarro, “Improved Grammar-Based Compressed Indexes”, in: *Proc. 19<sup>th</sup> SPIRE*, 2012, pp. 180–192.
- [158] Anthony J Cox, Markus J Bauer, Tobias Jakobi, and Giovanna Rosone, “Large-scale compression of genomic sequence databases with the Burrows-Wheeler transform”, in: *Bioinformatics* 28.11 (2012), pp. 1415–1419, DOI: 10.1093/bioinformatics/bts173.
- [159] Anthony J. Cox, Tobias Jakobi, Giovanna Rosone, and Ole B. Schulz-Trieglaff, “Comparing DNA Sequence Collections by Direct Comparison of Compressed Text Indexes”, in: *Algorithms in Bioinformatics*, ed. by Ben Raphael and Jijun Tang, Springer Berlin Heidelberg, 2012.
- [160] Johannes Fischer, Travis Gagie, Tsvi Kopelowitz, Moshe Lewenstein, Veli Mäkinen, Leena Salmela, and Niko Välimäki, “Forbidden Patterns”, in: *Proc. 10<sup>th</sup> LATIN*, 2012, pp. 327–337.
- [161] Wing-Kai Hon, Rahul Shah, Sharma V. Thankachan, and Jeffrey Scott Vitter, “Document Listing for Queries with Excluded Pattern”, in: *Proc. 23<sup>rd</sup> CPM*, 2012, pp. 185–195.
- [162] Kenrick Kin, Björn Hartmann, Tony DeRose, and Maneesh Agrawala, “Proton: multitouch gestures as regular expressions”, in: *2012 Conference on Human Factors in Computing Systems (CHI)*, 2012, pp. 2885–2894, ISBN: 9781450310154, DOI: 10.1145/2207676.2208694.
- [163] Yuichi Kodama, Martin Shumway, and Rasko Leinonen, “The Sequence Read Archive: explosive growth of sequencing data”, in: *Nucleic acids research* 40.D1 (2012), pp. D54–D56.
- [164] Ben Langmead and Steven L Salzberg, “Fast gapped-read alignment with Bowtie 2”, in: *Nature methods* 9.4 (2012), p. 357.
- [165] Terence Tao, Ernest Croot III, and Harald Helfgott, “Deterministic methods to find primes”, in: *AMS Mathematics of Computation* 81.278 (2012), pp. 1233–1246, DOI: 10.1090/S0025-5718-2011-02542-1.
- [166] Ajesh Babu, Nutan Limaye, Jaikumar Radhakrishnan, and Girish Varma, “Streaming algorithms for language recognition problems”, in: *Theoretical Computer Science* 494 (2013), pp. 13–23, DOI: 10.1016/j.tcs.2012.12.028.
- [167] Markus J Bauer, Anthony J Cox, and Giovanna Rosone, “Lightweight algorithms for constructing and inverting the BWT of string collections”, in: *Theoretical Computer Science* 483 (2013), pp. 134–148.

- 
- [168] Francine Blanchet-Sadri, Sean Simmons, Amelia Tebbe, and Amy Veprauskas, “Abelian periods, partial words, and an extension of a theorem of Fine and Wilf”, in: *RAIRO Theor. Informatics Appl.* 47.3 (2013), pp. 215–234, DOI: 10.1051/ita/2013034.
- [169] Timothy M. Chan, “Persistent Predecessor Search and Orthogonal Point Location on the Word RAM”, in: *ACM Trans. Algorithms* 9.3 (2013), 22:1–22:22.
- [170] Simone Faro and Thierry Lecroq, “The exact online string matching problem: A review of the most recent results”, in: *ACM Computing Surveys (CSUR)* 45.2 (2013), pp. 1–42.
- [171] Marcin Kubica, Tomasz Kulczyński, Jakub Radoszewski, Wojciech Rytter, and Tomasz Waleń, “A linear time algorithm for consecutive permutation pattern matching”, in: *Information Processing Letters* 113.12 (2013), pp. 430–433.
- [172] Moshe Lewenstein, “Orthogonal Range Searching for Text Indexing”, in: *Space-Efficient Data Structures, Streams, and Algorithms*, 2013, pp. 267–302.
- [173] Tanja Magoc, Stephan Pabinger, Stefan Canzar, Xinyue Liu, Qi Su, Daniela Puiu, Luke J. Tallon, and Steven L. Salzberg, “GAGE-B: an evaluation of genome assemblers for bacterial organisms”, in: *Bioinform.* 29.14 (2013), pp. 1718–1725, DOI: 10.1093/bioinformatics/btt273.
- [174] Aleksey V. Zimin, Guillaume Marcais, Daniela Puiu, Michael Roberts, Steven L. Salzberg, and James A. Yorke, “The MaSuRCA genome assembler”, in: *Bioinformatics* 29.21 (Aug. 2013), pp. 2669–2677.
- [175] Philip Bille, Inge Li Gørtz, Hjalte Wedel Vildhøj, and Søren Vind, “String indexing for patterns with wildcards”, in: *Theory of Computing Systems* 55 (2014), pp. 41–60.
- [176] Dany Breslauer and Zvi Galil, “Real-time streaming string-matching”, in: *ACM Transaction on Algorithms* 10.4 (2014), 22:1–22:12, DOI: 10.1145/2635814.
- [177] Jean-Pierre Duval, Thierry Lecroq, and Arnaud Lefebvre, “Linear computation of unbordered conjugate on unordered alphabet”, in: *Theor. Comput. Sci.* 522 (2014), pp. 77–84, DOI: 10.1016/j.tcs.2013.12.008.
- [178] Héctor Ferrada, Travis Gagie, Tommi Hirvola, and Simon J. Puglisi, “Hybrid indexes for repetitive datasets”, in: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 372.2016 (2014), p. 20130137.
- [179] Travis Gagie, Pawel Gawrychowski, Juha Kärkkäinen, Yakov Nekrich, and Simon J. Puglisi, “LZ77-Based Self-indexing with Faster Pattern Matching”, in: *Proc. 11<sup>th</sup> LATIN*, 2014, pp. 731–742.
- [180] Lidia A. Idiatulina and Arseny M. Shur, “Periodic Partial Words and Random Bipartite Graphs”, in: *Fundam. Informaticae* 132.1 (2014), pp. 15–31, DOI: 10.3233/FI-2014-1030.
- [181] Lilian Janin, Ole Schulz-Trieglaff, and Anthony J Cox, “BEETL-fastq: a searchable compressed archive for DNA reads”, in: *Bioinformatics* 30.19 (2014), pp. 2796–2801.
- [182] Jinil Kim, Peter Eades, Rudolf Fleischer, Seok-Hee Hong, Costas S Iliopoulos, Kunsoo Park, Simon J Puglisi, and Takeshi Tokuyama, “Order-preserving matching”, in: *Theoretical Computer Science* 525 (2014), pp. 68–79.
- [183] Dmitry Kosolobov, “Online Square Detection”, in: *CoRR* abs/1411.2022 (2014), arXiv: 1411.2022.

