ED 386 : Sciences Mathématiques de Paris Centre

Doctorat

THÈSE

pour obtenir le grade de docteur délivré par

Université Paris Diderot

Discipline "Informatique"

présentée et soutenue publiquement par

Nathanaël François

le 2 Septembre 2015

Algorithmes et Bornes Inférieures pour diverses variantes du Modèle de Streaming
Algorithms and Lower Bounds for variants of the Streaming Model

Directeur de thèse : **Frédéric Magniez**

Jury

Frédéric Magniez,	CNRS, Université Paris Diderot	Directeur
Claire Mathieu,	CNRS, École Normale Supérieure	Rapportrice
Dana Ron,	Tel Aviv University	Rapportrice (absente)
Benjamin Doerr,	École Polytechnique	Examinateur
Christoph Dürr,	CNRS, Université Pierre et Marie Curie	Examinateur
Omar Fawzi,	École Normale Supérieure de Lyon	Examinateur
Ashwin Nayak,	University of Waterloo	Examinateur

T
H
E
S
E

Remerciements

Tout d'abord je voudrais remercier Frédéric de m'avoir apporté sa sagesse et son soutien depuis quatre ans et demi qu'il m'encadre. Il a su me laisser de l'espace pour m'épanouir tout en n'hésitant pas à me stimuler quand c'était nécessaire. Je ne prétendrais pas que tout a toujours été parfait: il y a notamment eu des moments de frustration quand il me demandait de changer la structure d'une preuve trois fois de suite pour finalement la remettre dans son état d'origine, mais je pense que son insistance sur la manière d'expliquer était nécessaire, et a fini par payer. Ah, et puis je ne suis toujours pas convaincu qu'il n'a pas installé à la porte de son bureau une sorte de barrière magique qui neuf fois sur dix rend fausses les preuves quand on y entre.

Je voudrais aussi remercier Benjamin Doerr, Christoph Durr, Omar Fawzi et Ashwin Nayak d'avoir accepté de faire partie de mon jury, ainsi que Claire Mathieu et Dana Ron d'avoir accepté d'être rapportrices de ma thèse. Je remercie aussi Ashwin, Dana et Claire pour les possibilités qu'ils m'ont offertes d'être accueuilli respectivement à Waterloo University, Tel Aviv University et Brown University pour des périodes de temps allant de quelques jours à 3 mois. En plus de ces personnes, je remercie Rahul, Aarthi, Michel et Olivier, avec qui j'ai eu le plaisir de collaborer scientifiquement. Je souhaite également remercier Christian Sohler de m'acceuillir en postdoc l'année prochaine à Dortmund.

Cette thèse est le produit de plusieurs années passées au sein du LIAFA, que je quitte alors qu'il fusionne avec PPS et change de nom pour devenir IRIF. En même temps j'avais voté pour un autre nom[1], donc on va dire que ça tombe bien. Je tiens donc à remercier tous mes collègues présents et passés. Je pense en particulier à Loïc et Christian, mes "grands frères de thèse", à Jad, pour les discussions de voisins de bureau commencées en stage de M2 et qui continuent aujourd'hui, à Elie, pour sa classe incroyable et éternelle, à Antoine[2], Arthur et Clément, pour leur humours aussi inconventionnels les uns que les autres et les discussions absurdes entre deux bureaux, à Lila, pour les parties de jeux de plateau et les longues conversations probablement beaucoup trop sonores qui se terminent généralement par "Everything is terrible!", à Florian, pour m'avoir aidé à soûler tout le monde avec des discussions de "stratégie" à Magic: the Gathering[3], et aussi en vrac à Carola, Moti, Jaime, Vincent, Denis, Charles, Luc, Laure, Marc, Virginie, Nathanaël[4], Irène, Jehanne, Hervé, Inès, Isabelle, Olivier, et tous ceux que j'aurais oubliés.

Cependant, il serait absurde de prétendre que ma vie pendant ces quatre dernières années s'est résumé à deux bâtiments. Déjà, il y avait les enseignements. Je remercie tous les étudiants que j'ai pu avoir en L1 et en L2 pour m'avoir enseigné la patience, et tous les responsables de cours et les autres chargés de TD/TP avec qui j'ai eu le plaisir de travailler. Je pense en particulier à Mehdi avec qui j'ai co-encadré un TP pendant plusieurs mois, et qui m'a beaucoup appris dans ce domaine.

Plus sérieusement, il y a une vie en dehors de la thèse et je me dois de remercier tous les gens qui m'ont

[1]Ada Lovelace Institue, en bon fanboy de *The thrilling adventures of Lovelace and Babbage*.

[2]Les trois.

[3]Ces stratégies ayant souvent plus pour but de parvenir à montrer qu'on est un plus gros geek qu'à réellement gagner une partie.

[4]Non, pas moi évidemment, l'autre.

fait l'apprécier. Parfois la thèse peut être une entreprise quelque peu déprimante, et je dois beaucoup à Marthe, Florence et Élisa pour avoir su trouver les mots qu'il fallait pour me remonter le moral à certains moments où ça n'allait pas.

Je remercie Denise, Rémi, Christophe, Émilia, Jérôme, Averell, Claude, Ismaël, Auréliane, Adeline, Chloé, Paul, et tous les autres du Club Cirque pour leur bonne humeur perpétuelle et la certitude inébranlable que tant que ça tient on peut encore rajouter une voltigeuse de plus. C'est aussi grâce à eux que j'ai découvert l'EJC et la FAC et ainsi rencontré plein de nouveaux gens cools venant de partout dans le monde.

Mon deuxième passe-temps favori après empiler des acrobates consistant à empiler des cartes Magic (et jouer avec, aussi), je remercie Mikaël, Fathi, Max, David, François-Régis et Benoît pour tous les bons moments passés autour de ce jeu. Et pour les jeux qui ne sont pas Magic[5] je remercie aussi Alice, Axel, Sylvain et les autres.

Je remercie les escrimeurs du CÉANS pour avoir fait rimer Marrozo avec mauvais jeux de mots, et Fiore dei Liberi avec humour pourri. Je crains de ne pas pouvoir croiser le fer souvent l'année prochaine mais ce sport reste un des plus classes que j'ai pu pratiquer.

Je remercie aussi les membres du Club Inutile, unis dans la quête de l'absurde, du non-sens, et de la manière la plus élaborée de faire absolument rien. Et aussi ceux du RATON-LAVEUR, unis dans presque la même quête (contrevenant ainsi à l'article 12 du Club Inutile), mais avec des crêpes en plus. C'est cool les crêpes. Et du spam aussi, mais ça ne se mange pas, sauf si on parle du spam que les vikings aiment bien mais je crois comprendre que ça c'est pas bon.[6]

Je ne remercie pas, en revanche, Wizards of the Coast, Paradox Interactive et Fantasy Flight Games pour les heures de productivité perdues dont ils sont à l'origine. Enfin si, mais quand même c'est un chiffre qui fait peur...

Avant de finir, je veux saluer tous ceux que je n'ai pas mentionnés jusqu'ici mais pour qui ma reconnaissance n'est pas moindre : Béatrice, Mathias, Caroline, Marie, Julia, Alexis, Ilia, Albane, Dan, Julie, et toi-même lecteur. Je t'ai peut-être honteusement oublié, mais si tu es en train de lire ma thèse c'est probablement que tu ne le méritais pas.

Enfin, je voudrais remercier ma famille qui me supporte depuis presque 25 ans. Je ne saurais pas trouver les mots pour décrire ce que vous m'avez apporté.

En finissant d'écrire ces remerciements, je me suis rendu compte qu'ils étaient beaucoup plus brefs que ce que je pensais. Moment de panique: suis-je un ingrat ? J'ai pourtant tellement de gens à remercier, qui m'ont tant apporté pendant ces quatre années et avant... J'ai décidé de ne pas en rajouter cependant, parce que ça serait mal vous aimer que de faire du remplissage dans mes sentiments. Pour ceux qui n'ont pas l'habitude de me lire, vous savez maintenant que je suis aussi concis à l'écrit que je suis prolixe à l'oral. La taille de cette thèse en est un autre témoin.

[5] Oui ça existe...

[6] Cette phrase assez décousue vous est offerte par l'esprit du RATON-LAVEUR.

List of publications

This dissertation is based on the following publications:

[18] Nathanaël François and Frédéric Magniez. “Streaming Complexity of Checking Priority Queues”. In: *30th International Symposium on Theoretical Aspects of Computer Science* (2013), p. 454

[17] Nathanaël François, Rahul Jain, and Frédéric Magniez. “Unidirectional Input/Output Streaming Complexity of Reversal and Sorting”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques* (2014), p. 654

[19] Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre. “Streaming Property Testing of Visibly Pushdown Languages”. In: *arXiv preprint arXiv:1505.03334* (2015)

Contents

Chapter 1

Introduction

1.1 A (very) brief history of streaming and sublinear algorithms

When faced with a resource-hungry process that is not trusted to be correct, one may want to check it in real time while using minimal space resources. For example, while it can be useful to have an algorithm running on a server that can detect a Denial Of Service (DOS) attack before it crashes the server, it is impractical to use a significant fraction of the server's processing power or cache to do so, as DOS attacks are not very frequent. In a DOS attack, a relatively small number of clients send a very high number of requests in order to exceed the server's capacity. This can be distinguished from a normal peak in number of requests (which should not cause the serve to preemptively shut down) by the fact that some clients send many more requests than the rest. In 1996, Alon, Matias and Szegedy introduced the *Data Streaming Model* [2] for that purpose, and showed that given a stream of requests from clients with their addresses, it was possible to compute an approximation of the second frequency moment of the addresses, which is enough to detect a DOS attack, using only $\mathrm{O}(\log n)$ memory. In this model, the algorithm sees the input as a stream, and processes it with a head that can only move forward. The algorithm also has a random access working memory, but its space is restricted to some sublinear function of n the size of the input, for example $\mathrm{O}(\sqrt{n})$. Ideally, we want $\mathrm{polylog}(n)$ memory; having a better upper bound than $\log n$ is not possible in most cases as the algorithm needs that much memory just to maintain an index of the element it is reading in the stream. The restriction on memory is both what makes this model relevant for practical applications, and what makes the results different from standard algorithmic results, as otherwise the algorithm would just copy the input in its working memory. In the vast majority of cases, streaming algorithms are randomized, and deterministic algorithms for most problems require linear memory.

While it was introduced as a model for real-time processing of data, the streaming model later also became used to compute functions on very large inputs that do not fit in a random access memory. When a streaming algorithm checks a process in real time, such as in the frequency moments example above, naturally only one pass is allowed. This approach can be compared to online algorithms, but there are two key differences. First, an online algorithm has to make decisions, which will determine the output, while processing its input, unlike a streaming algorithm which can output the result at the end. Second, an online algorithm is not bounded computationally. It can remember the entire history if it needs to, and perform long computations between bits of the input. This contrasts to a real time streaming algorithm, where in addition to the memory constraints, we usually aim for $\mathrm{polylog}(n)$ time between processing bits. In the main other paradigm for streaming algorithm, the input corresponds to a very large amount of data stored in a way that makes random access costly, and multiple passes can be allowed, potentially in both directions. An example would be very large graph, such as the web graph (i.e. the graph representing all web-pages and hyperlinks), which we can visit several times but cannot store in our working memory. Maximum weighted matchings, for example, are significantly easier to approximate with a streaming algorithm that makes multiple passes ($(2+\varepsilon)$-approximation) than with a single pass algorithm (4.911-approximation) [33]. Another potential application is the currently open problem of finding ways to fold an RNA sequence that maximize the number of pairs. RNA sequences can be very long but if an algorithm is reading them, they have presumably been stored in some sort of memory already.

The streaming model can be seen as part of a larger trend. When the field of algorithmic complexity was first developed, the gold standard for an algorithm was linear time and linear space, as it was understood that the algorithm needs to store and read the input. The research focused on the complexity of the computations involved in computing many functions, in particular with famous complexity classes like P or BPP representing problems that can be solved either deterministically or using randomness in time polynomial in the input size. However, beginning in the 90s, as *big data*, i.e. problems with huge inputs, became more prevalent, it became apparent that even linear time and linear memory could be too much. The input would be stored in some external memory, different from the working memory, or even not stored locally and accessed by querying various parties. This new approach, called *sublinear algorithms*, includes the streaming model (sublinear memory), property testing (sublinear time), and distributed computing (sublinear communication in the congestion model, or sublinear time in the local model)[1]. Property testers, in the query complexity model, are algorithms which do not directly access the input but instead query an oracle for a small, possibly random, portion of it. Distributed computing in the congestion model is linked to communication complexity, as some number of players want to compute a function of their respective inputs by sending each other messages of total length sublinear in the total input size. As with streaming algorithms, randomness is almost always present in sublinear algorithms, as the functions we wish to compute generally depend on the whole input and not just the sublinear part of it we can access, remember, or share.

Communication complexity

Limitations of streaming algorithms are closely linked to lower bounds in *communication complexity*. In communication complexity, which was introduced by Yao in 1979 [41], two or more players have private inputs and want to compute some function of their inputs by sending message to each other.

A protocol determines the message each player will send based on the messages they previously received, and their input. The communication complexity of a protocol is the total size of the messages sent by its players in the worst case. For any functions, there exists a trivial protocol to compute it where each player will communicate their entire input to the others. The goal is therefore to find protocols that use sublinear communication, or to prove that no such protocols exist.

Communication complexity is a very rich field with many variants and many applications, but one of them in particular is extremely relevant to streaming algorithms. The most common way to prove a streaming lower bound (with only one pass and one stream) is to consider the communication problem defined by cutting the input in two in some natural way, giving the first half to Alice, the second half to Bob, and show that the resulting communication problem is hard. If there existed a streaming algorithm with small memory, Alice could just run it on her half of the input and send the memory to Bob, who would finish running the algorithm and output the value of the function. Therefore a communication complexity lower bound can be used to prove a streaming complexity lower bound.

More generally, we can cut the input in k consecutive parts and give each factor of the input obtained this way to a different player. The players are arranged in a cycle, know only their own input and send messages to the next player in the cycle: this is known as number-in-hand message-passing. For any streaming algorithm

[1]In the local model, players are unrestricted in the size of the messages they send, but messages are sent simultaneously during each round and the goal is to minimize the number of rounds. In the congestion model, the message size is the limiting factor.

using p passes and s memory space, k players can simulate this algorithm with total communication ksp: each player receives the current memory of size at most s from the previous player, runs the algorithm on their input, and sends their current memory to the next player, until the computation ends. By contraposition, if we can cut the input of a function in such a way that the resulting communication problem is hard, then we can prove that the streaming complexity of the function is high.

Property Testers

The third field in sublinear algorithms is *property testing*. Property testing was first implictly used in 1990 by Blum, Luby and Rubinfeld [8] although the concepts were formally introduced by Gemmell, Lipton, Rubinfeld and Sudan in 1991 [21]. for program checking and self correction: given a black-box program claimed to compute some function, the goal is to verify that, with high probability, the program is right on most inputs (program checking), and from this to build a program that is correct with high probability on all inputs (self correction). Property testing then expanded beyond program checking to include among others testing properties of graphs [23], membership in a language (such as the language of well-parenthesized words with two types of parentheses [36], regular languages [1],[35], or closeness of two probability distributions [9]. In each of those extensions, the goal of a tester is to accept with high probability all correct inputs and reject with high probability all inputs that are too far (with regard to a normalized distance) from being correct, but it may behave in an undefined manner for inputs that are not quite correct but close. For example, an ε-property tester for a language L and the Hamming distance $\mathsf{dist}_{\mathrm{H}}$ is an algorithm that, with high probability, accepts inputs $u \in L$ and rejects inputs u such that $\mathsf{dist}_{\mathrm{H}}(u, L) > \varepsilon|u|$ (with $|u|$ the length of u). This definition naturally lends itself to a (randomized) sublinear time approach, as if an algorithm is allowed to accept inputs that are only close to being correct, it can afford to not read all of it. We consider the *query complexity* of such a tester. In the query complexity model, the algorithm does not have direct access to the input and must query any bit it wants to see. This is considered the most costly operation and the query complexity, i.e. the number of queries the algorithm must send, represents best the complexity of the algorithm. Property testing and query complexity go naturally together as an algorithm with a small number of queries is always very unlikely to distinguish two inputs that are very close, but property testers get around this limitation by being allowed to accept bad inputs that are close to good ones.

Variations on the query model have also been considered. One of them is the *trial-and-error model*, where the algorithm is trying to solve some problem with small witnesses (for example, a problem in NP) where the input is hidden, and can submit a tentative answer with a witness to an oracle, which will either accept it, or refuse it by pointing to a part of the input which contradicts the witness [6]. Another variation is to allow the algorithm, either in addition to or instead of the standard query, to ask the oracle other questions about the input. For example, in the case of a graph, the algorithm could ask for the distance between two vertices [27]. A more recent variation involves testing *dynamic environments* [22], with the environment evolving over time and queries only being possible on the current state, similar to an online model.

However property testing is not inherently tied to the query model: we can also consider *streaming property testing*, which are simply streaming algorithms that are property testers. Streaming property testers, then called spot-checkers, were first introduced in 1998 by Ergün, Kannan, Kumar, Rubinfeld and Viswanathan [14], and have been used several times since [15],[12]. Here the algorithm will have access to the entire input, but will be forced to discard some of it by the restriction of having a sublinear memory. Any non-adaptive tester in the query model using s queries can easily be modeled by a streaming property tester using memory space s, as it just needs to remember the bits corresponding to the queries. However, adaptive testers, where queries potentially depend on the result of the previous ones, cannot be so easily simulated by a streaming algorithm. Note that streaming property testers can also be linked to the previously mentioned testers on dynamic environments: a streaming property tester using $\mathrm{O}(f(n)t)$ memory for an input of size

$\mathrm{O}(nt)$ with $t > n/f(n)$ can simulate a dynamic environment property tester (even with adaptive queries) for an environment of size n evolving over t steps which queries at most $f(n)$ elements at each step. The algorithm can simply store the entire environment at any step in memory because $t > f(n)$, and then choose the queries it wants to remember.

Note that, if there exists results in property testing that do not rely solely on a query model, the reverse is also true: the first results on query complexity (i.e. deciding properties while looking at only a fraction of the input), date from 1976 with Rivest and Vuillemin [37] for graph properties recognized for adjacency matrices, well before property testing. The query complexity of evaluating decision trees such as an AND-OR tree, was also extensively studied, and particularly gaps between algorithms with one-sided and two-sided error [39], [31][2]. Another major application of randomized queries is the study of the complexity class PCP, a generalization of NP defined as the class of problems for which there exists a (possibly exponential in size) witness, or proof, that can be probabilistically checked by a verifier in polynomial time [7].

1.2 Our results

In this thesis, we consider relaxations of the streaming model, and give algorithms and lower bounds for problems in those models. Two types of relaxations are considered:

- Augmenting the model to give more resources to our algorithm (more passes, more streams...)
- Relaxing constraints on the output (such as with streaming property testing)

Each time our study of relaxations comes from a double question : "what is the minimal relaxation we need to solve efficiently some given problem?" and "given this relaxation, what is the hardest problem we can efficiently solve with it?". Although we do not fully answer these guiding questions in every case, they structure our study.

Multiple passes is one of the most immediate relaxations. For multiple passes in the same direction the outcome is often either that the complexity does not change much ($s(p)$ the space required for p passes is equal to $s(1)/p$) or that the problem becomes much easier ($\mathrm{polylog}(n)$ memory) after some number of passes. For example, it requires $\Omega(|\Sigma|)$ memory to test if more than half the letters of $u \in \Sigma^n$ are the same with only one pass, but with two passes an algorithm can use the first pass to select the only possible candidate[3], and then use the second pass to check that it represents indeed more than half the letters of u, all with only $\mathrm{O}(\log|\Sigma| + \log n)$ memory [28]. In this basic model, we generalized a lower bound result originally by Chakrabarti, Cormode, Kondapalli, and McGregor [10] on priority queues. This language consists of tasks requested with a certain priority, that we call insertions, and tasks effected, that we call extractions, and any extraction must match the highest priority currently inserted but not yet extracted. They showed that recognizing priority queues in streaming with one pass required memory $\Omega(\sqrt{n})$, which matched their algorithm up to a logarithmic factor. This result was a consequence of the demonstration [26] that recognizing $\textsc{Dyck}[2]$, the languages of well-parenthesized words with two types of parentheses, in p passes requires memory space $\Omega(\sqrt{n}/p)$, as parenthesis languages can be seen as priority queues.

> In Section 3.1, using similar information theory tools, we show that recognizing priority queues augmented with timestamps in streaming with p passes requires memory space $\Omega(\sqrt{n}/p)$.

[2]Very recently, a decision tree with a gap larger than any previously known one was found, settling a 30 year old open problem [5].

[3]The first letter is the first candidate. Whenever the candidate is read, a counter is incremented. Whenever a non-candidate is read, the counter is decremented, except if it is already zero, in which case the current letter becomes the new candidate.

The language of priority queues with timestamps cannot, in general, be compared to DYCK$[2]$, although both languages can be seen as special cases of priority queues. This is especially relevant as prior to the algorithm from [10], the best streaming results on priority queues were both for priority queues augmented with timestamps [12]: a streaming algorithm using memory $\mathrm{O}(\sqrt{n}\log n)$, as well as a streaming property tester using memory $\mathrm{O}(\log n)$. The hard distribution we construct for our communication problem has a key difference to what can be seen in earlier proofs: it relies on the information complexity of an asymmetric 3-player communication problem, whereas several lemmas used for the proof of DYCK$[2]$, notably the cut-and-paste lemma, only work for 2 players.

After multiple passes in the same direction, multiple passes in different directions is a natural evolution: Magniez, Mathieu and Nayak [32] showed that having one pass in each direction dramatically speeds up the recognition of the languages of DYCK$[2]$ from $\Omega(\sqrt{n})$ to $\mathrm{polylog}(n)$. Improving on the techniques developed there, we showed that the same is true for the language of priority queues. While it is not a visibly pushdown language like DYCK or XML, it still consists of extractions and insertions. This allows us to use a technique similar to the one used for the bidirectional algorithm for DYCK$[2]$: divide the part of the input already read into a logarithmic number of blocks. Each time a new letter is read, a block of size 1 is created, and when two blocks have the same size they merge. This means the sequence of block sizes corresponds to a binary decomposition of the index of the letter the stream is currently processing. If the total size of the stream is a power of 2 (it can easily be with some padding), then the same blocks appear on passes in both direction. Merging naturally destroys some information: while the process does not cause the algorithm to forget everything about the order of the elements inside the block, the amount of information we use per block cannot depend on the size of the block. The difficulty therefore lies in finding a way to detect any witness of the input's non-membership in the language before block merging makes the witness undetectable. This is why we want the same blocks to exist in both directions, as sometimes witnesses will be seen from right to left, and sometimes from left to right. Things are further complicated by the fact that, unlike well-parenthesized words, priority queues are not stable by symmetry: insertions can happen in any order, and we need a different algorithm for the pass from left to right and the pass from right to left.

> In Section 4.1, we show that there exists a bidirectional algorithm for recognizing priority queues using memory space $\mathrm{O}(\log n^2)$.

The natural question is to ask whether this can be extended to other languages of the same form. However, beyond specific cases like Dyck languages [32], or XML documents representing binary trees [30], even general visibly push-down languages, let alone all languages composed of insertions and extractions, are hard to recognize with bidirectional streaming algorithms. Even an XML document representing a ternary tree can be reduced to a Disjointness communication problem, and therefore verifying its validity requires $\Omega(n)$ memory. Two relaxed approaches either add another stream, or loosen the constraints on the output (i.e. streaming property testing).

We first consider streaming property testers for visibly pushdown languages. Visibly pushdown languages are defined as the languages recognized by stack automata such that each letter in the alphabet either always causes a push, always causes a pop, or never affects the stack. In addition to being hard to recognize with streaming algorithms, VPLs are also relatively hard to test in the regular query model, requiring $\Omega(n^{1/3})$ queries[4]. We also have an example of a visibly pushdown language, a variation of DISJOINTNESS, that streaming algorithms cannot recognize with less than $\Omega(n)$ memory and for which property testers need

[4]Consider the XML encoding of a tree with $n^{2/3}$ branches of length $n^{1/3}$. Branches have to be of the form a^*b^*. This cannot be tested without two random samples in the same (random) branch, selected from the whole branch. By the birthday paradox, this requires at least $n^{1/3}$ samples.

$\Omega(n^{1/11})$ queries[5]. This language is the set of xy with $x \in \{0,1\}^n$, $y \in \{\overline{0}, \overline{1}\}$ such that for all i $x_i y_i = 0$. The streaming lower bound for the communication complexity lower bound follows naturally form the communication lower bound on DISJOINTNESS, whereas the query complexity lower bound for the property tester can be deduced from the proof of the lower bound for a property tester for DYCK$[2]$ in [36]. We designed a streaming property tester for visibly pushdown languages that uses logarithmic memory. The first step is to manage a sequence of push symbols followed by a sequence of pop symbols, which we call a peak: even recognizing peaks in a VPL has polynomial complexity in the models mentioned above. The way we do it is by seeing peaks as elements of a regular language over $\Sigma \times \Sigma$, where each push symbol is matched to a pop symbol on the other side. We can design a streaming property tester for peaks by using a non-adaptive query model property tester by [1], refined in [35]: the algorithm remembers the push symbols corresponding to queries it wants to make, and then obtain those queries by completing them with the matching pop symbols. We then show that this algorithm can be used more generally to transform such a peak into a big neutral symbol representing the possible transitions of the stack automaton. Ideally, we would like to repeat the process whenever we see such a sequence, however we must be careful not to accumulate error in a nested way that could result in us either accepting inputs too far from the language, or having to use too much memory for the first tester in order to keep the error small. This can be managed with a stack of peaks waiting to either be transformed or be resumed as a later peak is transformed.

> In Section 3.2, we give an ε-property tester with memory space $\mathrm{O}(\log^7 n/\varepsilon^4)$ for $\varepsilon > 0$ and any visibly pushdown language with the edit distance.

Finally, we discuss machines with more than one streaming tape. With a relatively small ($\mathrm{O}(\log n)$) number of passes and three tapes, it is very easy to sort a stream (and [24] proved that this is optimal), which makes it possible to transform or annotate inputs in order to recognize languages that are otherwise hard, such as XML (Konrad and Magniez [30], also some more general results for deterministic context free languages (which include VPLs) can be derived from results in [29] and [38]). The transformations that can be performed in this manner include sorting a subset of the input in $\mathrm{O}(\log n)$ passes and $\mathrm{O}(\log n)$ memory. This in turn allows for example evaluation of XPath expressions on an XML document. Here, our interest lies with transformations that can be done with a less powerful model : only two streaming tapes, one of them the input and the other the output, and passes are only from left to right. We then add other restrictions, such as preventing the algorithm from overwriting on the input stream (*Read-only*), or preventing the algorithm from reading its own output (*Write-only*), justified by the lower cost of such memory (compact disks and their successors, for example, are generally read-only). In this model, even the simple problem of reversing the input stream becomes non-trivial.

> We show in Section 4.2.1 that unsurprisingly, in the model where the algorithm can only read the input and can only write on the output, and only once in each cell (*Burn-only*), the naive algorithm that during each pass remembers as much as it can from the input, reverses it and writes the result in the correct place on the output streams performs best: the required memory space is $\Omega(n/p)$ for p passes for any randomized algorithm.

What is more unexpected is that this bound is not known to be true for randomized algorithms when the output stream is Write-only instead (i.e. the algorithm can write multiple times in a cell), even though intuitively it should not change anything.

> When either of the streams is *Read-Write* instead, then we show in Section 4.2.2 that the required space drops to $\Omega(n/p^2)$ and in Section 3.3.1.1 that this bound is met by a deterministic algorithm.

[5]Possibly more, this is not a tight bound.

> For two Read-Write streams, we show in Section 3.3.1.2 that there exists a deterministic algorithm using $\mathrm{O}(\log n)$ passes and $\mathrm{O}(\log n)$ memory.

Because this model has two independent streams, we were unable to use communication complexity, as it would be common to do to prove streaming lower bounds. Indeed if a player has a part of each stream, they could have "communicate" with other players simply by making one of the heads move while the other stayed in place. This would cause the two heads be in parts belonging to two different players, allowing communication that does not go through the algorithm's memory. We therefore had to develop new techniques for those proofs. When both streams are Read-Write, it is still possible to sort with $\mathrm{O}(\log n)$ passes and $\mathrm{O}(\log n)$ memory [11], as with three streams. However there is an apparent trade-off:

> In Section 3.3.2, we give both a deterministic algorithm, based on Merge Sort, that may use significantly more space on the stream than the input size, $\mathrm{O}(n \log n)$, and a randomized algorithm, based on Quick Sort, that always returns the correct result and does not expand the stream by more than a linear factor, but may require more than the expected $\mathrm{O}(\log n)$ passes to finish[6].

We do not know if there exists an algorithm in $\mathrm{O}(\log n)$ passes and $\mathrm{O}(\log n)$ memory that combines the best of both worlds by being deterministic and not requiring more than linear expansion of the stream.

[6]Alternatively, it always ends after $\mathrm{O}(\log n)$ passes but sometimes outputs nothing.

Chapter 2

Models and languages

2.1 Languages and functions considered

All languages we define in this section consist of push symbols, or insertions, and pop symbols, or extractions. A word is said to be *balanced* if it has as many push symbols (resp. insertions) as pop symbols (resp. extractions). Note that all the definitions will require in particular that the words be balanced, and a streaming algorithm can check that simply by counting the height of the stack, using $\mathrm{O}(\log n)$ memory. Therefore in the rest of this work we will often assume that the input is balanced.

2.1.1 Visibly Pushdown Languages

Visibly pushdown languages are languages recognized by stack automata where each symbol determines whether the automaton pushes or pops on the stack, and it does not depend on the state or current stack. They form a subclass of *deterministic context free languages*. Notable visibly pushdown languages include the XML language, the language of well-parenthesized words (or *Dyck language*), and the language of valid traces of the execution of a program. For Dyck language, we write $\mathrm{DYCK}[k]$ for the language of well-parenthesized words with k types of parentheses.

A *finite state automaton* is a tuple of the form $\mathcal{A} = (Q, \Sigma, Q_{in}, Q_f, \Delta)$ where Q is a finite set of control states, Σ is a finite input alphabet, $Q_{in} \subseteq Q$ is a subset of initial states, $Q_f \subseteq Q$ is a subset of final states and $\Delta \subseteq Q \times \Sigma \times Q$ is a transition relation. For a pair of states $p, q \in Q$ and $u \in \Sigma^*$, we write $p \xrightarrow{u} q$, to mean that there is a sequence of transitions in $\mathcal{A}$ from p to q while processing u, and we call (p, q) a *u-transitions*. For $\Sigma' \subseteq \Sigma$, the *Σ'-diameter* (or simply *diameter* when $\Sigma' = \Sigma$) of $\mathcal{A}$ is the maximum over all possible pairs $(p, q) \in Q^2$ of $\min\{|u| : p \xrightarrow{u} q \text{ and } u \in \Sigma'^*\}$, whenever this minimum is not over an empty set. We say that $\mathcal{A}$ is *Σ'-closed*, when $p \xrightarrow{u} q$ for some $u \in \Sigma^*$ if and only if $p \xrightarrow{u'} q$ for some $u' \in \Sigma'^*$.

A *pushdown alphabet* is a triple $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ that comprises three disjoint finite alphabets: Σ_+ is a finite set of *push symbols*, Σ_- is a finite set of *pop symbols*, and $\Sigma_=$ is a finite set of *neutral symbols*. For any such triple, let $\Sigma = \Sigma_+ \cup \Sigma_- \cup \Sigma_=$. Intuitively, a *visibly pushdown automaton* [4] over $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ is a pushdown automaton restricted so that it pushes onto the stack only upon reading a push symbol, it pops the stack only upon reading a pop symbol, and it does not modify the stack on reading a neutral symbol. Up to coding, this notion is similar to the one of input driven pushdown automata [34] and of nested word automata [3].

Definition 2.1.1 (Visibly pushdown automaton [4])**.** *A* visibly pushdown automaton *(*VPA*) over $\langle \Sigma_+, \Sigma_-, \Sigma_= \rangle$ is a tuple $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ where Q is a finite set of states, $Q_{in} \subseteq Q$ is a set of initial states, $Q_f \subseteq Q$ is a set of final states, Γ is a finite stack alphabet, and $\Delta \subseteq (Q \times \Sigma_+ \times Q \times \Gamma) \cup (Q \times \Sigma_- \times \Gamma \times Q) \cup (Q \times \Sigma_= \times Q)$ is the transition relation.*

To represent stacks we use a special bottom-of-stack symbol $\perp$ that is not in Γ. A *configuration* of a VPA $\mathcal{A}$ is a pair (σ, q), where $q \in Q$ and $\sigma \in \perp \cdot \Gamma^*$. For $a \in \Sigma$, there is an *a-transition* from a configuration (σ, q) to (σ', q'), denoted $(\sigma, q) \xrightarrow{a} (\sigma', q')$, in the following cases:

- If a is a push symbol, then $\sigma' = \sigma\gamma$ for some $(q, a, q', \gamma) \in \Delta$, and we write $q \xrightarrow{a} (q', \mathsf{push}(\gamma))$.
- If a is a pop symbol, then $\sigma = \sigma'\gamma$ for some $(q, a, \gamma, q') \in \Delta$, and we write $(q, \mathsf{pop}(\gamma)) \xrightarrow{a} q'$.
- If a is a neutral symbol, then $\sigma = \sigma'$ and $(q, a, q') \in \Delta$, and we write $q \xrightarrow{a} q'$.

For a finite word $u = a_1 \cdots a_n \in \Sigma^*$, if $(\sigma_{i-1}, q_{i-1}) \xrightarrow{a_i} (\sigma_i, q_i)$ for every $1 \leq i \leq n$, we also write $(\sigma_0, q_0) \xrightarrow{u} (\sigma_n, q_n)$. The word u is *accepted* by a VPA if there is $(p, q) \in Q_{in} \times Q_f$ such that $(\perp, p) \xrightarrow{u} (\perp, q)$. The language $L(\mathcal{A})$ of $\mathcal{A}$ is the set of words accepted by $\mathcal{A}$, and we refer to such a language as a *visibly pushdown language* (VPL). Note that a VPA can always be determinized, which implies that visibly pushdown languages are a subclass of deterministic context-free languages.

The way our algorithm works in Section 3.2 requires us to consider weighted words. A *weight function* on a word u with n letters is a function $\lambda : [n] \to \mathbb{N}^*$ on the letters of u, whose value $\lambda(i)$ is called the *weight of* $u(i)$. A *weighted word* over Σ is a pair (u, λ) where $u \in \Sigma^*$ and λ a weight function on u. We define $|u(i)| = \lambda(i)$ and $|u[i, j]| = \lambda(i) + \lambda(i+1) + \ldots + \lambda(j)$. The length of (u, λ) is the length of u. For simplicity, we will denote by u the weighted word (u, λ). Note that in our model, some letters can only be of certain weights.

2.1.2 Priority Queues

Data structures such as stacks, dictionnaries, symbol tables and priority queues were formalized by Flajolet, Françon and Vuillemin [16], who studied their combinatorial properties as well as give efficient implementations of those structures using lists and trees. Priority queues form a data structure where elements can be added with any key (priority) in an ordered set, but a deletion operation will always delete the element with the highest key. As their name indicate, they have applications for managing traffic with an insufficient bandwith and similar ressource allocation problems. Another application is to improve the running time of Dijkstra's algorithm for finding the shortest paths in a graph, although the best improvement uses a more powerful variant of priority queues that we do not study here [20]. There exist many ways of implementing priority queues using either tournaments or sorted lists [40].

In this thesis however we do not consider the problem of implementing a priority queue, but rather of externally checking that a priority queue is correct. The problem is therefore to recognize the language of valid priority queues history. From now on, we will use the name priority queues to refer to that language.

To formally define it, we first consider a more general language, COLLECTION. It corresponds to the valid history of dictionaries, i.e. a data structure where elements are inserted and extracted, but with no restriction on keys. We can then define Priority queues as a particular case of Collections.

Definition 2.1.2 (COLLECTION,PQ)**.** *Let Σ_0 be some alphabet. Let $\Sigma = \{\mathtt{ins}(a), \mathtt{ext}(a) : a \in \Sigma_0\}$. For $w \in \Sigma^N$, define inductively multisets M_i by $M_0 = \emptyset$, $M_i = M_{i-1} \setminus \{a\}$ if $w[i] = \mathtt{ext}(a)$, and $M_i = M_{i-1} \cup \{a\}$ if $w[i] = \mathtt{ins}(a)$.*
Then $w \in$ COLLECTION(Σ_0) if and only if $M_n = \emptyset$ and $a \in M_{i-1}$ when $w[i] = \mathtt{ext}(a)$, for $i = 1, \ldots, N$. Moreover, $w \in$ PQ(U), for $U \in \mathbb{N}$, if and only if $w \in$ COLLECTION$(\{0, 1, \ldots, U\})$ and $a = \max(M_{i-1})$ when $w[i] = \mathtt{ext}(a)$, for $i = 1, \ldots, N$.

A variant on priority queues involves adding timestamps. Timestamps point each extraction to the index of the matching insertion. Note that in the context we consider here, of an external verification of a system, requiring that the system adds the timestamps costs very little. Indeed, the system already has to manage the priority queue so know at each time the list of all requested tasks, and storing the additional information of the date of request costs little.

Definition 2.1.3 (PQ-TS)**.** *Let* $\Sigma = \{\mathtt{ins}(a), \mathtt{ext}(a) : a \in \{0, 1, \ldots, U\}\} \times \mathbb{N}$*. Let* $w \in \Sigma^N$*. Then* $w \in$ PQ-TS(U) *if and only if* $w \in$ COLLECTION(Σ)*,* $w[1, \ldots, N][1] \in$ PQ(U)*, and* $w[i][2] = i$ *when* $w[i][1] = \mathtt{ins}(a)$*.*

2.1.3 Reversing and Sorting

Last, we also consider the problem of sorting and reversing a stream, which are not language recognition tasks, we define formally the functions Sort and Reverse as follows.

Definition 2.1.4 (Reverse and Sort)**.** *For a sequence* $w = w[1]w[2]\ldots w[n] \in \Sigma^n$*, let us define* $\mathrm{Reverse}(w)$ *as* $w[n]w[n-1]\ldots w[1]$*. When* Σ *has a total order, also define* $\mathrm{Sort}(w)$ *as the sorted permutation of* w*.*

2.2 The streaming model

A *data stream* is a sequence $x = x_1, \ldots, x_n \in \Sigma^n$ where Σ is some finite alphabet. In the problems we consider, Σ will often be a set of insertions, or opening tags, or push symbols, and extractions, or closing tags, or pop symbols, and will also sometimes contain other symbols, called neutral symbols. However, in a general context, Σ could be many other finite sets: for example pairs of integers in $\{1, \ldots, m\}$ for some m, with x the multiset of edges in a graph of size m. In particular in many of those contexts, the order of x does not matter with regard to the object it represents, and can be assumed to have been chosen by an adversary. This is however not the case in the problem considered in this thesis.

Definition 2.2.1. *A* k-stream unidirectional streaming algorithm *is an algorithm* A *with a set of streams* $X_1, \ldots, X_k$ *such that:*

- *Initially* X_1 *contains the input and all other streams are empty*
- *At each time step, for each* $i \in [k]$*, the algorithm processes a single symbol* $x_{i,j}$ *on stream* X_i*. We says that the* i*-th head is on* x_i*. It cannot read or write on other letters of that stream.*
- *For all streams* X_i *with* $i \in [k]$*, the head can only move to the right, or back to the beginning of* X_i*. This means that if* A *is processing* $x_{i,j}$ *then at the next time step it can either still be processing* $x_{i,j}$*, be processing* $x_{i,j+1}$ *instead, or be processing* $x_{i,1}$ *instead.*

We say that A *makes* p *passes and uses memory space* s *on input* u *if for each computation on* u*, for each stream* X_i *with* $1 \leq i \leq k$ *the head moves back to the beginning of* X_i *at most* p *times, and the size of the memory never exceeds* s *bits.*

We also define bidirectional streaming algorithms. The definition is similar, except that each stream may run in different directions at different times.

Definition 2.2.2. *A* k-stream bidirectional streaming algorithm *is a deterministic algorithm* A *with a set of streams* $X_1, \ldots, X_k$ *and a function* $\mathrm{dir} : \{1, \ldots, k\} \times \mathbb{N} \longrightarrow \{-1, +1\}$ *such that:*

- *Initially* X_1 *contains the input and all other streams are empty*
- *At each time step, for each* $i \in [k]$*, the algorithm processes a single symbol* $x_{i,j}$ *on stream* X_i*. We says that the* i*-th head is on* x_i*.*
- *For all streams* X_i *with* $i \in [k]$*, the head can only move in the direction of* $\mathrm{dir}(i, t)$*, or jump to any end of* X_i*, where* t *is the number of time the head has jumped to an end of* X_i*. This means that if* A *is processing* $x_{i,j}$ *then at the next time step it can either still be processing* $x_{i,j}$*, be processing* $x_{i,j+\mathrm{dir}(i,t)}$ *instead, or be processing* $x_{i,1}$ *or* $x_{i,n}$ *instead.*

We say that A makes p passes and uses memory space s on input u if for each computation on u, for each stream X_i with $1 \leq i \leq k$ the head jumps to an end of X_i at most p times, and the size of the memory never exceeds s bits.

We also must define what exactly the algorithm can do with streams. The most common model is for the algorithm to only read an input stream, however other cases are possible.

Definition 2.2.3. *Let A be a streaming algorithm, X one of its streams. We say that X is:*

- Read-only*: if A does not change X,*
- Write-only*: if actions performed by A never depend of the content of the letter currently processed on X,*
- Burn-only*: if it is Write-only and, in addition, during any computation, A can change each letter of X at most once,*
- Read-Write*: if it is neither Read-only nor Write-only.*

From now on, if we do not specify otherwise, we will assume a streaming algorithm only uses one stream (which by definition contains the input) and that this stream is Read-only. This will be the case whenever we consider the problem of recognizing a language.

For algorithms with Read-Write streams, we define the concept of *expansion*. While a streaming algorithm is limited in memory, it could potentially "cheat" by expanding the stream and using the additional space for more memory. While multiplying the size of the stream by a constant can plausibly be afforded, even logarithmic expansion could be too much for very large inputs.