- 
- [184] Chris-Andre Leimeister and Burkhard Morgenstern, “Kmacs: the k-mismatch average common substring approach to alignment-free sequence comparison”, in: *Bioinformatics* 30.14 (2014), pp. 2000–2008, DOI: 10.1093/bioinformatics/btu331.
- [185] Heng Li, “Fast construction of FM-index for long sequence reads”, in: *Bioinform.* 30.22 (2014), pp. 3274–3275, DOI: 10.1093/bioinformatics/btu541.
- [186] Frédéric Magniez, Claire Mathieu, and Ashwin Nayak, “Recognizing well-parenthesized expressions in the streaming model”, in: *SIAM Journal on Computing* 43.6 (2014), pp. 1880–1905, DOI: 10.1137/130926122.
- [187] René Rahn, David Weese, and Knut Reinert, “Journaled string tree - a scalable data structure for analyzing thousands of similar genomes on your laptop”, in: *Bioinformatics* 30.24 (2014), pp. 3499–3505, DOI: 10.1093/bioinformatics/btu438.
- [188] Jouni Sirén, Niko Välimäki, and Veli Mäkinen, “Indexing graphs for path queries with applications in genome research”, in: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 11.2 (2014), pp. 375–388.
- [189] Karyn Meltz Steinberg, Valerie A. Schneider, Tina A. Graves-Lindsay, Robert S. Fulton, Richa Agarwala, John Huddleston, Sergey A. Shiryev, Aleksandr Morgulis, Urvashi Surti, Wesley C. Warren, Deanna M. Church, Evan E. Eichler, and Richard K. Wilson, “Single haplotype assembly of the human genome from a hydatidiform mole”, in: *Genome Research* 24.12 (Nov. 2014), pp. 2066–2076, DOI: 10.1101/gr.180893.114.
- [190] Tom Murphy VII, “What, if anything, is epsilon?”, in: *Conference in Celebration of Harry Q. Bovik’s 26<sup>th</sup> Birthday*, 2014.
- [191] Amir Abboud, Arturs Backurs, and Virginia Vassilevska Williams, “Tight Hardness Results for LCS and Other Sequence Similarity Measures”, in: *FOCS’15*, IEEE Computer Society, 2015, pp. 59–78, DOI: 10.1109/FOCS.2015.14.
- [192] Amir Abboud, Richard Ryan Williams, and Huacheng Yu, “More Applications of the Polynomial Method to Algorithm Design”, in: *SODA’15*, 2015, pp. 218–230, DOI: 10.1137/1.9781611973730.17.
- [193] Amihoud Amir, Estrella Eisenberg, and A. Levy, “Approximate periodicity”, in: *Information and Computation* 241 (2015), pp. 215–226, ISSN: 0890-5401, DOI: 10.1016/j.ic.2015.02.004.
- [194] Alexandr Andoni and Ilya Razenshteyn, “Optimal Data-Dependent Hashing for Approximate Near Neighbors”, in: *STOC’15*, 2015, pp. 793–801, DOI: 10.1145/2746539.2746553.
- [195] Philip Bille, Gad M. Landau, Rajeev Raman, Kunihiko Sadakane, Srinivasa Rao Satti, and Oren Weimann, “Random Access to Grammar-Compressed Strings and Trees”, in: *SIAM J. Comput.* 44.3 (2015), pp. 513–539.
- [196] Inanç Birol, Justin Chu, Hamid Mohamadi, Shaun D Jackman, Karthika Raghavan, Benjamin P Vandervalk, Anthony Raymond, and René L Warren, “Spaced seed data structures for de novo assembly”, in: *International journal of genomics* 2015 (2015).
- [197] Karel Břinda, Maciej Sykulski, and Gregory Kucherov, “Spaced seeds improve k-mer-based metagenomic classification”, in: *Bioinformatics* 31.22 (2015), pp. 3584–3592.
- [198] Karl Bringmann and Marvin Kunnemann, “Quadratic conditional lower bounds for string problems and dynamic time warping”, in: *2015 IEEE Symposium on Foundations of Computer Science (FOCS)*, 2015, pp. 79–97, DOI: 10.1109/FOCS.2015.15.



- 
- [199] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya, “Dictionary matching in a stream”, in: *2015 European Symposium on Algorithms (ESA)*, vol. 9294, LNCS, 2015, pp. 361–372, DOI: 10.1007/978-3-662-48350-3\_31.
- [200] 1000 Genomes Project Consortium et al., “A global reference for human genetic variation”, in: *Nature* 526.7571 (2015), p. 68.
- [201] Antoine Deza, Frantisek Franek, and Adrien Thierry, “How many double squares can a string contain?”, in: *Discret. Appl. Math.* 180 (2015), pp. 52–69, DOI: 10.1016/j.dam.2014.08.016.
- [202] Marc Dupont and Pierre-François Marteau, “Coarse-DTW for Sparse Time Series Alignment”, in: *AALTD’15*, vol. 9785, LNCS, 2015, pp. 157–172, DOI: 10.1007/978-3-319-44412-3\_11.
- [203] Johannes Fischer, Stepan Holub, Tomohiro I, and Moshe Lewenstein, “Beyond the Runs Theorem”, in: *String Processing and Information Retrieval - 22<sup>nd</sup> International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings*, ed. by Costas S. Iliopoulos, Simon J. Puglisi, and Emine Yilmaz, vol. 9309, Lecture Notes in Computer Science, Springer, 2015, pp. 277–286, DOI: 10.1007/978-3-319-23826-5\_27.
- [204] Tomás Flouri, Emanuele Giaquinta, Kassian Kobert, and Esko Ukkonen, “Longest common substrings with  $k$  mismatches”, in: *Information Processing Letters* 115.6-8 (2015), pp. 643–647, DOI: 10.1016/j.ipl.2015.03.006.
- [205] Travis Gagie and Simon J Puglisi, “Searching and indexing genomic databases via kernelization”, in: *Frontiers in Bioengineering and Biotechnology* 3 (2015), p. 12.
- [206] Szymon Grabowski, “A note on the longest common substring with  $k$ -mismatches problem”, in: *Information Processing Letters* 115.6-8 (2015), pp. 640–642, DOI: 10.1016/j.ipl.2015.03.006.
- [207] Dmitry Kosolobov, “Lempel-Ziv Factorization May Be Harder Than Computing All Runs”, in: *32<sup>nd</sup> International Symposium on Theoretical Aspects of Computer Science (STACS 2015)*, ed. by Ernst W. Mayr and Nicolas Ollinger, vol. 30, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 582–593, ISBN: 978-3-939897-78-1, DOI: 10.4230/LIPIcs.STACS.2015.582.
- [208] Dmitry Kosolobov, “Online Detection of Repetitions with Backtracking”, in: *Combinatorial Pattern Matching - 26<sup>th</sup> Annual Symposium, CPM 2015, Ischia Island, Italy, June 29 - July 1, 2015, Proceedings*, ed. by Ferdinando Cicalese, Ely Porat, and Ugo Vaccaro, vol. 9133, Lecture Notes in Computer Science, Springer, 2015, pp. 295–306, DOI: 10.1007/978-3-319-19929-0\_25.
- [209] Ben Langmead, *Algorithms for DNA Sequencing: Base calling and sequencing errors*, 2015, URL: <https://www.youtube.com/watch?v=U4QnpclIJhM> (visited on 03/22/2021).
- [210] Kasper Green Larsen, J. Ian Munro, Jesper Sindahl Nielsen, and Sharma V. Thankachan, “On hardness of several string indexing problems”, in: *Theor. Comput. Sci.* 582 (2015), pp. 74–82.
- [211] Veli Mäkinen, Djamel Belazzougui, Fabio Cunial, and Alexandru I Tomescu, *Genome-scale algorithm design*, Cambridge University Press, 2015.
- [212] Arturs Backurs and Piotr Indyk, “Which Regular Expression Patterns Are Hard to Match?”, in: *2016 IEEE Symposium on Foundations of Computer Science (FOCS)*, IEEE Computer Society, 2016, pp. 457–466, DOI: 10.1109/FOCS.2016.56.

- 
- [213] Raphaël Clifford, Allyx Fontaine, Ely Porat, Benjamin Sach, and Tatiana Starikovskaya, “The  $k$ -mismatch problem revisited”, in: *2016 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2016, pp. 2039–2052, DOI: 10.1137/1.9781611974331.ch142.
  - [214] Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Ritu Kundu, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen, “Near-Optimal Computation of Runs over General Alphabet via Non-Crossing LCE Queries”, in: *String Processing and Information Retrieval - 23<sup>rd</sup> International Symposium, SPIRE 2016, Beppu, Japan, October 18-20, 2016, Proceedings*, ed. by Shunsuke Inenaga, Kunihiro Sadakane, and Tetsuya Sakai, vol. 9954, Lecture Notes in Computer Science, 2016, pp. 22–34, DOI: 10.1007/978-3-319-46049-9\_3.
  - [215] Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre, “Streaming property testing of visibly pushdown languages”, in: *2016 European Symposium on Algorithms (ESA)*, vol. 57, LIPIcs, 2016, 43:1–43:17, DOI: 10.4230/LIPIcs.ESA.2016.43.
  - [216] Moses Ganardi, Danny Huc, and Markus Lohrey, “Querying regular languages over sliding windows”, in: *2016 Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, vol. 65, LIPIcs, 2016, 18:1–18:14, DOI: 10.4230/LIPIcs.FSTTCS.2016.18.
  - [217] Paweł Gawrychowski, Tomasz Kociumaka, Wojciech Rytter, and Tomasz Walen, “Faster Longest Common Extension Queries in Strings over General Alphabets”, in: *27<sup>th</sup> Annual Symposium on Combinatorial Pattern Matching, CPM 2016, June 27-29, 2016, Tel Aviv, Israel*, ed. by Roberto Grossi and Moshe Lewenstein, vol. 54, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2016, 5:1–5:13, DOI: 10.4230/LIPIcs.CPM.2016.5.
  - [218] Manuel L Gonzalez-Garay, “Introduction to isoform sequencing using pacific biosciences technology (Iso-Seq)”, in: *Transcriptomics and gene regulation*, Springer, 2016, pp. 141–160.
  - [219] Tsvi Kopelowitz and Robert Krauthgamer, “Color-distance oracles and snippets”, in: *27<sup>th</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2016)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.
  - [220] Tsvi Kopelowitz, Seth Pettie, and Ely Porat, “Higher Lower Bounds from the 3SUM Conjecture”, in: *Proc. 27<sup>th</sup> SODA*, 2016, pp. 1272–1287.
  - [221] Dmitry Kosolobov, “Computing runs on a general alphabet”, in: *Inf. Process. Lett.* 116.3 (2016), pp. 241–244, DOI: 10.1016/j.ipl.2015.11.016.
  - [222] Dmitry Kosolobov, “Finding the leftmost critical factorization on unordered alphabet”, in: *Theor. Comput. Sci.* 636 (2016), pp. 56–65, DOI: 10.1016/j.tcs.2016.04.037.
  - [223] Veli Mäkinen and Gonzalo Navarro, “Compressed Text Indexing”, in: *Encyclopedia of Algorithms*, Springer, 2016, pp. 394–397.
  - [224] Yoshiaki Matsuoka, Takahiro Aoki, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, “Generalized pattern matching and periodicity under substring consistent equivalence relations”, in: *Theor. Comput. Sci.* 656 (2016), pp. 225–233, DOI: 10.1016/j.tcs.2016.02.017.
  - [225] Abdullah Mueen, Nikan Chavoshi, Noor Abu-El-Rub, Hossein Hamooni, and Amanda Minnich, “Awarp: fast warping distance for sparse time series”, in: *ICDM’16*, IEEE, 2016, pp. 350–359.