Definition 2.2.4. *A streaming algorithm has* expansion $\lambda(n)$ *if for an input of size n, all its streams have total length at most $\lambda(n) \times n$ during its execution.*

2.3 Distance and Property Testers

2.3.1 Balanced/Standard Edit Distance

A property tester is defined for a certain distance. The usual distance between words in property testing is the Hamming distance. In this work, we consider an easier distance to manipulate in property testing but still natural and relevant for most applications, which is the edit distance.

Given any word u, we define two possible *edit operations*: a *deletion* of a letter in position i with corresponding cost $|u(i)|$, and its converse operation, the *insertion*, where we also select a weight, compatible with the restrictions on λ, for the new $u(i)$. Then the *(standard) edit distance* $\mathsf{dist}(u, v)$ between two weighted words u and v is simply defined as the minimum total cost of a sequence of edit operations changing u to v. Note that all letters that have not been inserted or deleted must keep the same weight. For a restricted set of letters Σ', we also define $\mathsf{dist}_{\Sigma'}(u, v)$ where the insertions are restricted to letters in Σ'.

We will also consider a restricted version of this distance for balanced words, motivated by our study of VPL. Similarly, *balanced-edit operations* can be deletions or insertions of letters, but each deletion of a push symbol (resp. pop symbol) requires the deletion of the matching pop symbol (resp. push symbol). Similarly for insertions: if a push (resp. pop) symbol is inserted, then a matching pop (resp. push) symbol must also be inserted simultaneously. The cost of these operations is the weight of the affected letters, as with the edit operations. Again, only insertions of letters with weight 1 are allowed. We define the *balanced-edit distance*

$\mathsf{bdist}(u, v)$ between two balanced words as the total cost of a sequence of balanced-edit operations changing u to v. Similarly to $\mathsf{dist}_{\Sigma'}(u, v)$ we define $\mathsf{bdist}_{\Sigma'}(u, v)$.

When dealing with a visibly pushdown language, we will always use the balanced-edit distance, whereas we will use the standard-edit distance for regular languages. We also say that u is (ε, Σ')-*far* from v if $\mathsf{dist}_{\Sigma'}(u, v) > \varepsilon|u|$, or $\mathsf{bdist}_{\Sigma'}(u, v) > \varepsilon|u|$, depending on the context. We omit Σ' when $\Sigma' = \Sigma$.

2.3.2 Property Testers

An ε-tester for a language L accepts all inputs which belong to L with probability 1 and rejects with high probability all inputs which are ε-far from L, *i.e.* that are ε-far from any element of L.

Definition 2.3.1 (Property tester). *Let $\varepsilon > 0$ and let L be a language. An* ε-tester *for L with one-sided error η is a randomized algorithm A such that, for any input x of length n:*

- *If $u \in L$, then A accepts with probability* 1*;*
- *If u is ε-far from L, then A rejects with probability at least* $1 - \eta$*;*

A property tester can be a *query property tester*, which accesses the input through queries and where the goal is to minimize the number of queries, or a *streaming property tester*, which accesses the input as a stream in a single pass and where the goal is to minimize the memory space used. We do not discuss other models in this thesis.

A query property tester is said to be *non-adaptive* if the queries it makes do not depend on the result of previous queries.

2.4 Information theory and communication complexity

Information theory was introduced by Claude Shannon in 1948 to study the limits on compression and communication of data. Communication complexity was introduced by Andrew Yao in 1979 to study the limit on communication needed between two players to compute some function of their private inputs. More recently, it was found that one of the best lower bounds for the communication complexity of a function was its information complexity, i.e. the information each player will learn about the other player's input from the transcript of a protocol, under a certain distribution. Shannon's compression theorems then imply it is impossible to have a transcript of smaller size than the information learned this way.

As explained in the introduction, communication complexity lower bounds are very useful for proving streaming complexity lower bounds. In Chapter 4, we rely on the definitions and facts presented here for our proofs.

2.4.1 Information theory

One of the most important concepts developed by Shannon is the *entropy* of a random variable. The entropy of a random variable X is the smallest number of bits required on average to describe the value of X, and can be though of as the quantity of randomness in X.

Definition 2.4.1. *Let X be a random variable taking its values in some set S. For $x \in S$, let p_x be $\Pr(X = x)$. Then the* entropy of X *is* $\mathrm{H}(X) = -\sum\limits_{x \in S} p_x \log p_x$.

For example, the entropy of a uniform random variable in $\{1, \ldots, n\}$ is $\log n$, while the entropy of a variable that only takes one value is 0.

Entropy can also be defined conditioned on another variable:

Definition 2.4.2. *Let X and Y be a random variables, X taking its values in some set S and Y in some set T. For $x \in S$ and $y \in T$, let $p_{x,y}$ be $\Pr(X = x|Y = y)$. Then the* entropy of X conditioned on Y *is* $\mathrm{H}(X|Y) = -\mathbb{E}_{y\in T}\left(\sum_{x\in S} p_{x,y}\log p_{x,y}\right)$.

The entropy of X conditioned on Y can be seen as the amount of information needed to describe the value of X knowing the value of Y. *Mutual information* is the complement of this, the information that the value of X gives on the value of Y.

Definition 2.4.3. *The* mutual information *between two random variables X, Y is* $\mathrm{I}(X : Y) = \mathrm{H}(X) - \mathrm{H}(X|Y)$.

For example, the mutual information of two independent coin tosses is 0. However, the mutual information between the result of the first toss and the number of heads obtained is $1/2$, as knowing the result of the first toss reduced the randomness of the number of heads.

Definition 2.4.4. *Let X, Y, Z be three random variables. The* conditional mutual information of X and Y on Z *is* $\mathrm{I}(X : Y|Z) = \mathrm{H}(X|Z) - \mathrm{H}(X|Y, Z)$.

We now state some generally useful facts regarding entropy and mutual information. For a proof of those, we refer to [13].

Fact 2.4.5. *For any two random variables X, Y, we have* $\mathrm{I}(X : Y) = \mathrm{I}(Y : X) = \mathrm{H}(X) + \mathrm{H}(Y) - \mathrm{H}(X, Y) = \mathrm{H}(X, Y) - \mathrm{H}(X|Y) - \mathrm{H}(Y|X)$.

Fact 2.4.6 (Data processing inequality)**.** *Let X, Y, Z, R be random variables such that R is independent from X, Y, Z. Then for every function f,*

$$\mathrm{H}(X|Y, Z) \leq \mathrm{H}(f(X, R)|Y, Z) \quad \textit{and} \quad \mathrm{I}(X : Y|Z) \geq \mathrm{I}(f(X, R) : Y|Z).$$

Fact 2.4.7. *Let X, Y, Z, R be random variables such X and Z are independent when conditioning on R, namely when conditioning on $R = r$, for each possible values of r. Then* $\mathrm{I}(X : Y|Z, R) \geq \mathrm{I}(X : Y|R)$.

Fact 2.4.8. *Let W, X, Y, Z be random variables. Then*

$$\mathrm{I}(W : X|Z) - \mathrm{H}(Y|Z) \leq \mathrm{I}(W : X|Y, Z) \leq \mathrm{I}(W : X|Z) + \mathrm{H}(Y|Z).$$

Fact 2.4.9. *Let X be a random variable uniformly distributed over $\{0, 1\}^n$, and let J be some random variable on $\{0, 1, \ldots, n\}$ that may depend on X. Then :*

$$\mathrm{H}(X[1, J]|J) \leq \mathbb{E}(J) \textit{ and } \mathrm{H}(X[1, J]|X[J + 1, n]) \geq \mathbb{E}(J) - \mathrm{H}(J).$$

Similarly,

$$\mathrm{H}(X[J + 1, n]|J) \leq n - \mathbb{E}(J) \textit{ and } \mathrm{H}(X[J + 1, n]|X[1, J]) \geq n - \mathbb{E}(J) - \mathrm{H}(J).$$

Proof. $\mathrm{H}(X[1, J]|J) \leq \mathbb{E}(J)$ and $\mathrm{H}(X[J + 1, n]|J) \leq n - \mathbb{E}(J)$ are direct. The second part uses the first one as follows:

$$\begin{aligned} \mathrm{H}(X[1, J]|X[J + 1, n]) &= \mathrm{H}(X|J, X[J + 1, n]) - \mathrm{H}(X[J + 1, n]|J, X[J + 1, n]) \\ &= \mathrm{H}(X|J) - \mathrm{H}(X[J + 1, n]|J) \\ &\geq \mathrm{H}(X) - \mathrm{H}(J) - n + \mathbb{E}(J) = \mathbb{E}(J) - \mathrm{H}(J). \end{aligned}$$

□

2.4.2 Communication Complexity

Many different models of communication complexity have been studied in the literature : deterministic, randomized, non-deterministic, blackboard, one-way, message passing, number-in-hand, number-on-forehead, etc... Here we will only consider deterministic and randomized message passing number-in-hand communication complexity.

Players compute messages based on their input and the messages they received. We do not in general restrict them in their computing power, since in this thesis we only use this model to prove lower bounds. However it is common when building protocols to only consider those that have polynomial time complexity. We now define formally the notions of protocol and of communication complexity.

Definition 2.4.10. *Let $A_0, \ldots, A_{k-1}$ be players, and let player A_i have input X_i for each $0 \leq i \leq k-1$. A* deterministic communication protocol *is defined by a k-tuple of functions $P = (P_0, \ldots, P_{k-1})$. The output of the protocol is computed in the following way: at step t player $i_t = t \mod k$ computes $P_{i_t}(X_{i_t}, M_{i_t})$, where M_{i_t} is an initially empty word. The result, which we call a* message from A_{i_t} to $A_{i_t+1 \mod k}$*, is then added to $M_{i_t+1 \mod k}$ the list of messages received. The output of P is the final non-empty message sent.*

A randomized communication protocol *is defined similarly, excepts P_i is a function of X_i, M_i, R a source of randomness shared by all players, and R_i a source of randomness private to player A_i. R is called the public coins, and R_i the private coins.*

The public coins along with the set of all messages sent by players is called Π the transcript of P.

Definition 2.4.11. *Let $A_1, \ldots, A_k$ be players, and let player A_i have input X_i for each $1 \leq i \leq k$. Let f be a k-ary function, and P be a communication protocol for $A_1, \ldots, A_k$.*

For a deterministic protocol P, we say that P computes f if the protocol eventually stops on all k-tuples of inputs and the output is $f(X_1, \ldots, X_k)$.

For a randomized protocol P, we say that P computes f with probability at least $1 - \varepsilon$ if the protocol eventually stops on all k-tuples of inputs and all sources of randomness and the output is $f(X_1, \ldots, X_k)$ with probability at least $1 - \varepsilon$.

Definition 2.4.12. *The communication cost of a protocol P is the total size of messages sent by it in the worst case. The randomized (resp. deterministic) communication complexity of a function is the minimal communication cost of a randomized (resp. deterministic) communication protocol computing f.*

A useful tool for proving deterministic communication complexity lower bounds is the *cut-and-paste property*. It is trivial in this setting but we will see later how it can be adapted to a randomized setting.

Fact 2.4.13 (Cut and paste)**.** *Let P be a 2-player deterministic protocol. Let $\Pi(x, y)$ denote the transcript in P when Players A, B have resp. inputs x, y. Then if $\Pi(x, y) = \Pi(x', y')$ for some (x, y) and (x', y'), $\Pi(x, y') = \Pi(x', y)$.*

2.4.3 The Hellinger distance

The *Hellinger distance* is the 2-norm on probability distributions expressed as vectors. Because of its properties, it is especially useful to prove streaming lower bounds using communication complexity.

Definition 2.4.14. *Let μ and ν be probability distributions with their supports included in some set $\{a_1, \ldots, a_n\}$. Then the Hellinger distance between μ and ν is $\mathrm{h}^2(\mu, \nu) = 1/2 \sum_{i=1}^{k}(\sqrt{\mu(a_i)} - \sqrt{\nu(a_i)})$.*

Note that the square of the Hellinger distance is convex, and that the Hellinger distance and the ℓ_1 distance are connected by the usual relation between 2-norm and 1-norm.

Fact 2.4.15. $\mathrm{h}(X,Y)^2 \leq \frac{1}{2}\|X-Y\|_1 \leq \sqrt{2}\mathrm{h}(X,Y)$.

The mutual information between two random variables is connected to the Hellinger distance. Because it is a distance between distributions but we deal mostly with random variables, for simplicity we write X for the underlying distribution of a random variable X. We also write $X|_{Y=y}$ for the marginal distribution of X when the random variable Y takes the value y.

Lemma 2.4.16 (Average encoding)**.** *Let X, Y be random variables. Then $\mathbb{E}_{y\leftarrow Y}\, \mathrm{h}^2(X|_{Y=y}, X) \leq \kappa \mathrm{I}(X : Y)$, where $\kappa = \frac{\ln 2}{2}$.*

The Hellinger distance also generalizes the cut-and-paste property of deterministic protocols to randomized ones.

Lemma 2.4.17 (Randomized Cut and paste)**.** *Let P be a 2-player randomized protocol. Let $\Pi(x,y)$ denote the random variable representing the transcript in P when players A, B have resp. inputs x, y. Let X, X', Y, Y' be random variables taking their values in the input space of A for the first two and of B for the last two.*

Then $\mathrm{h}(\Pi(X,Y), \Pi(X',Y')) = \mathrm{h}(\Pi(X,Y'), \Pi(X',Y))$.

For a reference on these results, see [26].

Chapter 3

Algorithms and Upper Bounds

3.1 Recognizing Priority Queues with one pass in each direction

3.1.1 Previous Results

3.1.1.1 A major tool: Hashcodes

In order to check that in a given subword of the input, the insertions and extractions match, we want a linear sketch that uses less memory than the whole factor. For this we use a hash function based on the one used by the Karp-Rabin algorithm for pattern matching.

For the rest this section, let p be a prime number in $\{\max(2U+1, n^{c+1}), \dots, 2\max(2U+1, n^{c+1})\}$, for some fixed constant $c \geq 1$, and where U is the maximal priority in the input and n the size of the input. Let α be a randomly chosen integer in $[0, p-1]$. Since our hash function is linear we only define it for single insertion/extraction as

$$\mathrm{hash}(\mathtt{ins}(a)) = \alpha^a \mod p, \quad \text{and} \quad \mathrm{hash}(\mathtt{ext}(a)) = -\alpha^a \mod p.$$

This is the unique use of randomness in all the algorithms presented in this section.

A hashcode h *encodes* a sequence v if $h = \mathrm{hash}(v)$ as a formal polynomial in α. In that case we say that each letter $v[i]$ in v is encoded in h. Moreover v is *strongly balanced* if the same integers have been inserted and extracted[1]. In that case it must be that $h = 0$. We also say that h is strongly balanced it it encodes a strongly balanced sequence w. The converse is also true with high probability by the Schwartz-Zippel lemma.

Fact 3.1.1. *Let v be some sequence that is not strongly balanced. Then* $\Pr(\mathrm{hash}(v) = 0) \leq \frac{N}{p} \leq \frac{1}{N^c}$.

3.1.1.2 Best unidirectional algorithm

The best unidirectional algorithm is due to Chakrabarti, Cormode, Kondapalli and McGregor in [10]. It uses $\tilde{\mathrm{O}}(\sqrt{n}/p)$ memory for p passes on an input of size n, and does not require timestamps[2]. In that same article, they show using the lower bound on the complexity of recognizing Dyck language that this bound is tight up to a polylogarithmic factor if the algorithm does not use timestamps. We will show in Section 4.1 it is in fact tight even in that case.

[1] v is (not necessarily strongly) balanced if it contains as many insertions as extractions, but this notion is not useful in this section. We use it in Section 3.2 however.

[2] An earlier article [12] gave an algorithm using timestamps.

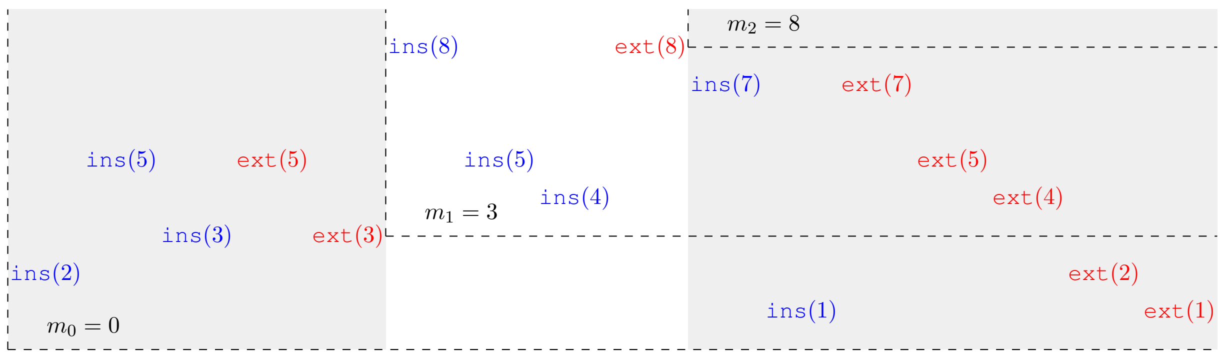


Figure 3.1: Algorithm 3.1 encodes each element in the hashcode matching the earliest possible block in which it could have appeared. For example, ext(2) is encoded in h_0 because $2 < 3$ and the first ext(5) is encoded in h_0 because it comes before ext(3). However, ext(4) comes after ext(3) and $3 < 4$, therefore it is encoded in h_1.

```
Data structure:
$i \leftarrow 0$ // block index corresponding to a valley
$m_0 \leftarrow 0$ // $m_i$ minimum value of extractions before the $i$-th valley
$h_0 \leftarrow 0$ // $h_i$ hashcode for $i$-th block
For $i \geq 1$, $\delta_i^{\rightarrow}$ counts the number of appearances of $m_i$ after the $i$-th valley
$x$ is the current symbol and $y$ the previous symbol
Code:
While $u$ not finished
    $x \leftarrow \text{Next}(u)$ // Read and process next symbol $x$
    If $x = \text{ins}(a)$ then
        If $y = \text{ext}(b)$ then // New valley
            $i \leftarrow i+1$, $m_i \leftarrow b$, $h_i \leftarrow 0$, $\delta_i^{\rightarrow} \leftarrow 0$
        $k \leftarrow \max\{j \leq i : m_j < a\}$ // Earliest possible valley before $x$
        Update($h_k, x$)
        For $l$ in $\{k, \ldots, i\}$
            If $m_l = a$ then $\delta_l^{\rightarrow} \leftarrow \delta_l^{\rightarrow} + 1$
    Else $x = \text{ext}(a)$ then
        If $y = \text{ext}(b)$ and $a > b$ then Reject // Check local order
        $k \leftarrow \max\{j \leq i : m_j < a\}$ // Earliest possible valley before $x$
        Update($h_k, x$)
        For $l$ in $\{k, \ldots, i\}$
            If $m_l = a$ then $\delta_l^{\rightarrow} \leftarrow \delta_l^{\rightarrow} - 1$
For all $m_i$ and $\delta_i^{\rightarrow}$:
    If $m_i \neq 0$ or $\delta_i^{\rightarrow} > 0$ then Reject
Accept
```

Algorithm 3.1: One-Pass Algorithm Recognizing Priority Queues.
The differences with Algorithm 3.1 are highlighted in red.

Theorem 3.1.2 (Chakrabarti, Cormode, Kondapalli, McGregor)**.** *There is a unidirectional* 1*-pass randomized streaming algorithm recognizing* $\mathrm{PQ}(U)$ *with memory space* $\mathrm{O}((\log U + \log n)\sqrt{n})$*, and one-sided bounded error* n^{-c}*, for inputs of length* n *and any constant* $c > 0$*.*

We give a slightly modified form of the algorithm for one pass used in [10] as its structure will be necessary to understand the two-pass algorithm. We define a *valley* as places in the input where a sequence of extractions ends and a sequence of insertion begins. *Blocks* are factors delimited by two consecutive valleys. The final algorithm is based on algorithm 3.1, which maintains a hashcode for each block. **It encodes each insertion and extraction in the hashcode of the earliest block where it could have occurred with the input remaining a priority queue** (see Figure 3.1). At the end, it checks that all hashcodes evaluate to zero. Another test can be necessary in some cases for elements that appear multiple times, as hashcodes are linear and do not depend on the order of elements encoded. Because Algorithm 3.1 maintains as many hashcodes as there are blocks, the memory it requires is $\mathrm{O}((\log n + \log U)r)$, where r is the number of blocks in the input.