- 
- [226] Paul Muir, Shantao Li, Shaoke Lou, Daifeng Wang, Daniel J Spakowicz, Leonidas Salichos, Jing Zhang, George M Weinstock, Farren Isaacs, Joel Rozowsky, et al., “The real cost of sequencing: scaling computation to keep pace with data generation”, in: *Genome biology* 17.1 (2016), pp. 1–9.
- [227] Gonzalo Navarro, *Compact data structures: A practical approach*, Cambridge University Press, 2016.
- [228] Gonzalo Navarro and Sharma V. Thankachan, “Reporting consecutive substring occurrences under bounded gap constraints”, in: *Theoretical Computer Science* 638 (2016), Pattern Matching, Text Data Structures and Compression, pp. 108–111, ISSN: 0304-3975, DOI: 10.1016/j.tcs.2016.02.005.
- [229] Takaaki Nishimoto, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, “Fully Dynamic Data Structure for LCE Queries in Compressed Space”, in: *Proc. 41<sup>st</sup> MFCS*, vol. 58, 2016, 72:1–72:15.
- [230] Brian D Ondov, Todd J Treangen, Páll Melsted, Adam B Mallonee, Nicholas H Bergman, Sergey Koren, and Adam M Phillippy, “Mash: fast genome and metagenome distance estimation using MinHash”, in: *Genome biology* 17.1 (2016), pp. 1–14.
- [231] Sharma V. Thankachan, Alberto Apostolico, and Srinivas Aluru, “A Provably Efficient Algorithm for the  $k$ -Mismatch Average Common Substring Problem”, in: *Journal of Computational Biology* 23.6 (2016), pp. 472–482, DOI: 10.1089/cmb.2015.0235.
- [232] Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Alberto Apostolico, and Srinivas Aluru, “ALFRED: A Practical Method for Alignment-Free Distance Computation”, in: *Journal of Computational Biology* 23.6 (2016), pp. 452–460, DOI: 10.1089/cmb.2015.0217.
- [233] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann, “Fine-grained complexity of analyzing compressed data: Quantifying improvements over decompress-and-solve”, in: *2017 IEEE 58<sup>th</sup> Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2017, pp. 192–203.
- [234] Fatemeh Almodaresi, Prashant Pandey, and Rob Patro, “Rainbowfish: a succinct colored de Bruijn graph representation”, in: *17<sup>th</sup> International Workshop on Algorithms in Bioinformatics (WABI 2017)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.
- [235] Amihoud Amir, Alberto Apostolico, Travis Gagie, and Gad M. Landau, “String cadences”, in: *Theoretical Computer Science* 698 (2017), Algorithms, Strings and Theoretical Approaches in the Big Data Era (In Honor of the 60<sup>th</sup> Birthday of Professor Raffaele Giancarlo), pp. 4–8, ISSN: 0304-3975, DOI: 10.1016/j.tcs.2017.04.019.
- [236] Hideo Bannai, Tomohiro I, Shunsuke Inenaga, Yuto Nakashima, Masayuki Takeda, and Kazuya Tsuruta, “The “Runs” Theorem”, in: *SIAM J. Comput.* 46.5 (2017), pp. 1501–1514, DOI: 10.1137/15M1011032.
- [237] Karl Bringmann, “A near-linear pseudopolynomial time algorithm for subset sum”, in: *2017 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2017, pp. 1073–1084, DOI: 10.1137/1.9781611974782.69.
- [238] Karl Bringmann, Allan Gronlund, and Kasper Green Larsen, “A dichotomy for regular expression membership testing”, in: *2017 IEEE Symposium on Foundations of Computer Science (FOCS)*, 2017, pp. 307–318, DOI: 10.1109/FOCS.2017.36.

- 
- [239] Dirk D Dolle, Zhicheng Liu, Matthew Cotten, Jared T Simpson, Zamin Iqbal, Richard Durbin, Shane A McCarthy, and Thomas M Keane, “Using reference-free compressed data structures to analyze sequencing reads from thousands of human genomes”, in: *Genome research* 27.2 (2017), pp. 300–309.
  - [240] Anne Driemel and Francesco Silvestri, “Locality-Sensitive Hashing of Curves”, in: *SoCG’17*, vol. 77, LIPIcs, 2017, 37:1–37:16, ISBN: 978-3-95977-038-5, DOI: 10.4230/LIPIcs.SoCG.2017.37.
  - [241] Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou, “Streaming periodicity with mismatches”, in: *2017 International Conference on Approximation Algorithms for Combinatorial Optimization (APPROX)*, 2017, 42:1–42:21, DOI: 10.4230/LIPIcs.APPROX-RANDOM.2017.42.
  - [242] Travis Gagie, Giovanni Manzini, and Jouni Sirén, “Wheeler graphs: A framework for BWT-based data structures”, in: *Theoretical computer science* 698 (2017), pp. 67–78.
  - [243] Shay Golan and Ely Porat, “Real-time streaming multi-pattern search for constant alphabet”, in: *2017 European Symposium on Algorithms (ESA)*, vol. 107, LIPIcs, 2017, 41:1–41:15, DOI: 10.4230/LIPIcs.ESA.2017.41.
  - [244] Melissa Gymrek, “A genomic view of short tandem repeats”, in: *Current Opinion in Genetics and Development* 44 (2017), Molecular and genetic bases of disease, pp. 9–16, ISSN: 0959-437X, DOI: 10.1016/j.gde.2017.01.012.
  - [245] Stepan Holub, “Prefix frequency of lost positions”, in: *Theor. Comput. Sci.* 684 (2017), pp. 43–52, DOI: 10.1016/j.tcs.2017.01.026.
  - [246] Youngha Hwang and Saul B Gelfand, “Sparse dynamic time warping”, in: *MLDM’17*, vol. 10358, LNCS, 2017, pp. 163–175.
  - [247] Felipe A. Louza, Simon Gog, and Guilherme P. Telles, “Inducing enhanced suffix arrays for string collections”, in: *Theoretical Computer Science* 678 (2017), pp. 22–39, DOI: 10.1016/j.tcs.2017.03.039.
  - [248] Martin D Muggli, Alexander Bowe, Noelle R Noyes, Paul S Morley, Keith E Belk, Robert Raymond, Travis Gagie, Simon J Puglisi, and Christina Boucher, “Succinct colored de Bruijn graphs”, in: *Bioinformatics* 33.20 (2017), pp. 3181–3187.
  - [249] Tatiana Starikovskaya, “Communication and streaming complexity of approximate pattern matching”, in: *2017 Symposium on Combinatorial Pattern Matching (CPM)*, vol. 78, LIPIcs, 2017, 13:1–13:11, DOI: 10.4230/LIPIcs.CPM.2017.13.
  - [250] Sharma V. Thankachan, Sriram P. Chockalingam, Yongchao Liu, Ambujam Krishnan, and Srinivas Aluru, “A greedy alignment-free distance estimator for phylogenetic inference”, in: *BMC Bioinformatics* 18.8 (2017), p. 238, DOI: 10.1186/s12859-017-1658-0.
  - [251] Amir Abboud and Karl Bringmann, “Tighter connections between formula-SAT and shaving logs”, in: *2018 International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 107, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018, 8:1–8:18, DOI: 10.4230/LIPIcs.ICALP.2018.8.
  - [252] Mai Alzamel, Lorraine A. K. Ayad, Giulia Bernardini, Roberto Grossi, Costas S. Iliopoulos, Nadia Pisanti, Solon P. Pissis, and Giovanna Rosone, “Degenerate String Comparison and Applications”, in: *18<sup>th</sup> International Workshop on Algorithms in Bioinformatics (WABI 2018)*, ed. by Laxmi Parida and Esko Ukkonen, vol. 113, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018, 21:1–21:14, ISBN: 978-3-95977-082-8, DOI: 10.4230/LIPIcs.WABI.2018.21.