The real algorithm is then obtained by having another algorithm pre-process the next $\sqrt{n}$ symbols of the stream, checking their local consistency and rearranging them so that they contain at most two valleys. This can be further simplified into one valley by removing matching insertions and extractions. Algorithm 3.2 does this using memory space $\mathrm{O}(\sqrt{n}\log U)$ (see Figure 3.2). Because it guarantees that Algorithm 3.1 will have an input with at most $2\sqrt{n}$ blocks, this results in an algorithm that uses at most memory space $\mathrm{O}((\log n + \log U)\sqrt{n})$.

```
Input:
$v$ a subword of $u$
Data structure:
$I \leftarrow \emptyset$ // multiset of unmatched insertions
$E \leftarrow \emptyset$ // multiset of unmatched extractions
$m \leftarrow 0$ // min of extractions seen so far
Code:
For $x$ in $v$
    If $x = \mathrm{ins}(a)$ then $I \leftarrow I \cup \{a\}$
    If $x = \mathrm{ext}(a)$ then
        If $a < \max(I)$ then Reject
        If $a = \max(I)$ then $I \leftarrow I \setminus \{a\}$
        If $a > \max(I)$ then
            If $a > m$ then Reject
            $E \leftarrow E \cup \{a\}$
        $m = \min(m, v)$
Output $\mathrm{ext}(E_{|E|}), \dots, \mathrm{ext}(E_1), \mathrm{ins}(I_1), \dots, \mathrm{ins}(I_{|I|})$
```

Algorithm 3.2: Algorithm for Checking Local Consistency of Priority Queues

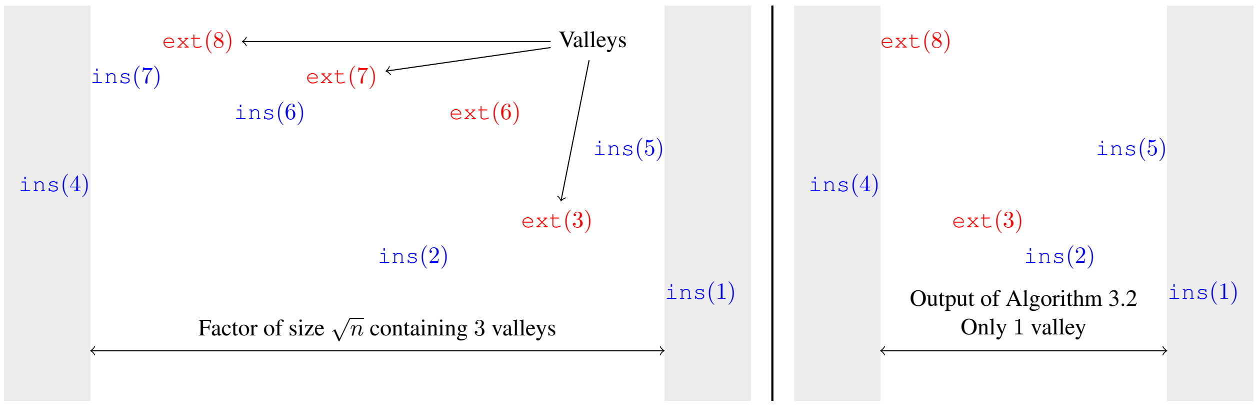


Figure 3.2: Transformation performed by Algorithm 3.2 on a factor of u of length $\sqrt{n \log n}$. Note that the occurrence of $\texttt{ins}(4)$ on the left of the picture is incompatible with the occurrence $\texttt{ext}(3)$ later, but because Algorithm 3.2 only performs local checks it does not reject : Algorithm 3.1 will however reject with high probability when the matching occurrence of $\texttt{ext}(4)$ is encoded in the hashcode for block i (if the current valley is the i-th valley) whereas $\texttt{ins}(4)$ is encoded no later than in block $i - 1$.

3.1.2 Our algorithm

As said above, we want to use an approach similar to the one in [32] with passes in each direction where the past is compressed into a logarithmic number of blocks. However, a crucial difference between priority queues and the Dyck language is that $u \in \mathrm{PQ}$ does not imply $\bar{u} \in \mathrm{PQ}$. In particular,, we first need to devise an algorithm for recognizing $\mathrm{Reverse}(u)$ when we have $u \in \mathrm{PQ}$. This algorithm is designed similarly to Algorithm 3.1. We do not need to improve it with Algorithm 3.2, as the sliding widow it provides would only help us get to $\tilde{\mathrm{O}}(\sqrt{n})$ memory, and we want $\mathrm{polylog} n$ memory, which will be achieved instead by modifying this algorithm, as well as Algorithm 3.1 as described above.

We will then show that using a modified form of Algorithm 3.1 and Algorithm 3.3 with blocks as in [32], namely Algorithms 3.4 and 3.5, we can detect any $u \notin \mathrm{PQ}$ with high probability before the blocks containing the witnesses of an error merge.

The analysis is complicated by the fact that some insertions and extractions may appear multiple times. For each value a, if $u \in \mathrm{PQ}$ then the occurrences of $\texttt{ins}(a)$ and $\texttt{ext}(a)$ in u form a well-parenthesized word. Let us define the notion of *a-balance*:

Definition 3.1.3. *Let $v \in \Sigma^n$ and $a \in \{1, \dots, U\}$. We say that v is a-*balanced *if it contains as many $\texttt{ins}(a)$ as $\texttt{ext}(a)$, i.e. $|\{t|v[t] = \texttt{ins}(a)\}| = |\{t|v[t] = \texttt{ext}(a)\}|$. It is* $\texttt{ins}(a)$-unbalanced *(resp. $\texttt{ext}(a)$-unbalanced) if it has an excess of $\texttt{ins}(a)$ (resp. of $\texttt{ext}(a)$), i.e. $|\{t|v[t] = \texttt{ins}(a)\}| > |\{t|v[t] = \texttt{ext}(a)\}|$ (resp. $|\{t|v[t] = \texttt{ins}(a)\}| < |\{t|v[t] = \texttt{ext}(a)\}|$).*

Note that if u in PQ, then none of its prefixes are $\texttt{ext}(a)$-unbalanced. Also note that a word in Σ^* is strongly balanced if and only if it is a-balanced for all $a \in \{1, \dots, U\}$

3.1.2.1 One-reverse-pass algorithm for PQ

Our one-reverse-pass algorithm, uses memory space $\mathrm{O}(r)$, where r is the number of valleys in w. As stated above, while we could use less memory space by combining it with Algorithm 3.2, this serves no purpose as our algorithm will eventually be modified for the bidirectional two-passes algorithm.

Theorem 3.1.4. *There is a* 1*-reverse-pass randomized streaming algorithm for* $\mathrm{PQ}(U)$ *with memory space* $\mathrm{O}(r(\log N + \log U))$ *and one-sided bounded error* N^{-c}*, for inputs of length* N *with* r *valleys, and any constant* $c > 0$.

As with algorithm 3.1, Algorithm 3.3 decomposes the input u into blocks, and associates a hashcode h_i and two integers m_i and $\delta_i^{\leftarrow}$ to each block. As before m_i is the minimum of extractions in the block. Here $\delta_i^{\leftarrow}$ counts how many times m_i has been encoded in the block (we do not keep track of the absolute number of occurrences of m_i since the block was created however, as we did with $\delta_i^{\rightarrow}$ in Algorithm 3.1). The main difference lies in the fact that instead of encoding both insertions and extraction to the earliest possible block for the insertion, **extractions are encoded in their own block, and insertions are encoded in the block of their matching extraction**, which can be known unambiguously. Figure 3.3 illustrates this.

We first state a crucial property of Algorithm 3.3, and then show that it satisfies Theorem 3.1.4.

Lemma 3.1.5. *Consider Algorithm 3.3 right after processing* $u[t] = \mathtt{ins}(a)$*. Assume some* $u[t'] = \mathtt{ext}(a)$ *has been already processed (i.e.* $t' > t$*). Let* $h_k, h_{k'}$ *be the respective hashcodes encoding* $u[t], u[t']$*. Then* $k = k'$ *if and only if all* $\mathtt{ext}(b)$ *occurring between* $u[t]$ *and* $u[t']$ *satisfy* $b \geq a$*, and no factor* $u[t+1, t'']$ *with* $u[t'']$ *in a block left of* k' *is* $\mathtt{ext}(a)$*-unbalanced.*

Proof. The algorithm always encodes extractions in the hashcode matching their block. Therefore the block index when $u[t']$ is processed is k'. Let $\tilde{k}$ be the current block index when $u[t]$ is processed.

Let us first assume $k = k'$. Then no block left of the k'-th can have been a viable candidate at line 18. In particular, for all $k' < l \leq \tilde{k}$, we have $m_l \geq a$. Since m_l is the minimum of extractions in the block l, and extractions left of t' in the block k' have to be greater than a, then all $\mathtt{ext}(b)$ occurring between $u[t]$ and $u[t']$ satisfy $b \geq a$. Now assume there exists some $u[t'']$ in a block l with $k' < l \leq \tilde{k}$ such that $u[t+1, t'']$ is $\mathtt{ext}(a)$-unbalanced. Let us choose the leftmost possible block, i.e. the largest possible l. Then we know $m_l \leq a$, as block l must contain at least one $\mathtt{ext}(a)$, and $m_l \geq a$ from our augment above. Since $u[t+1, t'']$ is $\mathtt{ext}(a)$-unbalanced and l is the leftmost block with such a t'', then after $u[t+1]$ is processed we must have $\delta_l^{\leftarrow} > 0$, and $k' = l$ which is a contradiction. Therefore for all $k' < l < \tilde{k}$, for all indices t'' such that $u[t'']$ is in l, $u[t+1, t'']$ is not $\mathtt{ext}(a)$-unbalanced.

Now let us assume $k \neq k'$. Then for block k to be a viable candidate at line 18 when $u[t]$ is processed, we must have either $m_k < a$, or $m_k = a$ and $\delta_k^{\leftarrow} > 0$. The first implies there exists an extraction $u[t''] = \mathtt{ext}(m_k)$ with $t < t'' < t'$ and $m_k < a$. The second implies $u[t+1, t'']$ is $\mathtt{ext}(a)$-unbalanced. $\square$

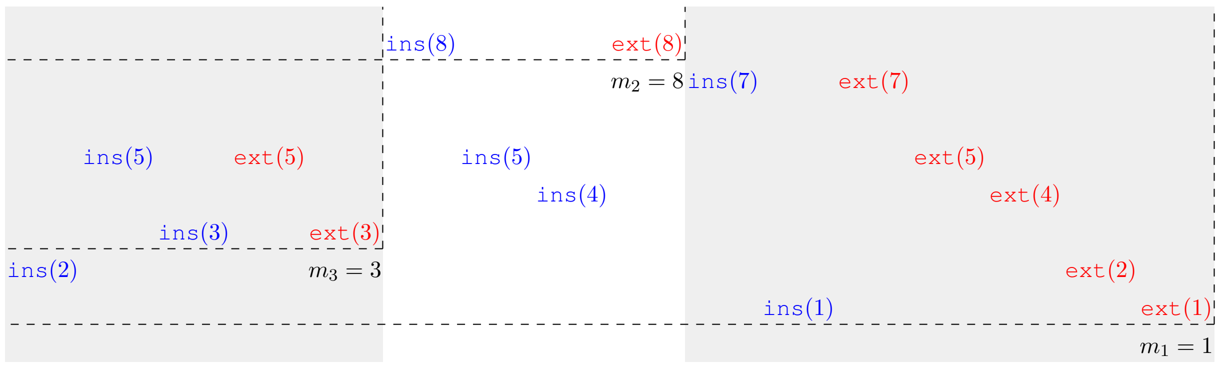


Figure 3.3: Algorithm 3.3 running on the same input as in Figure 3.1. As can be seen extractions are always encoded in their own block, unlike with Algorithm 3.1.

```
1  **Data structure:**
2  $i \leftarrow 0$ // block index corresponding to a valley
3  $m_0 \leftarrow 0$ // $m_i$ minimum value of extractions before the $i$-th valley
4  For $i \geq 1$, $h_i$ hashcode for $i$-th block
5  For $i \geq 1$, $\delta_i^{\leftarrow}$ counts the number of times $m_i$ is encoded in $h_i$
6  $x$ is the current symbol and $y$ the symbol to the right in $u$
7  **Code:**
8  While Reverse($u$) not finished
9      $x \leftarrow$ Next(Reverse($u$))
10     If $x = \texttt{ext}(a)$
11         If $y = \texttt{ins}(b)$ OR $x$ rightmost letter of $u$ //This is a valley.
12             $i \leftarrow i+1$, $m_i \leftarrow a$, $h_i \leftarrow 0$, $\delta_i^{\leftarrow} \leftarrow 1$  //We start a new block.
13         Else $y = \texttt{ext}(b)$
14             If $(a > b)$ then **Reject**  //Check local order
15         Update($h_i, x$)
16         If $m_i = a$ then $\delta_i^{\leftarrow} \leftarrow \delta_i^{\leftarrow} + 1$
17     Else $x = \texttt{ins}(a)$
18         $k \leftarrow \max\{j \leq i : m_j < a$ OR $(m_j = a$ AND $\delta_j > 0)\}$  //only block where $\bar{a}$ can be
19         If $k = 0$ then **Reject**
20         Update($h_k, x$)
21         If $m_k = a$ then $\delta_k^{\leftarrow} \leftarrow \delta_k^{\leftarrow} - 1$
22 For all $h_j$:
23     If $h_j \neq 0$ then **Reject**
24 **Accept**
```

Algorithm 3.3: One-reverse-pass algorithm for recognizing priority queues.
The differences with Algorithm 3.1 are highlighted in red. Note that the role of insertions and extractions is mostly switched, except for the test on the local order (line 14 here and line 18 in Algorithm 3.1) which only occurs when reading an extraction.

Proof of Theorem 3.1.4. We show that Algorithm 3.3 suits the conditions. Let $u \in \mathrm{PQ}(U)$. Then u always passes the test at line 14. Moreover, by Lemma 3.1.5, each insertion $\texttt{ins}(a)$ is necessarily in the same hashcode than its matching extraction $\texttt{ext}(a)$. Therefore, all hashcodes equal 0 at line 23 since they are strongly balanced. In conclusion, the algorithm accepts w with probability 1.

Assume now that $u \notin \mathrm{PQ}$. First we show that unbalanced inputs are rejected with high probability, that is at least $1 - N^{-c}$, at line 23, if they are not rejected before. Indeed, since each letter of u is encoded in some h_j, at least one h_j must be unbalanced. Then by Fact 3.1.1, the algorithm rejects w.h.p. We end the proof assuming u balanced. We remind that we process the stream from right to left. The two remaining possible errors are: (1) For some a, there exists a prefix of u that is $\texttt{ext}(a)$-unbalanced; and (2) $\texttt{ext}(a), \texttt{ext}(b), \texttt{ins}(a)$ are processed in this order with $b < a$ and possibly intermediate insertions/extractions. In both cases, we show that either some hashcodes are unbalanced at line 23, and therefore fail the test w.h.p by Fact 3.1.1, or the algorithm rejects with one of the other tests.

Consider case (1). Since u is strongly balanced, there must be $u[t, n]$ a suffix of u that is $\texttt{ins}(a)$-unbalanced. Consider the largest such t. Let k be the block where $u[t] = \texttt{ins}(a)$ is encoded. There is at least one valley between $u[t]$ and the next $\texttt{ext}(a)$, which means they will not be encoded in the same block, and h_k will be unbalanced at line 23. Consider now case (2). Lemma 3.1.5 gives that $\texttt{ext}(a)$ and $\texttt{ins}(a)$ are encoded in different hashcodes, that are again unbalanced at line 23. □

3.1.2.2 Block structure for a bidirectional algorithm

Our algorithm uses an approach similar to the one used in [32] for Dyck languages: by making a pass in each direction, it is possible to compress the part of the input already seen in a stack of blocks of logarithmic height, where older blocks are longer and the size of each block is a power of 2. While we will not extensively describe the algorithm for recognizing Dyck languages here, the block data structure is the same as the one our algorithm uses.

A block of size 2^i is of the form $[(q-1)2^i+1, q2^i]$, for $1 \leq q \leq N/2^i$. Two such blocks are always disjoint unless one is included in the other. We decompose dynamically the part of the input u that has been processed into a stack of blocks as follows. Each new letter of u defines a new block, put on top of the stack. When two blocks have same size, they merge. Note that only the two topmost blocks may merge (although the resulting block may merge again with the block that is now second from the top). Because the size of each block is a power of 2 and at most two blocks have the same size (before merging), there are at most $\log N + 1$ blocks at any time. To each block, we associate among other things a hashcode.

At the end there will remain only one block, and if the input is strongly balanced then the hashcode associated to the block will evaluate to 0. Therefore it is important to perform checks on hashcodes as soon as we know from the other data associated to the block (notably, the minimal height of a parenthesis in the block for $\textsc{Dyck}[2]$, or as we will see, the minimal extraction for PQ) that an hashcode should evaluate to 0.

From now on, we assume without loss of generality that the input size is a power of 2. Indeed, any strongly balanced input will have even size. The algorithm can simply pad the end of the input with repetitions of the pattern $\texttt{ins}(1)\texttt{ext}(1)$ until the size is a power of 2.

Lemma 3.1.6. *The same blocks appear in the pass from left to right and in the pass from right to left. Furthermore, if blocks B and C merge together in one of those passes, they also merge together in the other.*

Proof. We have $n = 2^k$ for some k. Any block that appears on the pass from left to right is of the form $[2^i(q-1)+1, 2^i q]$ with q an integer. If q is even, it will merge with a block on the left, otherwise with a block on the right. Any block appearing on the pass from right to left is of the form $[2^n - 2^i q' + 1, 2^n - 2^i(q'-1)]$. It suffices to take $q' = 2^{n-i} - q + 1$ to have both blocks be equal. This block will merge with a block on the left if q' is odd, otherwise with a block on the right. Since q' and q are of different parities, the block will merge with the same block in both directions. □

Lemma 3.1.6 point is crucial for the analysis, as merging blocks can potentially make us lose the information needed to detect an error. The main difficulty (which is significantly harder for PQ than for $\textsc{Dyck}[2]$) is to show that any error that would be lost after the blocks are merged is detected in at least one of the passes with high probability. The main idea is shown on Figure 3.4.

We also want to show that the padding will not affect the time per processing item. Because the padding is deterministic and only contains $\texttt{ins}(1)$ and $\texttt{ext}(1)$, in the pass from left to right all its elements will be encoded in the last block, and cancel each other out immediately. Therefore all the algorithm has got to do is to look at the size of the two blocks on top of the stack, increase the size of the top one until it is of the same size as the second one from top (which takes at most $\log(n)$ time), merge and repeat. The whole operation takes at most $\text{polylog}(n)$ time. For the pass from right to left, the same is true except it is made even easier by the lack of input before the padding, so the state depending on the size of the padding could even be pre-computed.

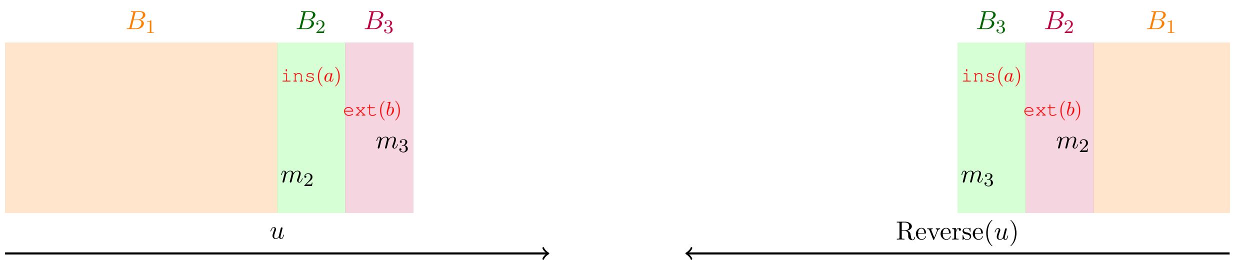


Figure 3.4: If the size of u is a power of 2, then the block decomposition is the same in both direction. In particular if the blocks about to merge contain an error, the (very simplified) general idea is that one of the blocks will have a smaller minimal extraction, and that this will cause us the algorithm to evaluate the hashcode in the corresponding direction and reject. In this case, the rejection would happen on the pass from right to left, as the hashcode started in the green block would be finished and could be checked at 0.

3.1.2.3 Bidirectional two-passes algorithm

Theorem 3.1.7. *There is a bidirectional* 2*-pass randomized streaming algorithm recognizing* $\mathrm{PQ}(U)$ *with memory space* $\mathrm{O}((\log n)(\log U + \log n))$*, time per processing item* $\mathrm{polylog}(n, U)$*, and one-sided bounded error* n^{-c}*, for inputs of length* n *and any constant* $c > 0$.

Our algorithm performs one pass in each direction using Algorithms 3.4 and 3.5. We use the following description of a block B: its hashcode h_B, the minimum m_B of its extractions, a counter $\delta_B^{\rightarrow}$ of $\delta_B^{\leftarrow}$ (depending on the direction of the pass) to deal with multiple occurrences of m_B (we will define them more precisely later), and its size ℓ_B. For the analysis, let t_B be an index such that $u[t_B] = \texttt{ext}(m_B)$. Note that t_B may not be unique. On the pass from right to left, all extractions from the block and matching insertions are encoded in h_B, similarly to Algorithm 3.3. On the pass from left to right, insertions are encoded in the hashcode of the earliest possible block where they could have been, and extractions are encoded with their matching insertions, similarly to Algorithm 3.1. The minimums $(m_B)_B$ are used to decide where to encode values.

At the end, only one block is left, and if u is strongly balanced then the only hashcode will evaluate to 0. Therefore we must check $h_B = 0$ during the execution, as soon as we know from m_B and δ_B that the hashcode should encode matching insertions and extractions. In the case of Algorithm 3.5 for the pass from right to left, in some cases we even need to evaluate a hashcode h_B when some insertions $\texttt{ins}(m_B)$ are missing. However with the counter $\delta_B^{\leftarrow}$ we know their number and can update h_B before evaluating. These tests are performed at line 26 of Algorithm 3.4 and line 25 of Algorithm 3.5.

Similarly to its role in Algorithm 3.1, $\delta_B^{\rightarrow}$ counts the difference between the number of occurrences of $\texttt{ins}(m_B)$ and the number of occurrences of $\texttt{ext}(m_B)$ since the earliest possible choice for t_B. Algorithm 3.4 does not use $\delta_B^{\rightarrow}$ to decide where to encode a particular element, only to check that if $u[t] = \texttt{ext}(a)$ with $a < m_B$, then $u[t_B + 1, t]$ is not $\texttt{ins}(m_B)$-unbalanced. This is crucial in Lemma 3.1.10. On the contrary, like in Algorithm 3.3, $\delta_B^{\leftarrow}$ is used to assign elements to a particular hashcode. When $\texttt{ins}(a)$ is read and the leftmost block satisfying $m_B \leq a$ in fact satisfies $m_B = a$, Algorithm 3.5 only encodes $\texttt{ins}(a)$ in h_B if there is still a spot left for an insertion of a, i.e. if $\delta_B^{\leftarrow} > 0$. This value represents the difference between the number of $\texttt{ext}(m_B)$ and the number of occurrences of $\texttt{ins}(m_B)$ encoded in h_B. Its value can never be negative, and it is also used as explain above when we need to evaluate a hashcode but h_B is still missing some instances of $\texttt{ins}(m_B)$.

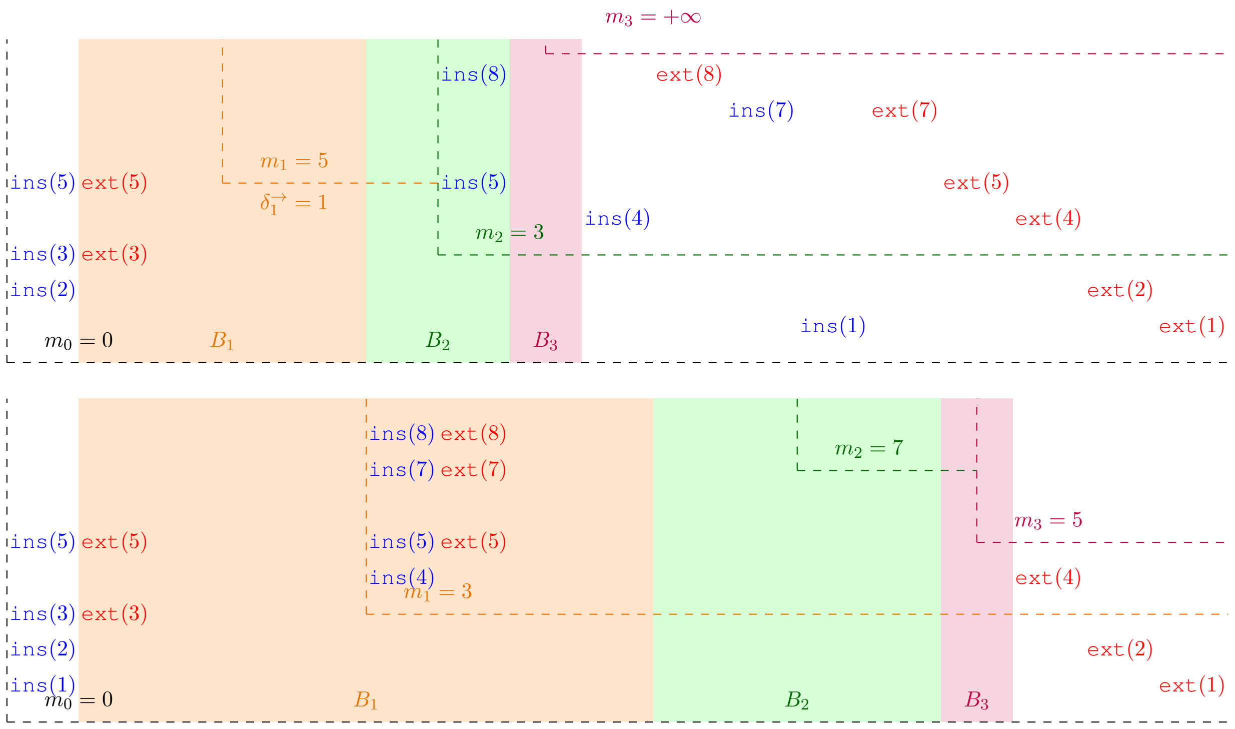


Figure 3.5: Evolution of the information contained in the blocks when the example from Figures 3.1 and 3.3 is read on the pass from left to right. The colored area represents the interval covered by the block, the north-east corners delimited by dashed lines represent values encoded in a hashcode. Note that on when $\texttt{ext}(3)$ was first read, B_1 covered the same part of the input as on the upper figure, and Algorithm 3.4 checked that $h_1 = 0$ and $\delta_1^{\rightarrow} \leq 0$. Later on $\delta_1^{\rightarrow}$ increased, but that information was lost in a block merge.

Note that ℓ_B is the size of the factor of u attributed to the block, not the number of letters of u encoded in h_B. Only h_B and δ_B can change without B being merged with another block. When there is some ambiguity, we denote by $h_B^{\rightarrow}$ and $h_B^{\leftarrow}$ the hashcodes for the left-to-right and right-to-left passes. Observe that m_B, t_B, ℓ_B are identical in both directions. Figure 3.5 shows how the information associated to blocks can evolve as Algorithm 3.4, the pass from left to right, is run on the input.

As with the 0-th block in Algorithm 3.1, in Algorithm 3.4 the stack S starts with a block of size 0 that will never merge or match a non-empty factor of u.

Proof of Theorem 3.1.7. We show that execution of both Algorithms 3.4 and 3.5 suits the conditions. The space constraints are satisfied because elements of S have size $\mathrm{O}(\log N + \log U)$ and S has size $\mathrm{O}(\log N)$. The processing time is from inspection.

As with Algorithms 3.1 and 3.3, inputs in $\mathrm{PQ}(U)$ are accepted with probability 1, and unbalanced inputs are rejected with high probability (at least $1 - N^{-c}$).

Let $u \notin \mathrm{PQ}$ be strongly balanced. For ease of notations, we create fictional letters corresponding to the empty block at the bottom of S for the pass from left to right: let $u[-1] = \texttt{ins}(-\infty)$ and $u[0] = \texttt{ext}(-\infty)$. Then because u is not a valid priority queue history, there are $\tau < \rho$ such that $u[\tau] = \texttt{ext}(b)$, $u[\rho] = \texttt{ext}(a)$, where $a > b$ and $u[\tau, \rho]$ is $\texttt{ext}(a)$-unbalanced (i.e. $\{t | u[t] = \texttt{ins}(a), \tau < t < \rho\}$ is smaller than $\{t | u[t] = \texttt{ext}(a), \tau < t < \rho\}$).

Among such pairs (τ, ρ), consider the ones with the smallest ρ. From those, select the one with the largest τ. Let B, C be the largest possible disjoint blocks such that τ is in B and ρ in C. Then B and C have same size, are contiguous, and by Lemma 3.1.6 they are simultaneously present in each direction before they merge.

Consider $\tau < v < \rho$ such that $u[v] = \mathtt{ext}(c)$. If $c < a$, then since $u[\tau, \rho]$ is $\mathtt{ext}(a)$ unbalanced then either $u[v, \rho]$ is $\mathtt{ext}(a)$-unbalanced, which contradicts the maximality of τ as v is a better candidate, or $u[\tau, v]$ is $\mathtt{ext}(a)$-unbalanced. In the latter case there is a least one $t' \in [\tau, v]$ such that $u[t'] = \mathtt{ext}(a)$. The largest such t' still verifies that $u[\tau, t']$ is $\mathtt{ext}(a)$-unbalanced, which contradicts the minimality of ρ. Therefore $c \geq a$. In particular, we can assume that the indices t_B and t_C of block minimal extractions m_B and m_C satisfy $t_C \geq \rho$ and $t_B \leq \tau$.

```
Data Structure:
$S \leftarrow [(0,0,0,0)]$  // $S$ starts with a 0-size block.
Elements of $S$ are of the form $B = (h_B, m_B, \delta_B^{\rightarrow}, \ell_B)$.
In this direction $\delta_B^{\rightarrow}$ counts the occurrences of $m_B$ since block $B$.
$x$ is the current letter of $u$.
Code:
While $u$ not finished
     Read(next letter $x$ on stream) // See below
     While the 2 topmost elements of $S$ have same block size $\ell$
         $(h_1, m_1, \delta_1^{\rightarrow}, \ell) \leftarrow S.\mathrm{pop}()$,  $(h_2, m_2, \delta_2^{\rightarrow}, \ell) \leftarrow S.\mathrm{pop}()$
         If $m_1 < m_2$
             $S.\mathrm{push}(h_1 + h_2 \mod p, m_1, \delta_1^{\rightarrow}, 2\ell)$
         Else $m_2 \leq m_1$
             $S.\mathrm{push}(h_1 + h_2 \mod p, m_2, \delta_2^{\rightarrow}, 2\ell)$
If $S = [(0,0,0,0), (0, m, \delta^{\rightarrow}, N)]$ for some $\delta^{\rightarrow} \leq 0$ then Accept else Reject

Function Read($x$):
Case $x = \mathtt{ins}(a)$ // When reading an insertion
     Let $(h, m, \delta^{\rightarrow}, \ell)$ be the topmost item of $S$ with $a > m$
     $h \leftarrow \mathtt{Update}(h, v)$
     For all $(h', m', \delta'^{\rightarrow}, \ell')$ in $S$ such that $a = m$
         $\delta'^{\rightarrow} \leftarrow \delta'^{\rightarrow} + 1$
     $S.\mathrm{push}(0, +\infty, 0, 1, 0)$
Case $x = \mathtt{ext}(a)$ // When reading an extraction
     For all items $(h, m, \delta^{\rightarrow}, \ell)$ on $S$ such that $m \geq a$:
        If $h \neq 0$ or $(m > a$ and $\delta^{\rightarrow} > 0)$ then Reject
     Let $(h, m, \delta^{\rightarrow}, \ell)$ be the topmost item of $S$ from top such that $a > m$
     $h \leftarrow \mathtt{Update}(h, v)$
     For all $(h', m', \delta'^{\rightarrow}, \ell')$ in $S$ such that $a = m$
         $\delta'^{\rightarrow} \leftarrow \delta'^{\rightarrow} - 1$
     $S.\mathrm{push}(0, a, 0, 1)$
```

Algorithm 3.4: Pass from left to right.
The differences with Algorithm 3.5 are highlighted in red.

Similarly, there exists $\tau' < \tau$ such that $u[\tau'] = \texttt{ins}(b)$ and $u[\tau', \tau]$ is b-balanced. Let us choose the largest such τ'. If there exists $\rho' < \rho$ such that $u[\rho'] = \texttt{ins}(a)$ and $u[\rho', \rho]$ is a-balanced, we also pick the largest such ρ', otherwise we pick the smallest ρ' such that $u[\rho'] = \texttt{ins}(a)$ and $u[\rho, \rho']$ is a-balanced, which is guaranteed to exist as u is strongly balanced.

We distinguish three cases based on the position of index ρ' (see Figure 3.6): $\rho' \notin [t_B, t_C]$, $t_B < \rho' < \tau$, and $\rho < \rho' < t_C$. These cases determine in which hashcode $\texttt{ins}(a)$ is encoded. We prove that in each of these cases, either Algorithm 3.4 or Algorithm 3.5 rejects with high probability in Lemmas 3.1.8, 3.1.9 and 3.1.10. This concludes the proof of Theorem 3.1.7. □

```
1  **Data Structure:**
2  $S \leftarrow []$   // $S$ starts as an empty stack
3  Elements of $S$ are of the form $B = (h_B, m_B, \delta_B^{\leftarrow}, \ell_B)$
4  In this direction $\delta_B^{\leftarrow}$ counts the occurrences of $m_B$ encoded in $h_B$
5  $x$ is the current letter of $u$
6  **Code:**
7  While Reverse($u$) not finished
8      Read(next letter $x$ on stream) // See below
9      While the 2 topmost elements of $S$ have same block size $\ell$
10         $(h_1, m_1, \delta_1^{\leftarrow}, \ell) \leftarrow S.\mathsf{pop}()$,  $(h_2, m_2, \delta_2^{\leftarrow}, \ell) \leftarrow S.\mathsf{pop}()$
11         If $m_1 < m_2$
12             $S.\mathsf{push}(h_1 + h_2 \mod p, m_1, \delta_1^{\leftarrow}, 2\ell)$
13         Else if $m_2 < m_1$
14             $S.\mathsf{push}(h_1 + h_2 \mod p, m_2, \delta_2^{\leftarrow}, 2\ell)$
15         Else $m_1 = m_2$
16             $S.\mathsf{push}(h_1 + h_2 \mod p, m_1, \delta_1^{\leftarrow} + \delta_2^{\leftarrow}, 2\ell)$
17 If $S = [(0, m, 0, N)]$ then **Accept** else **Reject**
18
19 Function Read($x$):
20 Case $v = \mathrm{ext}(a)$ // When reading an extraction
21     For all items $(h, m, \delta^{\leftarrow}, \ell)$ on $S$ such that $m \geq a$:
22         If $m = a$
23             For $j$ from 1 to $\delta^{\leftarrow}$
24                 $h \leftarrow \texttt{Update}(h, \texttt{ins}(m))$
25         If $h \neq 0$ then **Reject**
26         If $m = a$
27             For $j$ from 1 to $\delta^{\leftarrow}$
28                 $h \leftarrow \texttt{Update}(h, \texttt{ext}(m))$
29     $S.\mathsf{push}(\mathrm{hash}(v), a, 1, 1)$
30 Case $x = \texttt{ins}(a)$ // When reading an insertion
31     Let $(h, m, \delta^{\leftarrow}, \ell)$ be the topmost item of $S$ with $a > m$ or $(a = m$ and $\delta^{\leftarrow} > 0)$
32     $h \leftarrow \texttt{Update}(h, v)$
33     If $a = m$ then $\delta^{\leftarrow} \leftarrow \delta^{\leftarrow} - 1$
34     $S.\mathsf{push}(0, +\infty, 0, 1)$
```

Algorithm 3.5: Pass from right to left.
The differences with Algorithm 3.4 are highlighted in red. Note that the role of insertions and extractions is mostly switched, except for the rejection test (line 25 here and line 26 in Algorithm 3.4) which only occurs when reading an extraction.

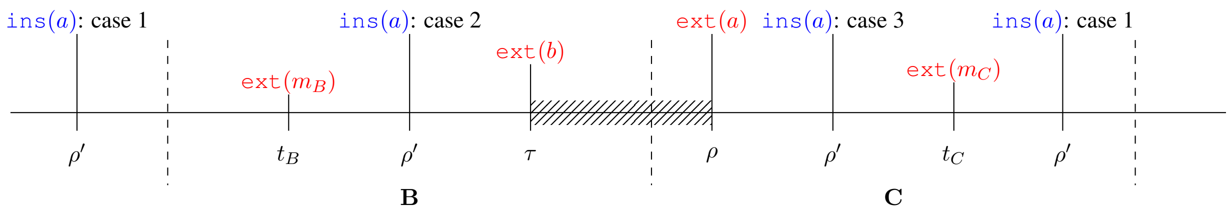


Figure 3.6: Relative positions of insertions and extractions in $u \notin$ PQ for the proof of Theorem 3.1.7. Because all $\texttt{ext}(c)$ in the hatched part verify $c \geq a$, t_B and t_C can be assumed to lie outside of it without loss of generality. The three cases analyzed in Lemmas 3.1.8, 3.1.9 and 3.1.10 depend on the position of ρ' relative to t_B and t_C.

Lemma 3.1.8 (Case 1)**.** *If $\rho' \notin [t_B, t_C]$, with high probability Algorithm 3.4 or Algorithm 3.5 rejects u.*

Proof. This is the most typical case, and also the one that corresponds the best to the very simplified Figure 3.4.

We will proceed by showing that $h_B^{\rightarrow}$ and $h_C^{\leftarrow}$ are unbalanced respectively when $u[t_C]$ and $u[t_B]$ are processed. From this it results that if $m_C \leq m_B$, w.h.p. Algorithm 3.4 detects $h_B^{\rightarrow} \neq 0$, and otherwise, $m_B < m_C$ and w.h.p. Algorithm 3.5 detects $h_C^{\leftarrow} \neq 0$. Note that for the second case, $m_B = m_C$ would not be enough as we need Algorithm 3.5 to evaluate $h_C^{\leftarrow}$ without modifying it.

We first prove that $h_B^{\rightarrow}$ is unbalanced when $u[t_C]$ is processed by Algorithm 3.4. Let us assume there exists $B_1 \subsetneq B$ such that $u[\rho'] = \texttt{ins}(a)$ is encoded in $h_{B_1}^{\rightarrow}$ when $u[t_B]$ is processed. Then, by definition of m_B, $m_{B_1} \geq m_B$. Therefore, Algorithm 3.4 checks $h_{B_1}^{\rightarrow} = 0$ at line 26 when processing $u[t_B]$. Moreover, $\rho \in C$, so $u[\rho] = \texttt{ext}(a)$ is not processed yet and not encoded in h_{B_1}. Therefore, Algorithm 3.4 rejects w.h.p.

We can now assume that there is no such $B_1 \subsetneq B$, and therefore that h_B does not encode $u[\rho'] = \texttt{ins}(a)$ when $u[t_C]$ is processed by Algorithm 3.4. But at this date $h_B^{\rightarrow}$ encodes $\texttt{ext}(a)$, as all extractions $\texttt{ext}(c)$ between τ and ρ satisfy $c \geq a$. Therefore $h_B^{\rightarrow}$ is unbalanced when t_C is processed.

The proof that $h_C^{\leftarrow}$ is unbalanced when $u[t_B]$ is processed by Algorithm 3.5 is similar. Let us assume there exists $C_1 \subsetneq C$ such that $u[\rho'] = \texttt{ins}(a)$ is encoded in $h_{C_1}^{\leftarrow}$ when $u[t_B]$ is processed. If $m_C < m_{C_1}$, or $m_C < a$, we can continue as above, as $h_{C_1}^{\leftarrow}$ would be unbalanced, which would be detected at line 25. We would then go on to assume that no such C_1 exist and conclude as above.

In the case that $m_C = m_{C_1} = a$, Algorithm 3.5 may add some $\texttt{ins}(a)$ to h_{C_1} before performing the check at line 25. Since $u[\rho']$ is encoded in h_{C_1} even though $a = m_{C_1}$, we know that $u[\rho' + 1, t_{C_1}]$ is $\texttt{ext}(a)$-unbalanced. By definition of ρ' and because $\rho' > t_C \geq \rho$, we know that for any $t < \rho$, $u[t, \rho]$ is $\texttt{ext}(a)$-unbalanced. In particular, it follows that $u[t_B, t_{C_1}]$ is $\texttt{ext}(a)$-unbalanced. Any other instanced of $\texttt{ins}(a)$ and $\texttt{ext}(a)$ from C to the right of t_{C_1} may only add to the unbalance as $m_C = a$ and the counter $\delta_D^{\leftarrow}$ makes it impossible for any block D to encode more instances of $\texttt{ins}(m_D)$ than of $\texttt{ext}(m_D)$. Therefore, $h_C^{\leftarrow}$ is unbalanced when t_B is processed. □

Lemma 3.1.9 (Case 2)**.** *If $t_B < \rho' < \tau$, with high probability Algorithm 3.5 rejects u.*

Proof. We show that when Algorithm 3.5 processes $u[t_B] = \texttt{ext}(m_B)$, it checks $h_D^{\leftarrow} = 0$ at line 25 for some $h_D^{\leftarrow}$ encoding $u[\rho']$ but not $u[\rho]$. By maximality of ρ', this factor of u cannot be a-balanced, thus Algorithm 3.5 will reject with high probability.

When $u[\rho'] = \texttt{ins}(a)$ is processed on the right-to-left pass, $\tau \in B_1$ with B_1 a block in the stack. $\tau \in B$, therefore $B_1 \subset B$. Because $u[\tau] = \texttt{ext}(b)$, we have $a > b \geq m_{B_1}$, and block B_1 is eligible at line 31 of Algorithm 3.5, meaning that $u[\rho'] = \texttt{ins}(a)$ is encoded in either $h^{\leftarrow}_{B_1}$ or a more recent hashcode $h^{\leftarrow}_{B_2}$. Since $\rho' \in B$, again $B_2 \subset B$. Last, when Algorithm 3.5 processes $u[t_B] = \texttt{ext}(m_B)$, since we are still within B, some hashcode h_{B_3}, with $B_3 \subset B$, encodes $u[\rho']$. Moreover, $h^{\leftarrow}_{B_3}$ does not encode $u[\rho] = \texttt{ext}(a)$ since $\rho \in C$ and C was processed before B, and is therefore $\texttt{ins}(a)$-unbalanced. Last, $m_{B_3} \geq m_B$, by definition of m_B. Hence, Algorithm 3.5 checks $h^{\leftarrow}_{B_3} = 0$ at line 25 when processing $u[t_B]$. Because $h^{\leftarrow}_{B_3}$ is $\texttt{ins}(a)$-unbalanced, any insertions Algorithm 3.5 may add to it will not make it balanced, and it rejects w.h.p. □

Lemma 3.1.10 (Case 3). *If $\rho < \rho' < t_C$, with high probability Algorithm 3.4 or Algorithm 3.5 rejects u.*

Proof. We show that unless u can also be made to match Case 1 and Lemma 3.1.8 by choosing another t_C, then when Algorithm 3.4 processes $u[t_C] = \texttt{ext}(m_C)$, it checks $h^{\rightarrow}_D = 0$ at line 26 for some $h^{\rightarrow}_D$ encoding $u[\rho']$ but not $u[\rho]$. By minimality of ρ', this factor of u cannot be a-balanced, thus algorithm 3.4 will rejects with high probability.