- 
- [253] Amihood Amir, Mika Amit, Gad M. Landau, and Dina Sokol, “Period recovery of strings over the Hamming and edit distances”, in: *Theoretical Computer Science* 710 (2018), pp. 2–18, ISSN: 0304-3975, DOI: 10.1016/j.tcs.2017.10.026.
- [254] Arturs Backurs and Piotr Indyk, “Edit Distance Cannot Be Computed in Strongly Sub-quadratic Time (Unless SETH is False)”, in: *SIAM Journal on Computing* 47.3 (2018), pp. 1087–1097, DOI: 10.1137/15M1053128.
- [255] Diptarka Chakraborty, Debarati Das, Elazar Goldenberg, Michal Koucký, and Michael E. Saks, “Approximating Edit Distance within Constant Factor in Truly Sub-Quadratic Time”, in: *FOCS’18*, 2018, pp. 979–990, DOI: 10.1109/FOCS.2018.00096.
- [256] Shubham Chandak, Kedar Tatwawadi, Idoia Ochoa, Mikel Hernaez, and Tsachy Weissman, “SPRING: a next-generation compressor for FASTQ data”, in: *Bioinformatics* 35.15 (2018), pp. 2674–2676, DOI: 10.1093/bioinformatics/bty1015.
- [257] Panagiotis Charalampopoulos, Maxime Crochemore, Costas S. Iliopoulos, Tomasz Kociumaka, Solon P. Pissis, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen, “Linear-Time Algorithm for Long LCF with  $k$  Mismatches”, in: *CPM’18*, 2018, 23:1–23:16, DOI: 10.4230/LIPIcs.CPM.2018.23.
- [258] Ioannis Z. Emiris and Ioannis Psarros, “Products of Euclidean Metrics and Applications to Proximity Questions among Curves”, in: *SoCG’18*, vol. 99, LIPIcs, 2018, 37:1–37:13, ISBN: 978-3-95977-066-8, DOI: 10.4230/LIPIcs.SoCG.2018.37.
- [259] Funda Ergün, Elena Grigorescu, Erfan Sadeqi Azer, and Samson Zhou, “Periodicity in data streams with wildcards”, in: *2018 International Computer Science Symposium in Russia (CSR)*, 2018, pp. 90–105, DOI: 10.1007/978-3-319-90530-3\_9.
- [260] Héctor Ferrada, Dominik Kempa, and Simon J. Puglisi, “Hybrid indexing revisited”, in: *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, SIAM, 2018, pp. 1–8.
- [261] Paolo Ferragina and Rossano Venturini, “Indexing Compressed Text”, in: *Encyclopedia of Database Systems (2<sup>nd</sup> ed.)* Springer, 2018.
- [262] Johannes Fischer, Tomohiro I, Dominik Köppl, and Kunihiro Sadakane, “Lempel-Ziv Factorization Powered by Space Efficient Suffix Trees”, in: *Algorithmica* 80.7 (2018), pp. 2048–2081, DOI: 10.1007/s00453-017-0333-1.
- [263] Travis Gagie, Gonzalo Navarro, and Nicola Prezza, “Optimal-Time Text Indexing in BWT-runs Bounded Space”, in: *Proc. 29<sup>th</sup> SODA*, 2018, pp. 1459–1477.
- [264] Moses Ganardi, Danny Hucce, Daniel König, Markus Lohrey, and Konstantinos Mamouras, “Automata theory on sliding windows”, in: *2018 Symposium on Theoretical Aspects of Computer Science (STACS)*, vol. 96, LIPIcs, 2018, 31:1–31:14, DOI: 10.4230/LIPIcs.STACS.2018.31.
- [265] Moses Ganardi, Danny Hucce, and Markus Lohrey, “Randomized sliding window algorithms for regular languages”, in: *2018 International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 107, LIPIcs, 2018, 127:1–127:13, DOI: 10.4230/LIPIcs.ICALP.2018.127.
- [266] Moses Ganardi, Danny Hucce, and Markus Lohrey, “Sliding window algorithms for regular languages”, in: *2018 International Conference on Language and Automata Theory and Applications (LATA)*, vol. 10792, 2018, pp. 26–35, DOI: 10.1007/978-3-319-77313-1\_2.

- 
- [267] Moses Ganardi, Artur Jeż, and Markus Lohrey, “Sliding windows over context-free languages”, in: *2018 International Symposium on Mathematical Foundations of Computer Science (MFCS)*, vol. 117, LIPIcs, 2018, 15:1–15:15, DOI: 10.4230/LIPIcs.MFCS.2018.15.
- [268] Erik Garrison, Jouni Sirén, Adam M Novak, Glenn Hickey, Jordan M Eizenga, Eric T Dawson, William Jones, Shilpa Garg, Charles Markello, Michael F Lin, et al., “Variation graph toolkit improves read mapping by representing genetic variation in the reference”, in: *Nature biotechnology* 36.9 (2018), pp. 875–879.
- [269] Pawel Gawrychowski, Adam Karczmarz, Tomasz Kociumaka, Jakub Lacki, and Piotr Sankowski, “Optimal Dynamic Strings”, in: *Proc. 29<sup>th</sup> SODA*, 2018, pp. 1509–1528.
- [270] Shay Golan, Tsvi Kopelowitz, and Ely Porat, “Towards optimal approximate streaming pattern matching by matching multiple patterns in multiple streams”, in: *2018 International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 107, LIPIcs, 2018, 65:1–65:16, DOI: 10.4230/LIPIcs.ICALP.2018.65.
- [271] Omer Gold and Micha Sharir, “Dynamic Time Warping and Geometric Edit Distance: Breaking the Quadratic Barrier”, in: *ACM Trans. Algorithms* 14.4 (2018), 50:1–50:17, DOI: 10.1145/3230734.
- [272] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny Shur, and Tomasz Walen, “String Periods in the Order-Preserving Model”, in: *35<sup>th</sup> Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, ed. by Rolf Niedermeier and Brigitte Vallée, vol. 96, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 38:1–38:16, ISBN: 978-3-95977-062-0, DOI: 10.4230/LIPIcs.STACS.2018.38.
- [273] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Juliusz Straszyński, Tomasz Waleń, and Wiktor Zuba, “Faster Recovery of Approximate Periods over Edit Distance”, in: *String Processing and Information Retrieval*, ed. by Travis Gagie, Alistair Moffat, Gonzalo Navarro, and Ernesto Cuadros-Vargas, Springer International Publishing, 2018, pp. 233–240, ISBN: 978-3-030-00479-8.
- [274] Heng Li, “Minimap2: pairwise alignment for nucleotide sequences”, in: *Bioinformatics* 34.18 (May 2018), pp. 3094–3100, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/bty191.
- [275] Oriol Pich, Ferran Muiños, Radhakrishnan Sabarinathan, Iker Reyes-Salazar, Abel Gonzalez-Perez, and Nuria Lopez-Bigas, “Somatic and germline mutation periodicity follow the orientation of the DNA minor groove around nucleosomes”, in: *Cell* 175.4 (2018), pp. 1074–1087.
- [276] Alberto Policriti and Nicola Prezza, “LZ77 computation based on the run-length encoded BWT”, in: *Algorithmica* 80.7 (2018), pp. 1986–2011.
- [277] Aviad Rubinstein, “Hardness of Approximate Nearest Neighbor Search”, in: *STOC’18*, 2018, pp. 1260–1268, DOI: 10.1145/3188745.3188916.
- [278] Daniel Valenzuela, Tuukka Norri, Niko Välimäki, Esa Pitkänen, and Veli Mäkinen, “Towards pan-genome read alignment to improve variation calling”, in: *BMC genomics* 19.2 (2018), pp. 123–130.
- [279] Jason Bentley, Daniel Gibney, and Sharma V Thankachan, “On the complexity of BWT-runs minimization via alphabet reordering”, in: *arXiv preprint arXiv:1911.03035* (2019).

- 
- [280] Christina Boucher, Travis Gagie, Alan Kuhnle, Ben Langmead, Giovanni Manzini, and Taher Mun, “Prefix-free parsing for building big BWTs”, in: *Algorithms for Molecular Biology* 14.1 (2019), pp. 1–15.
- [281] Vladimir Braverman, Moses Charikar, William Kuszmaul, David P. Woodruff, and Lin F. Yang, “The One-Way Communication Complexity of Dynamic Time Warping Distance”, in: *SoCG’19*, vol. 129, LIPIcs, 2019, 16:1–16:15, DOI: 10.4230/LIPIcs.SoCG.2019.16.
- [282] Raphaël Clifford, Paweł Gawrychowski, Tomasz Kociumaka, Daniel P Martin, and Przemysław Uznański, “RLE edit distance in near optimal time”, in: *arXiv preprint* (2019), eprint: arXiv:1905.01254.
- [283] Raphaël Clifford, Tomasz Kociumaka, and Ely Porat, “The streaming k-mismatch problem”, in: *2019 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019, pp. 1106–1125, DOI: 10.1137/1.9781611975482.68.
- [284] Lavinia Egidi, Felipe A Louza, Giovanni Manzini, and Guilherme P Telles, “External memory BWT and LCP computation for sequence collections with applications”, in: *Algorithms for Molecular Biology* 14.1 (2019), pp. 1–15.
- [285] Vincent Froese, Brijnesh J. Jain, Maciej Rymar, and Mathias Weller, “Fast Exact Dynamic Time Warping on Run-Length Encoded Time Series”, in: *CoRR* abs/1903.03003 (2019), arXiv: 1903.03003.
- [286] Moses Ganardi, Danny Hucke, Markus Lohrey, and Tatiana Starikovskaya, “Sliding window property testing for regular languages”, in: *2019 International Symposium on Algorithms and Computation (ISAAC)*, vol. 149, LIPIcs, 2019, 6:1–6:13, DOI: 10.4230/LIPIcs.ISAAC.2019.6.
- [287] Moses Ganardi, Artur Jez, and Markus Lohrey, “Balancing Straight-Line Programs”, in: *FOCS 2019*, IEEE Computer Society, 2019, pp. 1169–1183, DOI: 10.1109/FOCS.2019.00073.
- [288] Paweł Gawrychowski, Oleg Merkurev, Arseny M. Shur, and Przemysław Uznański, “Tight tradeoffs for real-time approximation of longest palindromes in streams”, in: *Algorithmica* 81.9 (2019), pp. 3630–3654, DOI: 10.1007/s00453-019-00591-8.
- [289] Paweł Gawrychowski, Jakub Radoszewski, and Tatiana Starikovskaya, “Quasi-periodicity in streams”, in: *2019 Symposium on Combinatorial Pattern Matching (CPM)*, vol. 128, LIPIcs, 2019, 22:1–22:14, DOI: 10.4230/LIPIcs.CPM.2019.22.
- [290] Paweł Gawrychowski and Tatiana Starikovskaya, “Streaming dictionary matching with mismatches”, in: *Algorithmica* (2019), pp. 1–21, DOI: 10.1007/s00453-021-00876-x.
- [291] Sara Giuliani, Zsuzsanna Lipták, Francesco Masillo, and Romeo Rizzi, “When a dollar makes a BWT”, in: *Theoretical Computer Science* (2019).
- [292] Shay Golan, Tsvi Kopelowitz, and Ely Porat, “Streaming Pattern Matching with  $d$  Wildcards”, in: *Algorithmica* 81.5 (2019), pp. 1988–2015, DOI: 10.1007/s00453-018-0521-7.
- [293] Youngha Hwang and Saul B. Gelfand, “Binary Sparse Dynamic Time Warping”, in: *MLDM’19*, ibai Publishing, 2019, pp. 748–759.
- [294] Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya, “Longest Common Substring with Approximately  $k$  Mismatches”, in: *Algorithmica* 81.6 (2019), pp. 2633–2652, DOI: 10.1007/s00453-019-00548-x.

- 
- [295] William Kuszmaul, “Dynamic Time Warping in Strongly Subquadratic Time: Algorithms for the Low-Distance Regime and Approximate Evaluation”, in: *ICALP’19*, vol. 132, LIPIcs, 2019, 80:1–80:15, ISBN: 978-3-95977-109-2, DOI: 10.4230/LIPIcs.ICALP.2019.80.
- [296] William Kuszmaul, “Dynamic Time Warping in Strongly Subquadratic Time: Algorithms for the Low-Distance Regime and Approximate Evaluation”, in: *CoRR* abs/1904.09690 (2019), DOI: 10.48550/ARXIV.1904.09690, eprint: 1904.09690.
- [297] Medhat Mahmoud, Nastassia Gobet, Diana Ivette Cruz-Dávalos, Ninon Mounier, Christophe Dessimoz, and Fritz J Sedlazeck, “Structural variant calling: the long and the short of it”, in: *Genome biology* 20.1 (2019), pp. 1–14.
- [298] Guillaume Marçais, Brad Solomon, Rob Patro, and Carl Kingsford, “Sketching and sub-linear data structures in genomics”, in: *Annual Review of Biomedical Data Science* 2 (2019), pp. 93–118.
- [299] Oleg Merkurev and Arseny M. Shur, “Searching long repeats in streams”, in: *2019 Symposium on Combinatorial Pattern Matching (CPM)*, vol. 128, LIPIcs, 2019, 31:1–31:14, DOI: 10.4230/LIPIcs.CPM.2019.31.
- [300] Oleg Merkurev and Arseny M. Shur, “Searching runs in streams”, in: *2019 Symposium on String Processing and Information Retrieval (SPIRE)*, vol. 11811, LNCS, 2019, pp. 203–220, DOI: 10.1007/978-3-030-32686-9\_15.
- [301] Will PM Rowe, “When the levee breaks: a practical guide to sketching algorithms for processing the flood of genomic data”, in: *Genome biology* 20.1 (2019), pp. 1–12.
- [302] Amir Abboud, Arturs Backurs, Karl Bringmann, and Marvin Künnemann, “Impossibility Results for Grammar-Compressed Linear Algebra”, in: *Proc. 34<sup>th</sup> NeurIPS*, 2020, pp. 8810–8823.
- [303] Jarno Alanko, Giovanna D’Agostino, Alberto Policriti, and Nicola Prezza, “Regular Languages Meet Prefix Sorting”, in: *Proceedings of the Thirty-First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA ’20, Society for Industrial and Applied Mathematics, 2020, pp. 911–930.
- [304] Shanika L Amarasinghe, Shian Su, Xueyi Dong, Luke Zappia, Matthew E Ritchie, and Quentin Gouil, “Opportunities and challenges in long-read sequencing data analysis”, in: *Genome biology* 21.1 (2020), pp. 1–16.
- [305] Alexandr Andoni, “Simple Constant-Factor Approximation to Edit Distance”, in: *CoRR* (2020).
- [306] Alexandr Andoni and Negev Shekel Nosatzki, “Edit distance in near-linear time: It’s a constant factor”, in: *2020 IEEE 61<sup>st</sup> Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2020, pp. 990–1001.
- [307] Uwe Baier, Thomas Büchler, Enno Ohlebusch, and Pascal Weber, “Edge Minimization in de Bruijn Graphs”, in: *Data Compression Conference, DCC 2020, Snowbird, UT, USA, March 24–27, 2020*, ed. by Ali Bilgin, Michael W. Marcellin, Joan Serra-Sagristà, and James A. Storer, IEEE, 2020, pp. 223–232, DOI: 10.1109/DCC47342.2020.00030.
- [308] Hideo Bannai, Travis Gagie, and I Tomohiro, “Refining the  $r$ -index”, in: *Theoretical Computer Science* 812 (2020), pp. 96–108.
- [309] Jason W Bentley, Daniel Gibney, and Sharma V Thankachan, “On the Complexity of BWT-Runs Minimization via Alphabet Reordering”, in: *28<sup>th</sup> Annual European Symposium on Algorithms (ESA 2020)*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.