When Algorithm 3.4 processed $u[\rho] = \texttt{ext}(a)$, it encodes it in h_B as $m_B \leq b < a$ and for every $t < \rho$ in C such that $u[t] = \texttt{ext}(c)$, $c \geq a$. When this same algorithm processes $u[\rho'] = \texttt{ins}(a)$, either it encodes it in h_B, or in some h_{C_1} with $C_1 \subset C$. Let us first assume it encodes it in h_B, and let $C_2 \subset C$ be the block containing ρ as $u[\rho']$ is processed. Clearly $m_{C_2} = a$, otherwise C_2 would be eligible at line 19 of Algorithm 3.4, and $u[\rho']$ would not be encoded in h_B. Then we have $\delta^{\rightarrow}_{C_2} > 0$ by minimality of ρ', and Algorithm 3.4 will reject as soon as it encounters $\texttt{ext}(d)$ with $d < a$ if B and C have not merged. This implies that if the algorithm does not reject, then $m_C = a$. In particular, we could take $t_C = \rho$ and be in the conditions of Lemma 3.1.8.

We now assume without loss of generality that $u[\rho']$ is encoded in some h_{C_1}. Then $a > m_{C_1} \geq m_C$. When Algorithm 3.4 processes $u[t_C] = \texttt{ext}(m_C)$, since we are still within C, some hashcode h_{C_3} with $C_3 \subset C$ encodes $u[\rho']$, but not $u[\rho] = \texttt{ext}(a)$ which is still encoded in $h^{\rightarrow}_B$. Hence, Algorithm 3.4 checks $h^{\rightarrow}_{C_3} = 0$ at line 26 when processing $u[t_C]$, and rejects w.h.p. □

3.1.3 Multiple unidirectional passes

While the authors did not mention it in their paper [10] as their focus was on one-pass algorithms, the result of Theorem 3.1.2 can easily be extended with a slightly modified algorithm to multiple passes. It suffices to first divide $\{1, \dots, U\}$ in $\mathrm{O}(p)$ intervals such that all of them have roughly the same number of elements, and then run the algorithm only on that reduced interval, while checking that the number of insertions and extractions above each of the bounds of the interval is coherent. Choosing the intervals requires potentially $\mathrm{O}(\log n)$ passes for each interval, but those can be parallelized since the algorithm has memory presumably larger than p (if it does not, then p is very large and other more costly strategies still use a negligible number of passes), so we only require that $p \geq 2 \log n$.

Another improvement on the memory is to make the factors of u given as input to Algorithm 3.2 of size $\sqrt{n(\log n + \log U)}$ instead of $\sqrt{n}$, which gives us even less valleys in u after it has been pre-processed.

Finally, it is possible to keep the processing time per letter low even though the we preprocess the input with Algorithm 3.2 first. For this it suffices to decrease the size of the factors by $\mathrm{O}(\log n + \log U)$ at each step so that their total number stays the same and the last factor is of size $\mathrm{O}(\log n + \log U)$. Then Algorithm 3.1 can be consistently one factor behind Algorithm 3.1 and still catch up in the end.

Corollary 3.1.11 (Chakrabarti, Cormode, Kondapalli, McGregor). *For all $p \geq 2 \log n$ there is a unidirectional p-pass randomized streaming algorithm recognizing* $\mathrm{PQ}(U)$ *with memory space* $\mathrm{O}(\sqrt{n(\log U + \log n)}/p)$, *and one-sided bounded error n^{-c}, for inputs of length n and any constant $c > 0$. Furthermore, the processing time per letter is* $\mathrm{polylog}(nU)$.

3.2 Streaming Property Tester for Visibly Pushdown Languages

We design a streaming property tester for visibly pushdown languages. Our algorithm works by reducing the problem to a query-model tester for weighted regular languages. It therefore uses a lot of sampling. We first describe how we obtain a random sampling of a weighted word in streaming.

3.2.1 Sampling weighted words

We want a sampling on small factors of u according to their weights. We introduce a specific notion adapted to our setting. For a weighted word u, we denote by *k-factor sampling on u* the sampling over factors $u[i, i+l]$ with probability $|u[i]|/|u|$, where $l \geq 0$ is the smallest integer such that $|u[i, i+l]| \geq k$ if it exists, otherwise l is such that $i+l$ is the last letter of u. More generally we call k-factor such a factor. For the special case of $k = 1$, we call this sampling a *letter sampling on u*. Observe that both of them can be implemented using a standard reservoir sampling (see Algorithm 3.6 for letter sampling).

```
Input: Data stream $u$, Integer parameter $t > 1$
Data structure:
  $\sigma \leftarrow 0$ // Current weight of the processed stream
  $S \leftarrow$ empty multiset // Multiset of sampled letters
Code:
$i \leftarrow 1$, $a \leftarrow \mathrm{Next}(u)$, $\sigma \leftarrow |a|$
$S \leftarrow$ $t$ copies of $a$
While $u$ not finished
    $i++$, $a \leftarrow \mathrm{Next}(u)$, $\sigma \leftarrow \sigma + |a|$
    For each $b \in S$
        Replace $b$ by $a$ with probability $|a|/\sigma$
Output $S$
```

Algorithm 3.6: Reservoir Sampling

Even though our algorithm will require several samples from a k-factor sampling, we can simulate that with only a sampling of either larger factors, more factors, or both. Let $\mathcal{W}_1$ be a sampler producing a random multiset S_1 of factors of some given weighted word u. Then $\mathcal{W}_2$ *over samples* $\mathcal{W}_1$ if it produces a random multiset S_2 of factors of u such that $\Pr(\mathcal{W}_2 \text{ samples } S_2 \geq S_1) \geq \Pr(\mathcal{W}_1 \text{ samples } S_1)$, where each probability term refers to random choices of the corresponding sampler.

3.2.2 Property Tester for Regular Languages in the Query Model

Our main algorithm uses as a basic routine a non-adaptive query property tester for weighted regular languages. Property testing of regular languages was first considered in [1] for the Hamming distance. A slightly different approach with the edit distance was considered by Ndione et al. in [35]. Here we consider a slightly weaker version of that tester to simplify our proof. For the proof we consider the graph of components of the automaton and focus on paths in this graph; we also introduce the criterion, κ-saturation, for some parameter $0 < \kappa < 1$, that significantly simplifies the correctness proof of the tester compared to the original.

For the rest of this section, fix a regular language L recognized by some finite state automaton $\mathcal{A}$ on Σ with a set of states Q of size $m \geq 2$, and a diameter $d \geq 2$. Define the directed graph $G_{\mathcal{A}}$ on vertex set Q whose edges are pairs (p, q) when $p \xrightarrow{a} q$ for some $a \in \Sigma$.

Theorem 3.2.1 (Ndione, Lemay and Niehren). *Let $\mathcal{A}$ be an automaton recognizing a language L with $m \geq 2$ states and diameter $d \geq 2$. There is an algorithm that:*

1. *Takes as input $\varepsilon > 0$, $\eta > 0$ and t factors of $v_1, \ldots, v_t$ of some weighted word u, such that $t \geq 2\lceil 2dm^3(\log 1/\eta)/\varepsilon \rceil$;*
2. *Accepts if $u \in L$;*
3. *Rejects with probability at least $1 - \eta$ if $\mathsf{dist}(u, L) > \varepsilon|u|$, when each factor v_i comes from an independent k-factor sampling on u with $k \geq \lceil 2dm/\varepsilon \rceil$.*

This is still true if the algorithm is given an over-sampling of each of factors v_i instead.

A *component* C of $G_{\mathcal{A}}$ is a maximal subset (w.r.t. inclusion) of vertices of $G_{\mathcal{A}}$ such that for every p_1, p_2 in C one has a path in $G_{\mathcal{A}}$ from p_1 to p_2. The *graph of components* $\mathcal{G}_{\mathcal{A}}$ of $G_{\mathcal{A}}$ describes the transition relation of $\mathcal{A}$ on components of $G_{\mathcal{A}}$: its vertices are the components and there is a directed edge (C_1, C_2) if there is an edge of $G_{\mathcal{A}}$ from a vertex in C_1 toward a vertex in C_2.

Definition 3.2.2. *Let C be a component of $G_{\mathcal{A}}$, let $\Pi = (C_1, \ldots, C_l)$ be a path in $\mathcal{G}_{\mathcal{A}}$.*

- *A word u is C-*compatible *if there are states $p, q \in C$ such that $p \xrightarrow{u} q$.*
- *A word u is Π-*compatible *if u can be partitioned into $u = v_1 a_1 v_2 \ldots a_{l-1} v_l$ such that $p_i \xrightarrow{v_i} q_i$ and $q_i \xrightarrow{a_i} p_{i+1}$, where v_i is a factor, a_i a letter, and $p_i, q_i \in C_i$.*
- *A sequence of factors $(v_1, \ldots, v_t)$ of a word u is Π-*compatible *if they are factors of another Π-*compatible *word with the same relative order and same overlap.*

Note that the above properties are easy to check. Indeed, C-compatibility is a reachability property while the two others easily follow from C-compatibility checking.

We now give a criterion that characterizes those words u that are ε-far to every Π-compatible word. Note that it will not be used in the tester that we design for Theorem 3.2.1 for weighted regular languages, but only in Lemma 3.2.4 which is the key tool to prove its correctness.

For a component C and a C-incompatible word v, let $v_1 \cdot a$ be the shortest C-incompatible prefix of v. We define and denote the *C-cut* of v as $v = v_1 \cdot a \cdot v_2$. When v_1 is not the empty word, we say that v_1 is a *C-factor* and a is a *C-separator for* v_1, otherwise we say that a is a *strong C-separator*.

Fix a path $\Pi = (C_1, \ldots, C_l)$ in $\mathcal{G}_{\mathcal{A}}$, a parameter $0 < \kappa \leq 1$, and consider a weighted word u. We define a natural partition of u according to Π, that we call the *Π-partition* of u. For this, start with the first component $C = C_1$, and consider the C_1-cut $u_1 \cdot a \cdot u_2$ of u. Next, we inductively continue this process with either the suffix $a \cdot u_2$ if a is a C_1-separator, or the suffix u_2 if a is a strong C_1-separator. Based on some criterion defined below we will move from the current component C_i to a next component C_j of Π, where most often $j = i + 1$, until the full word u is processed. If we reach $j = l + 1$, we say that u *κ-saturates* Π and the process stops. We now explain how we move on in Π. We stay within C_i as long as both the number of C_i-factors and the total weight of strong C_i-separators are at most $\kappa|u|$ each. Then, we continue the decomposition with some fresh counting and using a new component C_j selected as follows. One sets $j = i + 1$ except when the transition is the consequence of a strong C_i-separator a of weight greater than $\kappa|u|$, that we call a *heavy strong separator*. In that case only, one lets $j \geq i + 1$, if exists, to be the minimal integer such that $q \xrightarrow{a} q'$ with $q \in C_{j-1} \cup C_j$ and $q' \in C_j$, and $j = l + 1$ otherwise.

Proposition 3.2.3. *Let $0 < \kappa \leq \varepsilon/(2dl)$. If u is ε-far to every Π-compatible word, then u κ-saturates Π.*

Proof. The proof is by contraposition. For this we assume that u does not κ-saturate Π and we correct u to a Π-compatible word as follows.

First, we delete each strong separator of weight less that $\kappa|u|$. Their total weight is at most $2l\kappa|u|$. Because u does not saturate, each strong separator of weight larger than $\kappa|u|$ fits in the Π-partition, and does not need to be deleted.

We now have a sequence of consecutive C_i-factors and of heavy strong C_i-separators, for some $1 \leq i \leq l$, in an order compatible with Π. However, the word is not yet compatible with Π since each factor may end with a state different than the first state of the next factor. However, for each such pair there is a path connecting them. We can therefore bridge all factors by inserting a factor of weight at most d, the diameter of $\mathcal{A}$. The resulting word is then Π-compatible by construction, and the total cost of the edit operations is at most $(2l + dl)\kappa|u| \leq \varepsilon|u|$, since $d \geq 2$. □

For a weighted word u, we recall that the k-factor sampling on u is defined in Section 3.2.1. The following lemma is the key lemma for the proof of Theorem 3.2.1.

Lemma 3.2.4. *Let u be a weighted word, let $\Pi = C_1 \dots C_l$ be a path in $\mathcal{G}_{\mathcal{A}}$. Let $0 < \kappa \leq \varepsilon/(2dl)$ and let $\mathcal{W}$ denote the $\lceil 2/\kappa \rceil$-factor sampling on u. Then for every $0 < \eta < 1$ and $t \geq 2l(\log 1/\eta)/\kappa$, the probability $P(u, \Pi) = \Pr_{(v_1,\dots,v_t)\sim\mathcal{W}^{\otimes t}}[(v_1, \dots, v_t)$ is Π-compatible$]$ satisfies $P(u, \Pi) = 1$ when u is Π-compatible, and $P(u, \Pi) \leq \eta$ when u is ε-far for from being Π-compatible.*

Proof. The first part of the theorem is immediate. For the second part, assume that u is ε-far from any Π-compatible word. For simplicity we assume that $2/\kappa$ and $\kappa|u|/2$ are integers. We first partition u according to Π and κ. Then, Proposition 3.2.3 tells us that u κ-saturates Π. For each C_i, we have three possible cases.

1. There are $\kappa|u|$ disjoint C_i-factors in u. Since they have total weight at most $|u|$, there are at least $\kappa|u|/2$ of them whose weight is at most $2/\kappa$ each. Since each letter has weight at least 1, the total weight of the first letters of each of those factors is at least $\kappa|u|/2$. Therefore one of them together with its C_i-separator is a sub-factor of some sampled factor v_j with probability at least $1 - (1 - \kappa/2)^t$.

2. The total weight of strong C_i-separators of u is at least $\kappa|u|$. Therefore one of them is the first letter of some sampled factor v_j with probability at least $1 - (1 - \kappa)^t$.

3. There is not any C_i-factor and any C_i-separator of u, because of a strong $C_{i'}$-separator of weight greater than $\kappa|u|$, for some $i' < i$. This separator is the first letter of some sampled factor v_j with probability at least $1 - (1 - \kappa)^t$.

By union bound, the probability that one of the above mentioned samples fails to occurs is at most $l(1 - \kappa)^t \leq \eta$. We assume now that they all occur, and we show that they form a Π-incompatible sequence. For each i, let w_i be the above described sub-factors of those samples. Each w_i appears in u after w_{i-1} or, in the case of a strong separator of heavy weight, $w_i = w_{i-1}$. Moreover each factor w_i which is distinct from w_{i-1} forces next factors to start from some component $C_{i'}$ with $i' > i$. As a result $(w_1, \dots, w_l)$ is not Π-compatible, and as a consequence $(v_1, \dots, v_t)$ neither, so the result. □

From Lemma 3.2.4 we can design a non-adaptive tester for L and, even more, also approximate the action of u on $\mathcal{A}$. The existence of the second algorithm, from Theorem 3.2.6, implies Theorem 3.2.1.

Definition 3.2.5. *Let $\Sigma' \subseteq \Sigma$ and $R \subseteq Q \times Q$. Then R* (ε, Σ')-approximates a word u on $\mathcal{A}$ (or simply ε-approximates when $\Sigma' = \Sigma$)*, if for all $p, q \in Q$: (1) $(p, q) \in R$ when $p \xrightarrow{u} q$; (2) u is (ε, Σ')-close to some word v satisfying $p \xrightarrow{v} q$ when $(p, q) \in R$.*

Theorem 3.2.6 (Inspired by Ndione, Lemay and Niehren)**.** *Let $\mathcal{A}$ be an automaton with $m \geq 2$ states and diameter $d \geq 2$. There is an algorithm that:*

1. *Takes as input $\varepsilon > 0$, $\eta > 0$ and t factors of $v_1, \ldots, v_t$ of some weighted word u, such that $t \geq 2\lceil 2dm^3(\log 1/\eta)/\varepsilon\rceil$;*
2. *Outputs a set $R \subseteq Q \times Q$ that ε-approximates u on $\mathcal{A}$ with one-sided error η, when each factor v_i comes from an independent k-factor sampling on u with $k \geq \lceil 2dm/\varepsilon\rceil$.*

This is still true with any combination of the following generalization:

- *The algorithm is given an over-sampling of each of factors v_i instead.*
- *When $\mathcal{A}$ is Σ'-closed, and d is the Σ'-diameter of $\mathcal{A}$, then R also (ε, Σ')-approximates u on $\mathcal{A}$.*

Proof. The algorithm is very simple:

```
1. Set $R = \emptyset$
2. For all states $p, q \in Q$
    (a) Check if factors $v_1, \ldots, v_t$ could come from a word $v$ such that $p \xrightarrow{v} q$
        // Step (a) is done using the graph $\mathcal{G}_{\mathcal{A}}$ of connected components of $\mathcal{A}$
    (b) If yes, then add $(p, q)$ to $R$
3. Return $R$
```

It is clear that this R contains every (p, q) such that $p \xrightarrow{u} q$. Now for the converse, we will show that, with bounded error η, the output set R only contains pairs (p, q) such that there exists a path $\Pi = C_1, \ldots, C_l$ on $\mathcal{G}_{\mathcal{A}}$ such that $p \in C_1$, $q \in C_l$, and u is Π-compatible. In that case, there is an ε-close word v satisfying $p \xrightarrow{v} q$.

Indeed, using $l \leq m$ and Lemma 3.2.4 with t, $\kappa = \varepsilon/(2dm)$ and $\eta' = \eta/2^m$, the samples satisfy $P(u, \Pi) \leq \eta/2^m$, when u is not Π-compatible. Therefore, we can conclude using a union bound argument on all possible paths on $\mathcal{G}_{\mathcal{A}}$, which have cardinality at most 2^m, that, with probability at least $1 - \eta$, there is no Π such that the samples are Π-compatible but u is not Π-compatible.

The structure of the tester is such that it has only more chances to reject a word that is not Π-compatible given an over-sampling as input instead. Words u such that $p \xrightarrow{u} q$ will always be accepted no matter the amount and length of samples. Therefore the theorem still holds with an over sampling.

Last, $\mathcal{A}$ being Σ'-closed ensures that the notions of compatibility and saturation remain unchanged. Using the Σ'-diameter in Lemma 3.2.4 (and therefore in Proposition 3.2.3) let us use bridges in Σ'^* instead of Σ^* with weight at most d. □

3.2.3 Our result

Theorem 3.2.7. *Let $\mathcal{A}$ be a* VPA *for L with $m \geq 2$ states, and let $\varepsilon, \eta > 0$. Then there is a streaming ε-tester for L and the edit distance with one-sided error η and memory space $\mathrm{O}(m^5 2^{3m^2}(\log^6 n)(\log 1/\eta)/\varepsilon^4)$, where n is the input length.*

The notions of *edit distance* and *balanced edit distance* are defined in Section 2.3.1. Note that for all balanced u and v, $\mathsf{dist}(u, v) \leq \mathsf{bdist}(u, v)$, so a property tester for the balanced edit distance is also a property tester for the edit distance. From now on, we will always use the balanced edit distance for words in a visibly push-down languages (which have pop and push symbols), and use the edit distance in a regular language (since the balanced edit distance is not properly defined).

3.2.3.1 A simple case : Non-Alternating Sequences

We first consider restricted instances consisting only of a *peak*, that is sequences of push symbols followed by a sequence of pop symbols, with possibly intermediate neutral symbols. These sequences are elements of the language $\Lambda = \bigcup_{j\geq 0}((\Sigma_=)^* \cdot \Sigma_+)^j \cdot (\Sigma_=)^* \cdot (\Sigma_- \cdot (\Sigma_=)^*)^j$.

Those instances are already hard for both streaming algorithms and property testing algorithms. Indeed, consider the language DISJOINTNESS $\subseteq \Lambda$ over alphabet $\Sigma = \{0, 1, \overline{0}, \overline{1}, a\}$ and defined by the union of all $a^* \cdot x(1) \cdot a^* \cdot \ldots \cdot x(j) \cdot a^* \cdot \overline{y(j)} \cdot a^* \cdot \ldots \cdot \overline{y(1)} \cdot a^*$, where $j \geq 1$, $x, y \in \{0, 1\}^j$, and $x(i)y(i) \neq 1$ for all i.

Then DISJOINTNESS can be recognized by a VPA with 3 states, $\Sigma_+ = \{0, 1\}$, $\Sigma_- = \{\overline{0}, \overline{1}\}$ and $\Sigma_= = \{a\}$. However, DISJOINTNESS is hard to recognize for both models. The hardness for streaming algorithms (without any notion of approximation) comes from a standard reduction to a communication complexity problem known as Set-Disjointness, and remains valid for *p-pass* streaming algorithms, that is streaming algorithms that are allowed to make up to p sequential passes (in any direction) on the input stream. The hardness for query model property testing algorithms comes from a similar result due to [36] for parenthesis languages with two types of parenthesis (note that, since we consider a non-alternating sequence, the language has to include a neutral symbol a for that result to hold), and for the Hamming distance. The result remains valid for both our language: we take the hard instance, and write 01 for (, 10 for [, $\overline{10}$ for) and $\overline{01}$ for]. The balanced edit distance is, in the case of non-alternating sequences, within a constant factor of the Hamming distance, so this does not affect the proof either.