- 
- [310] Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering, “Hide and Mine in Strings: Hardness and Algorithms”, in: *2020 IEEE International Conference on Data Mining (ICDM 2020)*, 2020, pp. 924–929, DOI: 10.1109/ICDM50108.2020.00103.
- [311] Joshua Brakensiek and Aviad Rubinfeld, “Constant-factor approximation of near-linear edit distance in near-linear time”, in: *Proceedings of the 52<sup>nd</sup> Annual ACM SIGACT Symposium on Theory of Computing*, 2020, pp. 685–698.
- [312] Lavinia Egidi and Giovanni Manzini, “Lightweight merging of compressed indices based on BWT variants”, in: *Theoretical Computer Science* 812 (2020), pp. 214–229.
- [313] Travis Gagie, Gonzalo Navarro, and Nicola Prezza, “Fully functional suffix trees and optimal text searching in BWT-runs bounded space”, in: *Journal of the ACM (JACM)* 67.1 (2020), pp. 1–54.
- [314] Pawel Gawrychowski, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen, “Universal reconstruction of a string”, in: *Theoretical Computer Science* 812 (2020), In memoriam Danny Breslauer (1968-2017), pp. 174–186, ISSN: 0304-3975, DOI: 10.1016/j.tcs.2019.10.027.
- [315] Shay Golan, Tomasz Kociumaka, Tsvi Kopelowitz, and Ely Porat, “The streaming k-mismatch problem: tradeoffs between space and total time”, in: *2020 Symposium on Combinatorial Pattern Matching (CPM)*, vol. 161, LIPIcs, 2020, 15:1–15:15, DOI: 10.4230/LIPIcs.CPM.2020.15.
- [316] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny Shur, and Tomasz Waleń, “String periods in the order-preserving model”, in: *Information and Computation* 270 (2020), p. 104463, DOI: 10.1016/j.ic.2019.104463.
- [317] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya, “Approximating Longest Common Substring with k mismatches: Theory and Practice”, in: *31<sup>st</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2020), June 17-19, 2020, Copenhagen, Denmark*, ed. by Inge Li Gørtz and Oren Weimann, vol. 161, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 16:1–16:15, DOI: 10.4230/LIPIcs.CPM.2020.16.
- [318] Michal Koucký and Michael Saks, “Constant factor approximations to edit distance on far input pairs in nearly linear time”, in: *Proceedings of the 52<sup>nd</sup> Annual ACM SIGACT Symposium on Theory of Computing*, 2020, pp. 699–712.
- [319] Alan Kuhnle, Taher Mun, Christina Boucher, Travis Gagie, Ben Langmead, and Giovanni Manzini, “Efficient construction of a complete index for pan-genomics read alignment”, in: *Journal of Computational Biology* 27.4 (2020), pp. 500–513.
- [320] Akihiro Nishi, Yuto Nakashima, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda, “Towards Efficient Interactive Computation of Dynamic Time Warping Distance”, in: *SPIRE’20*, vol. 12303, LNCS, 2020, pp. 27–41, DOI: 10.1007/978-3-030-59212-7\_3.
- [321] Takaaki Nishimoto and Yasuo Tabei, “Faster Queries on BWT-runs Compressed Indices”, in: *arXiv preprint arXiv:2006.05104* (2020).
- [322] Enrico Petrucci, Laurent Noé, Cinzia Pizzi, and Matteo Comin, “Iterative spaced seed hashing: closing the gap between spaced seed hashing and k-mer hashing”, in: *Journal of Computational Biology* 27.2 (2020), pp. 223–233.

- 
- [323] Jakub Radoszewski and Tatiana Starikovskaya, “Streaming  $k$ -mismatch with error correcting and applications”, in: *Journal of Information and Computation* 271 (2020), p. 104513, DOI: 10.1016/j.ic.2019.104513.
  - [324] Yoshifumi Sakai and Shunsuke Inenaga, “A Reduction of the Dynamic Time Warping Distance to the Longest Increasing Subsequence Length”, in: *ISAAC’20*, vol. 181, LIPIcs, 2020, 6:1–6:16, DOI: 10.4230/LIPIcs.ISAAC.2020.6.
  - [325] Philipp Schepper, “Fine-grained complexity of regular expression pattern matching and membership”, in: *2020 European Symposium on Algorithms (ESA)*, vol. 173, Leibniz International Proceedings in Informatics (LIPIcs), 2020, 80:1–80:20, DOI: 10.4230/LIPIcs.ESA.2020.80.
  - [326] Adrien Thierry, “A proof that a word of length  $n$  has less than  $1.5n$  distinct squares”, in: *CoRR* abs/2001.02996 (2020), arXiv: 2001.02996.
  - [327] Gabriel Bathie and Tatiana Starikovskaya, “Property testing of regular languages with applications to streaming property testing of visibly pushdown languages”, in: *2021 International Colloquium on Automata, Languages, and Programming (ICALP)*, vol. 198, Leibniz International Proceedings in Informatics (LIPIcs), 2021, 119:1–119:17, DOI: 10.4230/LIPIcs.ICALP.2021.119.
  - [328] Djamal Belazzougui, Dmitry Kosolobov, Simon J. Puglisi, and Rajeev Raman, “Weighted Ancestors in Suffix Trees Revisited”, in: *32<sup>nd</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2021)*, vol. 191, Leibniz International Proceedings in Informatics (LIPIcs), 2021, 8:1–8:15, ISBN: 978-3-95977-186-3, DOI: 10.4230/LIPIcs.CPM.2021.8.
  - [329] Anders Roy Christiansen, Mikko Berggren Ettienne, Tomasz Kociumaka, Gonzalo Navarro, and Nicola Prezza, “Optimal-Time Dictionary-Compressed Indexes”, in: *ACM Trans. Algorithms* 17.1 (2021), 8:1–8:39, DOI: 10.1145/3426473.
  - [330] Francisco Claude, Gonzalo Navarro, and Alejandro Pacheco, “Grammar-compressed indexes with logarithmic search time”, in: *J. Comput. Syst. Sci.* 118 (2021), pp. 53–74.
  - [331] Clara Delahaye and Jacques Nicolas, “Sequencing DNA with nanopores: Troubles and biases”, in: *PloS one* 16.10 (2021), e0257521.
  - [332] Diego Díaz-Domínguez and Gonzalo Navarro, “A grammar compressor for collections of reads with applications to the construction of the BWT”, in: *2021 Data Compression Conference (DCC)*, IEEE, 2021, pp. 83–92.
  - [333] Diego Díaz-Domínguez, Gonzalo Navarro, and Alejandro Pacheco, “An LMS-Based Grammar Self-index with Local Consistency Properties”, in: *Proc. 28<sup>th</sup> SPIRE*, 2021.
  - [334] Jonas Ellert and Johannes Fischer, “Linear Time Runs Over General Ordered Alphabets”, in: *48<sup>th</sup> International Colloquium on Automata, Languages, and Programming (ICALP 2021)*, ed. by Nikhil Bansal, Emanuela Merelli, and James Worrell, vol. 198, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 63:1–63:16, ISBN: 978-3-95977-195-5, DOI: 10.4230/LIPIcs.ICALP.2021.63.
  - [335] Travis Gagie, Garance Gourdel, and Giovanni Manzini, “Compressing and Indexing Aligned Readsets”, in: *21<sup>st</sup> International Workshop on Algorithms in Bioinformatics, (WABI 2021), August 2-4, 2021, Virtual Conference*, ed. by Alessandra Carbone and Mohammed El-Kebir, vol. 201, LIPIcs, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021, 13:1–13:21, DOI: 10.4230/LIPIcs.WABI.2021.13.

- 
- [336] Daniel Gibney and Sharma V. Thankachan, “Text Indexing for Regular Expression Matching”, in: *Algorithms* 14.5 (2021), p. 133.
- [337] Sara Giuliani, Shunsuke Inenaga, Zsuzsanna Lipták, Nicola Prezza, Marinella Sciortino, and Anna Toffanello, “Novel Results on the Number of Runs of the Burrows-Wheeler Transform”, in: *SOFSEM 2021: Theory and Practice of Computer Science*, ed. by Tomáš Bureš, Riccardo Dondi, Johann Gamper, Giovanna Guerrini, Tomasz Jurdziński, Claus Pahl, Florian Sikora, and Prudence W.H. Wong, Springer International Publishing, 2021, pp. 249–262, ISBN: 978-3-030-67731-2.
- [338] Yao-Ting Huang, Po-Yu Liu, and Pei-Wen Shih, “Homopolish: a method for the removal of systematic errors in nanopore sequencing by homologous polishing”, in: *Genome Biology* 22.1 (Mar. 31, 2021), p. 95, ISSN: 1474-760X, DOI: 10.1186/s13059-021-02282-6, (visited on 08/23/2021).
- [339] Costas S Iliopoulos, Ritu Kundu, and Solon P Pissis, “Efficient pattern matching in elastic-degenerate strings”, in: *Information and Computation* 279 (2021), p. 104616.
- [340] Alice M Kaye and Wyeth W Wasserman, “The Genome Atlas: Navigating a New Era of Reference Genomes”, in: *Trends in Genetics* (2021).
- [341] William Kuszmaul, “Binary Dynamic Time Warping in Linear Time”, in: *CoRR* abs/2101.01108 (2021), arXiv: 2101.01108.
- [342] Antoine Pietri, “Organizing the graph of public software development for large-scale mining. (Organisation du graphe de développement logiciel pour l’analyse à grande échelle)”, PhD thesis, University of Paris, France, 2021, URL: <https://tel.archives-ouvertes.fr/tel-03515795>.
- [343] Nicola Prezza, “On Locating Paths in Compressed Tries”, in: *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2021, pp. 744–760.
- [344] Nicola Prezza, “Subpath Queries on Compressed Graphs: A Survey”, in: *Algorithms* 14.1 (2021), p. 14.
- [345] Marina Svetec Miklenić and Ivan Krešimir Svetec, “Palindromes in DNA – a risk for genome stability and implications in cancer”, in: *International Journal of Molecular Sciences* 22.6 (2021), p. 2840.
- [346] Mahdi Belbasi, Antonio Blanca, Robert S Harris, David Koslicki, and Paul Medvedev, “The minimizer Jaccard estimator is biased and inconsistent”, in: *Bioinformatics* 38 (June 2022), pp. i169–i176, ISSN: 1367-4803, DOI: 10.1093/bioinformatics/btac244.
- [347] Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P. Pissis, Eva Rotenberg, and Teresa Anna Steiner, “Gapped String Indexing in Subquadratic Space and Sublinear Query Time”, in: *CoRR* abs/2211.16860 (2022), DOI: 10.48550/arXiv.2211.16860, arXiv: 2211.16860.
- [348] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, Eva Rotenberg, and Teresa Anna Steiner, “String indexing for top- $k$  close consecutive occurrences”, in: *Theor. Comput. Sci.* 927 (2022), pp. 133–147, DOI: 10.1016/j.tcs.2022.06.004.
- [349] Philip Bille, Inge Li Gørtz, Max Rishøj Pedersen, and Teresa Anna Steiner, “Gapped indexing for consecutive occurrences”, in: *Algorithmica* (2022), pp. 1–23.
- [350] Srecko Brlek and Shuo Li, “On the number of squares in a finite word”, in: *CoRR* abs/2204.10204 (2022), arXiv: 2204.10204.

- 
- [351] Davide Cenzato and Zsuzsanna Lipták, “A Theoretical and Experimental Analysis of BWT Variants for String Collections”, in: *33<sup>rd</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2022)*, ed. by Hideo Bannai and Jan Holub, vol. 223, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 25:1–25:18, ISBN: 978-3-95977-234-1, DOI: 10.4230/LIPIcs.CPM.2022.25.
  - [352] Bartłomiej Dudek, Pawel Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya, “Streaming Regular Expression Membership and Pattern Matching”, in: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022), Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022*, ed. by Joseph (Seffi) Naor and Niv Buchbinder, SIAM, 2022, pp. 670–694, DOI: 10.1137/1.9781611977073.30.
  - [353] Moses Ganardi and Pawel Gawrychowski, “Pattern Matching on Grammar-Compressed Strings in Linear Time”, in: *SODA 2022*, 2022, pp. 2833–2846, DOI: 10.1137/1.9781611977073.110.
  - [354] Arun Ganesh, Tomasz Kociumaka, Andrea Lincoln, and Barna Saha, “How compression and approximation affect efficiency in string distance measures”, in: *Proceedings of the 2022 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2022, pp. 2867–2919.
  - [355] Garance Gourdel, Anne Driemel, Pierre Peterlongo, and Tatiana Starikovskaya, “Pattern Matching Under DTW Distance”, in: *String Processing and Information Retrieval - 29<sup>th</sup> International Symposium, SPIRE 2022, Concepción, Chile, November 8-10, 2022, Proceedings*, ed. by Diego Arroyuelo and Barbara Poblete, vol. 13617, Lecture Notes in Computer Science, Springer, 2022, pp. 315–330, DOI: 10.1007/978-3-031-20643-6\_23.
  - [356] Tomasz Kociumaka, Ely Porat, and Tatiana Starikovskaya, “Small-space and streaming pattern matching with  $k$  edits”, in: *2021 IEEE 62nd Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2022, pp. 885–896.
  - [357] Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, and Tomasz Walen, “A periodicity lemma for partial words”, in: *Inf. Comput.* 283 (2022), p. 104677, DOI: 10.1016/j.ic.2020.104677.
  - [358] Shuo Li, Jakub Pachocki, and Jakub Radoszewski, “A note on the maximum number of  $k$ -powers in a finite word”, in: *CoRR* abs/2205.10156 (2022), arXiv: 2205.10156.
  - [359] Oleg Merkurev and Arseny M. Shur, “Computing The Maximum Exponent in a Stream”, in: *Algorithmica* 84.3 (2022), pp. 742–756, DOI: 10.1007/s00453-021-00883-y.
  - [360] Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering, “Hide and Mine in Strings: Hardness, Algorithms, and Experiments”, in: *IEEE Transactions on Knowledge and Data Engineering* 35.6 (2023), pp. 5948–5963, DOI: 10.1109/TKDE.2022.3158063.
  - [361] Itai Boneh, Shay Golan, Shay Mozes, and Oren Weimann, “Near-Optimal Dynamic Time Warping on Run-Length Encoded Strings”, in: *arXiv preprint arXiv:2302.06252* (2023).
  - [362] Davide Cenzato, Veronica Guerrini, Zsuzsanna Lipták, and Giovanna Rosone, “Computing the optimal BWT of very large string collections”, in: *2023 Data Compression Conference (DCC)*, IEEE, 2023, pp. 71–80.

- 
- [363] Jonas Ellert, Pawel Gawrychowski, and Garance Gourdel, “Optimal Square Detection Over General Alphabets”, in: *Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, ed. by Nikhil Bansal and Viswanath Nagarajan, SIAM, 2023, pp. 5220–5242, DOI: 10.1137/1.9781611977554.ch189.
- [364] Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner, “Compressed Indexing for Consecutive Occurrences”, in: *CoRR* abs/2304.00887 (2023), Accepted to CPM 2023, DOI: 10.48550/arXiv.2304.00887, arXiv: 2304.00887.
- [365] Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner, “Compressed Indexing for Consecutive Occurrences”, in: *34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023)*, June 26-28, 2023, Marne-la-Vallée, France, ed. by Laurent Bulteau and Zsuzsanna Lipták, vol. 259, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 12:1–12:22, DOI: 10.4230/LIPIcs.CPM.2023.12.
- [366] <https://web.archive.org/web/20230228010327/https://dumps.wikimedia.org/enwiki/20221201/>.
- [367] <https://www.softwareheritage.org/>.
- [368] <https://www.polytechnique-insights.com/en/columns/economy/source-code-building-a-universal-software-archive/>.
- [369] <https://web.archive.org/>.
- [370] <https://www.genomicsengland.co.uk/initiatives/100000-genomes-project>.
- [371] <https://www.ebi.ac.uk/ena/browser/about/statistics>.
- [372] <https://www.ncbi.nlm.nih.gov/sra/docs/sragrowth/>.
- [373] <https://web.archive.org/web/20211130235813/http://archive.org/>.
- [374] [https://en.wikipedia.org/wiki/Table\\_canon](https://en.wikipedia.org/wiki/Table_canon).
- [375] [https://en.wikipedia.org/wiki/Crab\\_canon](https://en.wikipedia.org/wiki/Crab_canon).
- [376] <https://kamimrcht.github.io/webpage/sketch.html>.
- [377] <https://github.com/spotify/annoy>.
- [378] LeetCode, *Problem 139. Word break*, <https://leetcode.com/problems/word-break/>.