Fact 3.2.8. *Any randomized p-pass streaming algorithm for* DISJOINTNESS *with bounded error $1/3$ requires memory space $\Omega(n/p)$, where n is the input length. Moreover, any (non-streaming) (2^{-6})-tester for* DISJOINTNESS *requires to query $\Omega(n^{1/11}/\log n)$ letters of the input word.*

Surprisingly, for every $\varepsilon > 0$, such languages (actually any language of the form $L \cap \Lambda$ where L is a VPL) become easy to ε-test by streaming algorithms. This is mainly because, given their full access to the input, streaming algorithms can perform an sample the input at a given height with only a single pass and few memory. This makes the property testing task significantly easier. We will first prove it in the case of a sequence of pushes and pops with no neutral symbol, and then generalize our proof to the case where neutral symbols exist.

These results are not only interesting as a simpler toy problem: our algorithm presented in Section 3.2.3.2 will work by reducing a general element of L to several elements of Λ. However the test will not be to determine whether these elements are in L but to which transitions of the automaton for L they could correspond, similarly to the difference between Theorem 3.2.1 and Theorem 3.2.6. Therefore we will not use the tester from Theorem 3.2.10 as a subroutine, but instead design another algorithm based on the exact same proof.

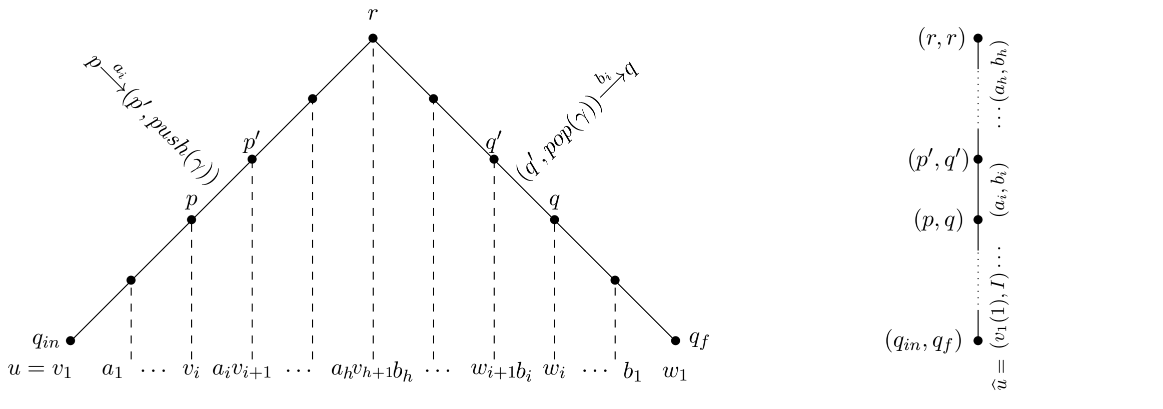
Theorem 3.2.9. *Let $\mathcal{A}$ be a* VPA *for L with $\Sigma_= = \emptyset$ and let $\varepsilon > 0$. There is a streaming ε-tester for $L \cap \Lambda$ with constant one-sided error with memory space $\mathrm{O}((c \log n)(\log 1/\varepsilon)/\varepsilon)$, where n is the input length and $c > 0$ depends only on $\mathcal{A}$.*

Theorem 3.2.10. *Let $\mathcal{A}$ be a* VPA *for L with $m \geq 2$ states, and let $\varepsilon, \eta > 0$. Then there is a streaming ε-tester for $L \cap \Lambda$ with one-sided error η and memory space $\mathrm{O}(m^8 (\log 1/\eta)/\varepsilon^2)$, where n is the input length.*

We will show that, for every VPL L, one can construct a regular language $\widehat{L}$ such that testing whether $u \in L \cap \Lambda$ is equivalent to test whether some other word $\widehat{u}$ belongs to $\widehat{L}$. This will let us use the query model property tester from Theorem 3.2.6. For this, let I be a special symbol not in $\Sigma_=$. Consider a word $u = \left(\prod_{i=1}^{j} v_i \cdot a_i\right) \cdot v_{j+1} \cdot \left(\prod_{i=j}^{1} \overline{b_i} \cdot w_i\right)$, where $a_i \in \Sigma_+$, $\overline{b_i} \in \Sigma_-$, and $v_i, w_i \in (\Sigma_=)^*$. Define the *slicing* of u (see Figure 3.7) as the word $\widehat{u}$ over the alphabet $\widehat{\Sigma} = (\Sigma_{+} \times \Sigma_{-}) \cup (\Sigma_{=} \times \{\mathrm{I}\}) \cup (\{\mathrm{I}\} \times \Sigma_{=})$ defined by $\widehat{u} = \Big(\prod_{i=1}^{j} (v_i(1), \mathrm{I}) \cdots (v_i(|v_i|), \mathrm{I}) \cdot (\mathrm{I}, w_i(1)) \ldots (\mathrm{I}, w_i(|w_i|)) \cdot (a_i, \overline{b_i})\Big) \cdot (v_{j+1}(1), \mathrm{I}) \cdots (v_{j+1}(|v_{j+1}|), \mathrm{I})$.

Definition 3.2.11. *Let $\mathcal{A} = (Q, \Sigma, \Gamma, Q_{in}, Q_f, \Delta)$ be a* VPA. *The* slicing *of $\mathcal{A}$ is the finite automaton $\widehat{\mathcal{A}} = (\widehat{Q}, \widehat{\Sigma}, \widehat{Q_{in}}, \widehat{Q_f}, \widehat{\Delta})$ where $\widehat{Q} = Q \times Q$, $\widehat{Q_{in}} = Q_{in} \times Q_f$, $\widehat{Q_f} = \{(p,p) : p \in Q\}$, and the transitions $\widehat{\Delta}$ are:*

1. *$(p,q) \xrightarrow{(a,b)} (p',q')$ when $p \xrightarrow{a} (p', \mathsf{push}(\gamma))$ and $(q', \mathsf{pop}(\gamma)) \xrightarrow{b} q$ are both transitions of Δ.*
2. *$(p,q) \xrightarrow{(c,\mathrm{I})} (p',q)$, resp. $(p,q) \xrightarrow{(\mathrm{I},c)} (p,q')$, when $p \xrightarrow{c} p'$, resp. $q \xrightarrow{c} q'$, is a transition of Δ.*


r
$p \xrightarrow{a_i} (p', push(\gamma))$
$(q', pop(\gamma)) \xrightarrow{b_i} q$
p'
q'
p
q
q_{in}
q_f
$u = v_1 \quad a_1 \cdots v_i \quad a_i v_{i+1} \cdots a_h v_{h+1} b_h \cdots w_{i+1} b_i \quad w_i \cdots b_1 \quad w_1$
(r,r)
(p',q')
(p,q)
(q_{in},q_f)
$\widehat{u} = (v_1(1), I) \cdots (a_i, b_i) \cdots (a_h, b_h)$

Run in the VPA $\mathcal{A}$ on u Run in the slicing automaton $\widehat{\mathcal{A}}$ on $\widehat{u}$

Figure 3.7: Slicing of a word $u \in \Lambda$ and evolution of the stack height for u.

Lemma 3.2.12. *If $\mathcal{A}$ is a* VPA *accepting L, then $\widehat{\mathcal{A}}$ is a finite automaton accepting $\widehat{L} = \{\widehat{u} : u \in L \cap \Lambda\}$.*

Proof. Because transitions on push symbols do not depend on the top of the stack, transitions in $\widehat{\Delta}$ correspond to slices that are valid for Δ (see Figure 3.7). Finally, $\widehat{Q_{in}}$ ensures that a run for L must start in Q_{in} and end in Q_f, and $\widehat{Q_f}$ that a state at the top of the peak is consistent from both sides. □

Observe also that, for $u, v \in \Lambda$, we have $\mathsf{bdist}(u,v) \leq 2\mathsf{dist}(\widehat{u}, \widehat{v})$. Those non-adaptive samples from the property tester of Theorem 3.2.6 of $\widehat{u}$ can be understood as a random sketch of u. To adapt this to a streaming algorithm for testing whether $u \in L \cap \Lambda$, we need to build an appropriate sampling procedure on u. In the case where u contains only pushes and pops (i.e. $\Sigma_{=} = \emptyset$), performing a reservoir sampling on the pushes and then taking the matching pops is enough. We can now prove Theorem 3.2.9.

Proof of Theorem 3.2.9. The tester of Theorem 3.2.6 samples uniformly at random several factors of the input word of several given lengths and it is still correct if it takes an over-sampling. Those samples on $\widehat{u}$ can be done in two steps. We describe it for a single factor of length k. Let u_{+} be the prefix of u before its first pop symbol, and let u_{-} be the remaining suffix including the first pop symbol. First we sample uniformly a random position in u_{+} and remember its position, which requires $\mathrm{O}(\log n)$ memory, and the following k letters in u_{+}. This sampling can be done without knowing the length of u_{+} in advance, using standard reservoir sampling techniques. Second, we complete the factor while reading u_{-}. That way, we simply have more letters than needed in the sampled factor. □

We could directly generalize the previous algorithm when $\Sigma_= \neq \emptyset$ by slightly modifying our sampling procedure. However, we prefer to take a different approach enlightening the main idea of our general algorithm in Section 3.2.3.2. Given any maximal factor $v \in (\Sigma_=)^*$ (for the sub-factor relation $\leq$) of the input stream, we will consider it as a single letter of weight $|v|$. More precisely, fix a VPA $\mathcal{A}$ recognizing L. Then, we compress v by its corresponding relation $R_v = \{(p,q) : p \xrightarrow{v} q\}$, and we see the subset $R_v \subseteq Q \times Q$ as a new letter, call it R, and the possible weights for R correspond to the weights of words v such that $R = R_v$. We augment $\Sigma_=$ by those new letters, and call this new (finite) alphabet Σ_0.

We also extend the automaton $\mathcal{A}$ and the language L with Σ_0. Doing so, we have compressed $u \in \Lambda$ to a weighted word of $\Lambda_1 = \bigcup_{j \geq 1} (\Sigma_0 \cdot \Sigma_+)^j \cdot \Sigma_0 \cdot (\Sigma_- \cdot \Sigma_0)^j$. Since there is a correspondence between letters $R \in \Sigma_0$ and words $v \in (\Sigma_=)^*$ with $|v| = |R|$ and $R = R_v$, we can arbitrarily reason on either the old or the new alphabet. Moreover, the corresponding slicing automaton $\widehat{\mathcal{A}}$ has still diameter at most $2m^2$.

Lemma 3.2.13. *Let $v \in \Lambda_1$ be s.t. $(p,q) \xrightarrow{\widehat{v}} (p',q')$. There is $w \in \Lambda_1$ s.t. $|w| \leq 2m^2$ and $(p,q) \xrightarrow{\widehat{w}} (p',q')$.*

Proof. This is simply due to the fact that the diameter of $\widehat{\mathcal{A}}$ is at most m^2 its number of states, given that all transitions have weight 1. Therefore $|\widehat{w}| \leq m^2$ and $|w| \leq 2m^2$. □

Let us now build a tester for $L \cap \Lambda$ using the same idea as in Theorem 3.2.9: to test a word u we use a tester for $\widehat{u}$ against $\widehat{L}$, which is now a language of weighted words. More precisely, the weight of a letter in $\widehat{u}$ is defined by $|(a_+, a_-)| = 1$ and $|(\mathrm{I}, R)| = |(R, \mathrm{I})| = |R|$. Theorem 3.2.6 gives us such a tester. The remaining difficulty is to provide to this tester an appropriate sampling on $\widehat{u}$ while processing u.

Our tester for weighted regular languages is based on k-factor sampling on $\widehat{u}$ that we will simulate by an over-sampling built from a letter sampling on u, that is according to the weights of the letters of u only. This new sampling can be easily performed given a stream of u using a standard reservoir sampling.

Definition 3.2.14. *For a weighted word $u \in \Lambda$, denote by $\mathcal{W}_k(u)$ the sampling over factors of $\widehat{u}$ constructed as follows: (1) sample a letter $u(i)$ of u with probability $|u(i)|/|u|$; (2) if $u(i)$ is in a push sequence, extends it to the factor $u[i, i+l+1]$ where $u[i, i+l]$ is a k-factor, and complete it with its matching pop sequence.*

Lemma 3.2.15. *Let u be a weighted word, and let k be such that $4k \leq |u|$. Then $4k$ independent copies of $\mathcal{W}_k(u)$ over samples the k-factor sampling on $\widehat{u}$.*

Proof. Denote by $\widehat{\mathcal{W}}$ the k-factor sampling on $\widehat{u}$, and by $\mathcal{W}$ some $4k$ independent copies of $\mathcal{W}_k(u)$. For any k-factor v of $\widehat{u}$, we will show that the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$. For that, we distinguish the following three cases:

- v is a single letter. Then, if $v = (R, I)$ the probability that it is sampled by $\widehat{\mathcal{W}}$ equals the probability that $\mathcal{W}_k(u)$ samples the factor v augmented by one letter; if $v = (I, R)$ the probability that it is sampled by $\mathcal{W}$ again equals the probability that $\mathcal{W}_k(u)$ samples it. Hence, the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.

- v is not a single letter and starts by a letter in $\Sigma_+ \times \Sigma_-$ or by a letter in $\Sigma_0 \times \{I\}$. Then the probability that it is sampled by $\widehat{\mathcal{W}}$ equals at most twice the probability that $\mathcal{W}_k(u)$ samples the factor v augmented by one letter, as a (push,pop) pair in $\widehat{u}$ has weight 2 when a push has weight 1 in u. Hence, the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.

- v is not a single letter and starts by a letter in $\Sigma_0 \times \{I\}$. Since $|\widehat{u}| \geq |u|/2$, we get

$$\Pr(\mathcal{W}_k(u) \text{ samples the factor } (a,b)\cdot v) = 1/|u| \quad \text{and} \quad \Pr(\widehat{\mathcal{W}} \text{ samples } v) \leq k/|\widehat{u}| \leq 2k/|u|.$$

Thus the probability that one of the $4k$ samples of $\mathcal{W}$ has the factor $(a,b)\cdot v$ is $1-(1-1/|u|)^{4k}$. As $1-(1-1/|u|)^{4k} \geq 1 - \frac{1}{1+4k/|u|} = \frac{4k}{|u|+4k} \geq \frac{2}{k}$ when $|u| \geq 4k$, we conclude again that the probability that v is sampled by $\widehat{\mathcal{W}}$ is at most the probability that v is a factor of an element sampled by $\mathcal{W}$.

□

We know have everything we need to prove Theorem 3.2.10.

Proof of Theorem 3.2.10. Observe that $\mathsf{bdist}(u,v) \leq 2\mathsf{dist}(\widehat{u},\widehat{v})$, and moreover the slicing automaton has diameter d at most $2m^2$ by Lemma 3.2.13. Given a word u as a data stream, we simulate a data stream on its compression u_1, which is a weighted word in Λ_1, and then obtain with Lemma 3.2.15 an over-sampling of t k-factor samplings on $\widehat{u_1}$, with $t = 4\lceil 4dm^3(\log 1/\eta)/\varepsilon\rceil$ and $k = \lceil 4dm/\varepsilon\rceil$. We then use the tester from Theorem 3.2.6 on the samples. Note that this is possible because this tester has non-adaptive queries. □

3.2.3.2 Exact Algorithm

We first describe an exact algorithm for recognizing VPLs with the same structure as our final algorithm, but require linear memory. We will then modify it to reduce the memory used, which will create the ε gap.

Fix a VPA $\mathcal{A}$ recognizing some VPL L. A general balanced input instance u will have more than one peak $v \in \Lambda$ and we cannot easily interpret u as an element of a regular language. However, we will recursively replace each factor $v \in \Lambda$ by $R_v = \{(p,q) : p \xrightarrow{v} q\}$ with weight $|v|$. The alphabet $\Sigma_=$ of neutral symbols will increase as follows. We start with Σ_0 encoding all possible relations R_v for $v \in \Sigma_=^*$. Then Λ_{h+1} is simply Λ over an alphabet $\Sigma_= = \Sigma_h$, and Σ_h encodes all possible relations R_v for words $v \in (\Lambda_h)^*$. As before, we naturally augment the automaton $\mathcal{A}$ and the language L with these new sets. However we keep the notation Σ as $\Sigma_+ \cup \Sigma_- \cup \Sigma_=$.

Since there is a finite set of possible relations, this construction has smallest fixed points Σ_∞ and Λ_∞. Denote by $\mathrm{Prefix}(\Lambda_\infty)$ the language of prefixes of words in Λ_∞. For $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$, the $\widehat{\Sigma'}$-diameter of the slicing automaton $\widehat{A}$ is simply the Σ-diameter of $\mathcal{A}$, that we bound as follows. For simpler languages, as those coming from DTD, this bound can be lowered to m.

Fact 3.2.16. *Let $\mathcal{A}$ be a VPA with m states. Then the Σ-diameter of $\mathcal{A}$ is at most 2^{m^2}.*

Proof. A similar statement is well known for any context-free grammar given in Chomsky normal form. Let N be the number of non-terminal symbols used in the grammar. If the grammar produces one balanced word from some non-terminal symbol, then it can also produce one whose length is at most 2^N from the same non-terminal symbol. This is proved using a pumping argument on the derivation tree. We refer the reader to the textbook [25].

Now, in the setting of visibly pushdown languages one needs to transform $\mathcal{A}$ into a context-free grammar in Chomsky normal form. For that, consider first an intermediate grammar whose non-terminal symbols are all the X_{pq} where p and q are states from $\mathcal{A}$: such a non-terminal symbol will produce exactly those words u such that $p \xrightarrow{u} q$, hence our initial symbol will be those of the form $X_{q_0 q_f}$ where q_0 is an initial state and q_f is a final state.

The rewriting rules are as follows:

- $X_{pp} \to \varepsilon$
- $X_{pq} \to X_{pr}X_{rq}$
- $X_{pq} \to aX_{p'q'}b$ whenever one has in the automaton $p \xrightarrow{a} (p', \mathsf{push}(\gamma))$ and $(q', \mathsf{pop}(\gamma)) \xrightarrow{a} q$ for some push symbol a, pop symbol b and stack letter γ.
- $X_{pq} \to aX_{p'q}$ whenever one has in the automaton $p \xrightarrow{a} p'$ for some neutral symbol a.
- $X_{pq} \to X_{pq'}a$ whenever one has in the automaton $q' \xrightarrow{a} q$ for some neutral symbol a.

Obviously, this grammar generates language $L(\mathcal{A})$.

As we are here interested only in the length of the balanced words produced by the grammar, we can replace any terminal symbol by a dummy symbol $\sharp$. Now, once this is done we can put the grammar in Chomsky normal form by using an extra non-terminal symbol (call it $X_\sharp$ as it is used to produce the $\sharp$ terminal). As we have $m^2 + 1$ non-terminal in the resulting grammar we are almost done. To get to the tight bound announced in the statement, one simply removes the extra non-terminal symbol $X_\sharp$ and reasons on the length of the derivation directly. □

We start by a simple algorithm maintaining a stack of small height, but whose elements can be of linear size. We will later explain how to replace the stack elements by appropriated small sketches. While having processed the prefix $u[1, i]$ of the data stream u, Algorithm 3.7 maintains a suffix $u_0 \in \mathrm{Prefix}(\Lambda_\infty)$ of $u[1, i]$, that is an unfinished peak, with some simplifications of factors v in Λ_∞ by their corresponding relation R_v. Therefore u_0 consists of a sequence of push symbols and neutral symbols possibly followed by a sequence of pop symbols and neutral symbols. The algorithm also maintains a subset $R_{\text{temp}} \subseteq Q \times Q$ that is the set of transitions for the maximal prefix of $u[1, i]$ in Λ_∞. When the stream is over, the set R_{temp} is used to decide whether $u \in L$ or not.

We now need define the $\bullet$ operation used by the algorithm, to concatenate while merging adjacent neutral symbols, and the depth of a factor for the analysis.

Definition 3.2.17. *Let u be a weighted word, and let a, b be weighted letters such that b is neutral. Then $(ua) \bullet b$ is defined as uab when a is not neutral, and otherwise as $u \cdot R_{ab}$, where R_{ab} denotes the set of ab-transitions.*

Definition 3.2.18. *For each factor constructed in Algorithm 3.7,* Depth *is defined dynamically by* $\mathrm{Depth}(a) = 0$ *when* $a \in \Sigma$, $\mathrm{Depth}(v) = \max_i \mathrm{Depth}(v(i))$ *and* $\mathrm{Depth}(R_v) = \mathrm{Depth}(v) + 1$.

When a push symbol a comes after the pop sequence, u_0 is no longer in $\mathrm{Prefix}(\Lambda_\infty)$, and Algorithm 3.7 puts it on a stack of unfinished peaks (see lines 10 to 11 and the upper part of Figure 3.8) and u_0 is reset to a. In other situations, one adds a to u_0. In case u_0 becomes a word of Λ_∞ (see lines 13 to 16 and the middle part of Figure 3.8), Algorithm 3.7 computes the set of u_0-transitions $R_{u_0} \in \Sigma_\infty$, and adds R_{u_0} to the previous unfinished peak, which is found on top of the stack and now becomes the current unfinished peak; in the special case where the stack is empty one simply updates the set R_{temp} by taking its composition with R_{u_0}.