# List of Publications

---

- [272] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny Shur, and Tomasz Walen, “String Periods in the Order-Preserving Model”, in: *35<sup>th</sup> Symposium on Theoretical Aspects of Computer Science (STACS 2018)*, ed. by Rolf Niedermeier and Brigitte Vallée, vol. 96, Leibniz International Proceedings in Informatics (LIPIcs), Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018, 38:1–38:16, ISBN: 978-3-95977-062-0, DOI: 10.4230/LIPIcs.STACS.2018.38.
- [310] Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering, “Hide and Mine in Strings: Hardness and Algorithms”, in: *2020 IEEE International Conference on Data Mining (ICDM 2020)*, 2020, pp. 924–929, DOI: 10.1109/ICDM50108.2020.00103.
- [316] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, Wojciech Rytter, Arseny Shur, and Tomasz Waleń, “String periods in the order-preserving model”, in: *Information and Computation* 270 (2020), p. 104463, DOI: 10.1016/j.ic.2019.104463.
- [317] Garance Gourdel, Tomasz Kociumaka, Jakub Radoszewski, and Tatiana Starikovskaya, “Approximating Longest Common Substring with k mismatches: Theory and Practice”, in: *31<sup>st</sup> Annual Symposium on Combinatorial Pattern Matching (CPM 2020)*, June 17–19, 2020, Copenhagen, Denmark, ed. by Inge Li Gørtz and Oren Weimann, vol. 161, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020, 16:1–16:15, DOI: 10.4230/LIPIcs.CPM.2020.16.
- [335] Travis Gagie, Garance Gourdel, and Giovanni Manzini, “Compressing and Indexing Aligned Readsets”, in: *21<sup>st</sup> International Workshop on Algorithms in Bioinformatics (WABI 2021)*, August 2–4, 2021, Virtual Conference, ed. by Alessandra Carbone and Mohammed El-Kebir, vol. 201, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, 13:1–13:21, DOI: 10.4230/LIPIcs.WABI.2021.13.
- [352] Bartłomiej Dudek, Paweł Gawrychowski, Garance Gourdel, and Tatiana Starikovskaya, “Streaming Regular Expression Membership and Pattern Matching”, in: *Proceedings of the 2022 ACM-SIAM Symposium on Discrete Algorithms (SODA 2022)*, Virtual Conference / Alexandria, VA, USA, January 9 - 12, 2022, ed. by Joseph (Seffi) Naor and Niv Buchbinder, SIAM, 2022, pp. 670–694, DOI: 10.1137/1.9781611977073.30.
- [355] Garance Gourdel, Anne Driemel, Pierre Peterlongo, and Tatiana Starikovskaya, “Pattern Matching Under DTW Distance”, in: *String Processing and Information Retrieval - 29<sup>th</sup> International Symposium, SPIRE 2022, Concepción, Chile, November 8–10, 2022, Proceedings*, ed. by Diego Arroyuelo and Barbara Poblete, vol. 13617, Lecture Notes in Computer Science, Springer, 2022, pp. 315–330, DOI: 10.1007/978-3-031-20643-6\_23.
- [360] Giulia Bernardini, Alessio Conte, Garance Gourdel, Roberto Grossi, Grigorios Loukides, Nadia Pisanti, Solon P. Pissis, Giulia Punzi, Leen Stougie, and Michelle Sweering, “Hide and Mine in Strings: Hardness, Algorithms, and Experiments”, in: *IEEE Transactions on Knowledge and Data Engineering* 35.6 (2023), pp. 5948–5963, DOI: 10.1109/TKDE.2022.3158063.

- 
- [363] Jonas Ellert, Pawel Gawrychowski, and Garance Gourdel, “Optimal Square Detection Over General Alphabets”, *in: Proceedings of the 2023 ACM-SIAM Symposium on Discrete Algorithms (SODA 2023)*, ed. by Nikhil Bansal and Viswanath Nagarajan, SIAM, 2023, pp. 5220–5242, DOI: 10.1137/1.9781611977554.ch189.
- [365] Pawel Gawrychowski, Garance Gourdel, Tatiana Starikovskaya, and Teresa Anna Steiner, “Compressed Indexing for Consecutive Occurrences”, *in: 34th Annual Symposium on Combinatorial Pattern Matching (CPM 2023), June 26-28, 2023, Marne-la-Vallée, France*, ed. by Laurent Bulteau and Zsuzsanna Lipták, vol. 259, LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023, 12:1–12:22, DOI: 10.4230/LIPIcs.CPM.2023.12.

# The Cat and the Yarn

Inspired by the book "Le chat au pays des nombres" by Ivar Ekeland and John O'Brien, which presents the Hilbert's paradox of the Grand Hotel in a kids book format, I decided (on my own time) to sketch a comic inspired by this thesis. I dedicate it to Hana, Mira, and all the little girls that may take an interest in computer science one day.







## **Titre :** Approches Basé sur les Sketches pour le Traitement massif de Chaînes de Caractères

**Mot clés :** Chaîne de Caractères, Algorithmes, Structures de données, Flots, Recherche Approchée

**Résumé :** La simplicité des chaînes de caractères rendent leur traitement crucial pour de nombreuses applications, telles que la bio-informatique, la recherche d'informations et la cybersécurité. Le problème de la recherche exacte d'un motif a naturellement été largement étudié [94], cependant, de nombreuses applications nécessitent également des requêtes plus complexes. De plus, dans ces domaines applicatifs, la quantité de données à traiter augmente à une vitesse stupéfiante [226], et les complexités des requêtes ne permettent pas toujours de passer à l'échelle. Dans cette thèse, nous proposons plusieurs algorithmes efficaces en temps et en espace pour divers problèmes sur les chaînes de caractères, en nous appuyant sur des « sketches » : des compressions (avec ou sans perte) qui ne conservent que les caractéristiques essentielles de l'entrée pour répondre à une requête précise.

Dans la première partie de cette thèse, nous étudions des requêtes complexes telles que la recherche par expressions régulières, la recherche de motifs consécutifs avec espacement et la détection de carrés. Pour la recherche d'expressions régulières, nous présentons un algorithme utilisant peu d'espace dans le modèle de flot de données (« streaming ») : les caractères du texte arrivent un par un, et nous ne pouvons accéder aux anciens que si nous les avons stockés explicitement. Ensuite, nous étudions la recherche de motifs consécutifs avec espacement, un type de requête plus simple, où étant donnés deux motifs  $P_1$ ,  $P_2$  et un inter-

valle  $[a, b]$ , il faut renvoyer toutes les occurrences consécutives (sans autres occurrences des motifs entre les deux) de  $P_1$  suivies de  $P_2$  espacées d'une distance comprise entre  $a$  et  $b$ . Nous étudions ce problème sous plusieurs angles : l'indexation compressée et la recherche de motifs dans un texte compressé. Motivés par l'importance de la périodicité, nous étudions ensuite la détection de carrés pour alphabets sans ordres (le cadre le plus abstrait dans lequel les carrés peuvent être définis). Nous fournissons un algorithme optimal et répondons à une question ouverte posée par Main et Lorentz [27] en 1984.

La seconde partie de cette thèse propose quelques utilisations d'approximations pour aider à passer à l'échelle sur des grandes quantités de données, en particulier avec application à la bio-informatique. Nous étudions tout d'abord la recherche approximative de motifs, où nous devons rapporter toutes les occurrences à une distance au plus égale à  $k$  pour une mesure de similarité donnée. Nous fournissons des algorithmes paramétrés efficaces pour calculer la longueur de la plus longue sous-chaîne commune avec environ  $k$  différences, puis pour permettre la recherche de motifs apparaissant avec une distance de « dynamic time warping » au plus  $k$ . Enfin, nous proposons un index compressé pour des collections de lectures de séquençage. Cet index tire parti d'alignements sur un génome assemblé pour améliorer la compression, mais l'index est approximatif car il peut renvoyer des faux positifs lors de ses requêtes.

## **Title:** Sketch-Based Approaches to Process Massive String Data

**Keywords:** Strings, Algorithms, Data Structure, Streaming, Approximate Search

**Abstract:** The simplicity of strings and their impactful usage puts their processing at the heart of many applications, including Bioinformatics, Information Retrieval, and Cybersecurity. Exact pattern matching has been extensively studied [94] as the most natural problem, however, many applications also need more complex queries. Additionally, in all those application fields, the quantity of information to process has been increasing at such a staggering rate [226], that obtaining scalable algorithms is difficult. In this thesis we contribute multiple space- and time-efficient algorithms for various string problems, by relying on sketches: compressions (lossless or lossy) that only keep the essential characteristic of the input needed to answer a given query.

In the first part of this thesis, we study complex queries such as regular expressions search, gapped consecutive matching, and square detection. For regular expression search, we provide a space-efficient algorithm in the streaming model: characters of the text arrive one at a time, and we can only access past characters if we explicitly store them. Next, gapped consecutive matching is a simpler type of query where, given two patterns  $P_1$ ,  $P_2$  and a range  $[a, b]$ , one must report all consecutive occurrences of  $P_1$  followed

by  $P_2$  separated by a distance in  $[a, b]$ . We study this problem in two settings: compressed indexing and pattern matching on a compressed text. Motivated by the importance of periodicity detection, next, we investigate square detection for general alphabets (the most abstract setting where squares can be defined). We give an optimal algorithm which answers an open question asked by Main and Lorentz [27] in 1984.

The second part of this thesis proposes several ways to use approximation toward scaling up to large amounts of data in diverse applications including Bioinformatics. We first study approximate matching, where we must report all occurrences at distance at most  $k$  for a given similarity measure. We provide efficient parametrized algorithms for computing the length of the longest common substring with approximately  $k$  mismatches and to compute all positions of a text where a pattern occurs with dynamic time warping distance at most  $k$ . Finally, we propose a compressed index for redundant collections of next-generation sequencing reads, which takes advantage of alignments to an assembled genome to improve the overall compression but can incur false positive occurrences.