In order to bound the size of the stack, Algorithm 3.7 considers the maximal well-balanced suffix v_2 of the topmost element $v_1 \cdot v_2$ of the stack and, when $|u_0| \geq |v_2|/2$, it computes the relation R_{v_2} and continues with a bigger current peak starting with v_1 (see lines 17 to 19 and the lower par of Figure 3.8). A consequence of this compression is that the elements in the stack have geometrically decreasing weight and therefore the height of the stack used by Algorithm 3.7 is logarithmic in the length of the input stream.

```
Input: Well-balanced data stream $u$
Data structure:
  $Stack \leftarrow$ empty stack // Stack of items $v$ with $v \in \text{Prefix}(\Lambda_\infty)$
  $u_0 \leftarrow \emptyset$ // $u_0 \in \text{Prefix}(\Lambda_\infty)$ is a suffix of the processed part $u[1,i]$ of $u$
        // with possibly some factors $v \in \Lambda_\infty$ replaced by $R_v$
  $R_{\text{temp}} \leftarrow \{(p,p)\}_{p \in Q}$ // Set of transitions for the maximal prefix of $u[1,i]$ in $\Lambda_\infty$
Code:
While $u$ not finished
  $a \leftarrow \text{Next}(u)$ //Read and process a new symbol $a$
  If $a \in \Sigma_+$ and $u_0$ has a letter in $\Sigma_-$ // $u_0 \cdot a \notin \text{Prefix}(\Lambda_k)$
    Push $u_0$ on $Stack$, $u_0 \leftarrow a$
  Else $u_0 \leftarrow u_0 \bullet a$
  If $u_0$ is well-balanced // $u_0 \in \Lambda_\infty$: compression
    Compute $R_{u_0}$ the set of $u_0$-transitions
    If $Stack = \emptyset$, then $R_{\text{temp}} \leftarrow R_{\text{temp}} \bullet R_{u_0}$, $u_0 \leftarrow \emptyset$
    Else Pop $v$ from $Stack$, $u_0 \leftarrow v \bullet R_{u_0}$
  Let $(v_1 \cdot v_2) \leftarrow \text{top}(Stack)$ s.t. $v_2$ is maximal and well-balanced // $v_2 \in \Lambda_\infty$
  If $|u_0| \geq |v_2|/2$  // $u_0$ is big enough and $v_2$ can be replaced by $R_{v_2}$
    Compute $R_{v_2}$ the set of $v_2$-transitions, Pop $v$ from Stack, $u_0 \leftarrow (v_1 \cdot R_{v_2}) \cdot u_0$
If $(Q_{in} \times Q_f) \cap R_{\text{temp}} \neq \emptyset$, Accept; Else Reject // $u = u_0$ and $R_{\text{temp}} = R_u$
```

Algorithm 3.7: Exact Tester for a VPL

The following proposition comes from a direct inspection of Algorithm 3.7.

Proposition 3.2.19. *Algorithm 3.7 accepts exactly words $u \in L$, while maintaining a stack of at most* $\log n$ *items of types v with $v \in \text{Prefix}(\Lambda_{\text{Depth}(v)})$, and a variable u_0 with $u_0 \in \text{Prefix}(\Lambda_{\text{Depth}(u_0)})$.*

We state that Algorithm 3.7 considers at most $\mathrm{O}(\log n)$ nested picks, that is $\text{Depth}(u) = \mathrm{O}(\log n)$, where Depth is dynamically defined in each letter and factor inside Algorithm 3.7.

Lemma 3.2.20. *Let v be the factor used to compute R_v at line either 14 or 19 of Algorithm 3.7. Then $|v(i)| \leq 2|v|/3$, for all i. In particular, it holds that* $\text{Depth}(u) = \mathrm{O}(\log n)$.

Proof. One only has to consider letters in Σ_∞. Hence, let R_w belongs to v for some w: either w was simplified into R_w at line 14 or at line 19 of Algorithm 3.7.

Let us first assume that it was done at line 19. Therefore, there is some $v' \in \text{Prefix}(\Lambda_\infty)$ to the right of w with total weight greater than $|w|/2 = |R_w|/2$. This factor v' is entirely contained within v: indeed, when R_w is computed v includes v'. Therefore $|R_w| \leq 2|v|/3$.

If R_w comes from line 14, then $w = u_0$ and this u_0 is well-balanced and compressed. We claim that at the previous round the test in line 18 failed, that is $|u_0| - 1 \leq |v_2|/2$ where v_2 is the maximal well-balanced suffix of $\text{top}(Stack)$. Indeed, when performing the sequence of actions following a positive test in line 18, the number of unmatched push symbols in the new u_0 is augmented at least by 1 from the previous u_0: hence, it cannot be equal to 1 as the elements in the stack have pending call symbols and therefore in the next round u_0 cannot be well-balanced. Therefore one has $|u_0| - 1 \leq |v_2|/2$. Now when $R_w = R_{u_0}$ is created, it is contains in a factor that also contains v_2 and at least one pending call before v_2. Hence, $|R_w| \leq 2|v|/3$.

Finally, the fact that $\text{Depth}(u) = \mathrm{O}(\log n)$ is a direct consequence of the definition of Depth and of the fact that the weight decreases at least geometrically with nesting. □

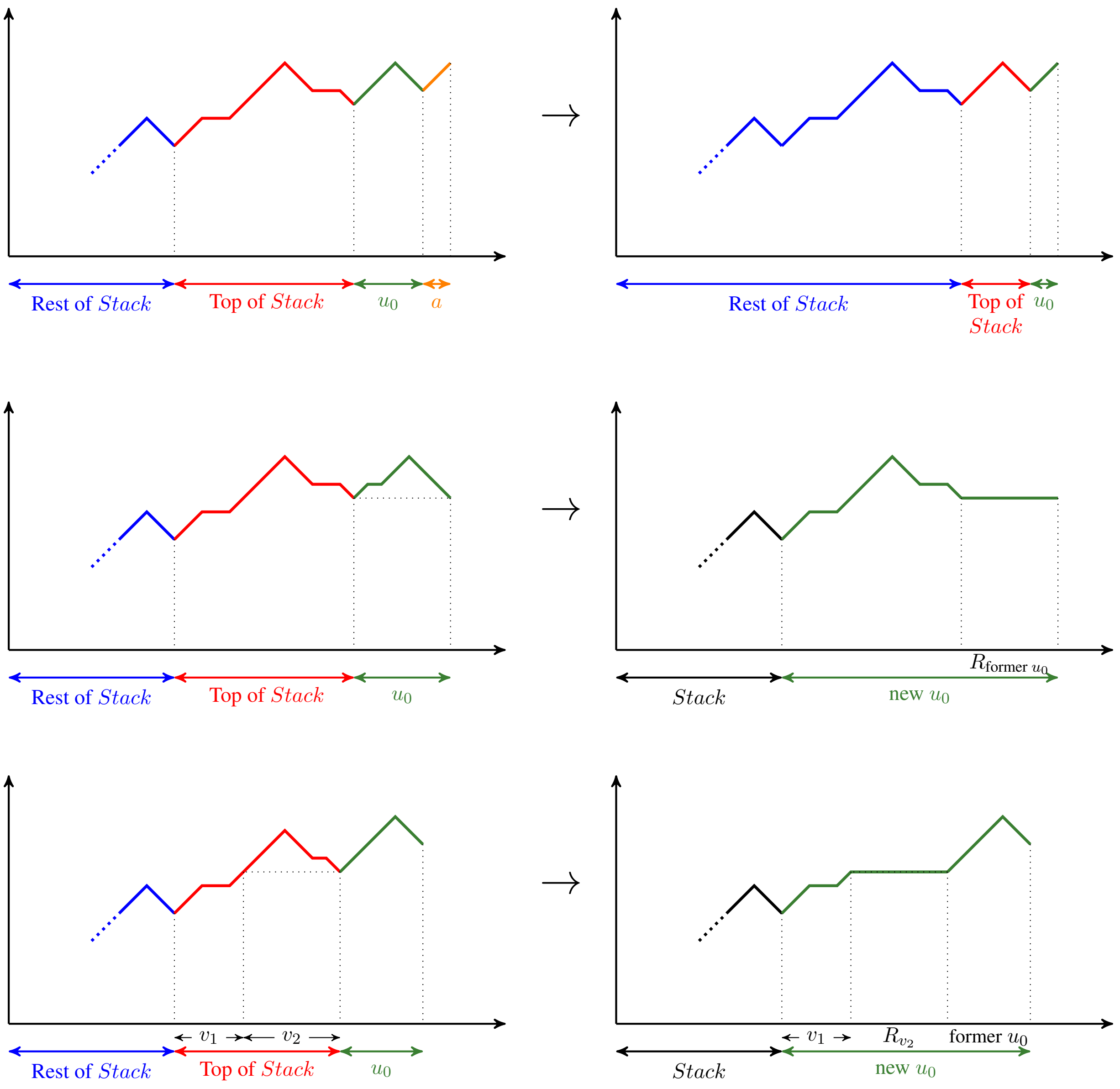


Figure 3.8: Illustration of Algorithm 3.7. The upper part corresponds to lines 10 to 11, the middle part to lines 13 to 16, and the lower part to lines 17 to 19

3.2.3.3 Sketching using Suffix Sampling

We now describe the sketches our algorithm uses. They are based on a notion of suffix samplings, which ensures a good letter sampling on each suffix of some data stream. Recall that the letter sampling on a weighted word u samples a random letter $u(i)$ (with its position) with probability $|u(i)|/|u|$.

Definition 3.2.21. *Let u be a weighted word and let $\alpha > 1$. An α-*suffix decomposition of u of size s *is a sequence of suffixes $(u^l)_{1 \le l \le s}$ of u such that: $u^1 = u$, u^s is the last letter of u, and for all l, u^{l+1} is a strict suffix of u^l and if $|u^l| > \alpha|u^{l+1}|$ then $u^l = a \cdot u^{l+1}$ where a is a single letter.*

An (α, t)-suffix sampling *on* u *of size* s *is an* α*-suffix decomposition of* u *of size* s *with* t *letter samplings on each suffix of the decomposition.*

An (α, t)-suffix sampling can be either concatenated to another one, or compressed as stated below.

Proposition 3.2.22. *Given as input an* (α, t)*-suffix sampling* D_u *on* u *of size* s_u *and another one* D_v *on* v *of size* s_v*, there is an algorithm* ***Concatenate***(D_u, D_v) *computing an* (α, t)*-suffix sampling on the concatenated word* $u \cdot v$ *of size at most* $s_u + s_v$ *in time* $\mathrm{O}(s_u)$.
Moreover, given as input an (α, t)*-suffix sampling* D_u *on* u *of size* s_u*, there is also an algorithm* ***Simplify***(D_u) *computing an* (α, t)*-suffix sampling on* u *of size at most* $2\lceil \log |u| / \log \alpha \rceil$ *in time* $\mathrm{O}(s_u)$.

We describe those procedures in Algorithm 3.8. A direct inspection suffices to prove that it satisfies Proposition 3.2.22.

```
Data structure:
  // $D$, $D_u$, $D_v$, $D_{\text{temp}}$ stacks of items $(\sigma, b)$, one for each suffix
  // of the decomposition where $\sigma$ encodes the weight and $b$ the $t$ samples
Code:
Concatenate($D_u, D_v$)
  $D \leftarrow D_u$
  $(c_1, \ldots, c_t) \leftarrow$ all $t$ samples on $v$ (the largest suffix in $D_v$)
  For each $(\sigma, b) \in S$ where $b = (b_1, \ldots, b_t)$
    Replace each $b_i$ by $c_i$ with probability $|v|/(|v| + \sigma)$
    Replace $(\sigma, b)$ by $(\sigma + |v|, b)$
  Append $D_v$ to the top of $D$
  Return $D$
Simplify($D_u$)
  $D \leftarrow D_u$
  For each $(\sigma, b) \in D$ from top to bottom
    $D_{\text{temp}} \leftarrow$ elements $(\tau, c) \in D$ below $(\sigma, b)$ with $\tau \leq \alpha\sigma$
    Replace $D_{\text{temp}}$ in $D$ by the bottom most element of $D_{\text{temp}}$
  Return $D$
Online-Suffix-Sampling
  $D \leftarrow \emptyset$
  While $u$ not finished
    $a \leftarrow \mathrm{Next}(u)$
    Concatenate($D, a$) where $a$ encodes the suffix sampling $(|a|, (a, \ldots, a))$
    Simplify($D$)
  Return $D$
```

Algorithm 3.8: α-Suffix Sampling

Using this Proposition 3.2.22, one can easily design a streaming algorithm constructing online a suffix decomposition of small size. Starting with an empty suffix-sampling S, simply concatenate S with the next processed letter a of the stream, and then simplify it. We formalize this together with functions **Concatenate** and **Simplify** in Algorithm 3.8.

3.2.3.4 Algorithm with sketches

We first describe a data structure that can be used to encode each unfinished peak v of the stack and u_0. Then, we explain how the operations of Algorithm 3.7 can be performed using our data structure. As a result our final algorithm is simply Algorithm 3.7 with the new data structure described in Algorithm 3.9 and the adapted operations defined in Algorithm 3.10. We will refer to the whole algorithm as Algorithm 3.7+3.10.

```
Parameters: real $\varepsilon' > 0$, integer $T \geq 1$
Data structure for a weighted word $v \in \mathrm{Prefix}(\Lambda_\infty^{\varepsilon'})$
  Weights of $v$ and of its first letter $v(1)$
  Heights of $v(1)$
  Boolean indicating whether $v$ contains a pop symbol
  $(1+\varepsilon')$-suffix decomposition $v^1,\dots,v^s$ of $v$ encoded by
     Estimates $|v^l|_{\mathtt{low}}$ and $|v^l|_{\mathtt{high}}$ of $|v^l|$
     $T$ independent samplings $S_{v^l}$ on $v^l$ // see details below
        with corresponding weights and heights
```
Algorithm 3.9: Sketch for an unfinished peak

We now detail the methods, where we implicitly assume that each letter processed by the algorithm comes with its respective height and (exact or approximate) weight. They use functions **Concatenate** and **Simplify** form Algorithm 3.8, while adapting them.

```
Adaption of functions from Proposition 3.2.22
 Concatenate($D_u, D_v$) with an exact estimate of $|v|$ is modified s.t.
   the replacement probability is now $|v|/(\sigma_{\mathtt{high}} + |v|)$
   and $|u^l \cdot v|_z \leftarrow |u^l|_z + |v|$, for $z = \mathtt{low}, \mathtt{high}$
 Simplify($D_u$) with $\alpha = 1+\varepsilon'$ has now the relaxed condition $\tau_{\mathtt{high}} \leq (1+\varepsilon')\sigma_{\mathtt{low}}$
Adaption of operations on factors used in Algorithm 3.7
 Bullet-concatenation with a neutral letter: $v \leftarrow u \bullet a$
   $b \leftarrow$ last letter of $u$
   $D_v \leftarrow$ Concatenate($D_u, a$)
   If $b$ is neutral
     $R_{ba} \leftarrow$ set of $ba$-transitions
     Delete suffix $b$ from $D_v$
     Replace every samples consisting of either $a$ or $b$ by $R_{ab}$
   $D_v \leftarrow$ Simplify($D_v$)
 Compute relation: $R_v$
   Run the algorithm of Corollary 3.2.28 using samples in $D_v$
 Decomposition: $v_1 \cdot v_2 \leftarrow v$
   Find largest suffix $v^i$ in $D_v$ s.t. $v^i \in \mathrm{Prefix}(\Lambda_\infty)$ // i.e. s.t. $v^i$ is in $v_2$
   $D_{v|v_1} \leftarrow$ suffixes $(v^l)_{l<i}$ with their samples
   $D_{v_2} \leftarrow$ suffix $v^i$ with its samples and weight estimates: // for computing $R_{v_2}$
     - $(|v^i|_{\mathtt{high}}, |v^i|_{\mathtt{low}})$ when $v^{i-1}$ and $v^i$ differ by exactly one letter (then $v^i = v_2$)
     - $(|v^{i-1}|_{\mathtt{high}}, |v^i|_{\mathtt{low}})$ otherwise
 Test: $|u_0| \geq |v_2|/2$ using $|v_2|_{\mathtt{low}}$ instead of $|v_2|$
 Concatenation: $u_0 \leftarrow (v_1 \cdot R_{v_2}) \cdot u_0$
   $D_{v'} \leftarrow (D_{v|v_1}, R_{v_2})$ replacing each samples of $D_{v|v_1}$ in $v_2$ by $R_{v_2}$
   \\ The height of a sample determines whether it is in $v_2$
   $D_{u_0} \leftarrow$ Simplify(Concatenate($D_{v'}, D_{u_0}$))
```
Algorithm 3.10: Adaptation of Algorithm 3.7 using sketches

In Section 3.2.3.5, we show that the samplings S_{v^l} are close enough to an $(1+\varepsilon')$-suffix sampling on v^l. This lets us build an over sampling of an $(1+\varepsilon')$-suffix sampling. We also show that it only require a polylogarithmic number of samples. Then, we explain how to recursively apply an adaptation of Theorem 3.2.6 (with ε') in order to obtain the compressions at line 14 and 19 while keeping a cumulative error below ε. We now state our main result whose proof uses results from the following section.

Proof of Theorem 3.2.7. We use Algorithm 3.7+3.10, which the tester from Corollary 3.2.28 for the compressions at lines 14 and 19 of Algorithm 3.7. We know from Lemma 3.2.29 and Lemma 3.2.15 that it is enough to choose $\varepsilon' = \varepsilon/(6\log n)$, $\eta' = \eta/n$, and Fact 3.2.16 gives us $d = 2^{m^2}$. Therefore we need $T = 2304m^4 2^{2m^2}(\log^2 n)(\log 1/\eta)/\varepsilon^2$ independent k-factor samplings of u augmented by one, with $k = 24m2^{m^2}(\log n)/\varepsilon$. Lemma 3.2.26 tells us that using twice more samples from our algorithm, that is for each S_{v^l}, is enough in order to over-sample them.

Because of the sampling variant we use, the size of each decomposition is at most $96(\log^2 n)/\varepsilon + \mathrm{O}(\log n)$ by Lemma 3.2.26. The samplings in each element of the decomposition use memory space k, and there are $2T$ of them. Furthermore, each element of the stack has its own sketch, and the stack is of height at most $\log n$. Multiplying all those together gives us the upper bound on the memory space used by Algorithm 3.7+3.10. □

3.2.3.5 Final analysis

As our final algorithm may fail at various steps, the relations it considers may not correspond to any word. But still, it will produces relations R such that for any $(p,q) \in R$, there is a balanced word $u \in \Sigma^*$, such that $p \xrightarrow{u} q$. We therefore consider the alphabet extension by any such relations R with any weight. We define Σ_Q to be the alphabet $\Sigma_=$ augmented by all such relations R, and we again extend the automaton and the language. Then, Λ_Q is simply Λ_1 with $\Sigma_= = \Sigma_Q$.

Proposition 3.2.23. *Each relation R that Algorithm 3.7+3.10 produces is in Σ_Q.*

Still the resulting automaton is $\widehat{\Sigma}'$-closed with $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$, and we remind that Fact 3.2.16 bounds the $\widehat{\Sigma}'$-diameter of $\widehat{\mathcal{A}}$ by 2^{m^2}.

Proposition 3.2.24. *The slicing automaton $\widehat{\mathcal{A}}$ that we define over $(\Sigma_+ \cup \widehat{\Sigma_- \cup \Sigma_Q}))$ is $\widehat{\Sigma}'$-closed with $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$.*

Stability. We want to show that the decomposition, weights and sampling we maintain are close enough to an $(1+\varepsilon')$-suffix sampling with correct weights. Recall that $\varepsilon' = \varepsilon/(6\log n)$.

Proposition 3.2.25. *Let v be an unfinished peak, and let $v^1, \ldots, v^s$ be the suffix decomposition maintained by the algorithm. The following is true:*
(1) $v^1, \ldots, v^s$ is a valid $(1+\varepsilon')$-suffix decomposition of v.
(2) For each letter a of every v^l, and for every sample s, $\Pr[S_{v^l} = a] \geq |a|/|v^l|_{\mathtt{high}}$.
(3) Each v^l satisfies $|v^l|_{\mathtt{high}} - |v^l|_{\mathtt{low}} \leq 2\varepsilon'|v^l|_{\mathtt{low}}/3$.

Proof. Property (1) is guaranteed by the (modified) **Simplify** function used in Algorithm 3.10, which preserves even more suffixes than the original algorithm.

Properties (2) and (3) are proven by induction on the last letter read by Algorithm 3.7+3.10. Both are true when no symbol has been read yet.

We start with property (2). Let us first consider the case where we use bullet-concatenation after the last letter was read. Then for all v^l, the (modified) **Concatenate** function ensures S_{v^l} becomes a with probability $1/|v^l|_{\mathtt{high}}$. Otherwise, S_{v^l} remains unchanged and by induction $S_{v^l} = b$ with probability at least $(1 - 1/|v^l|_{\mathtt{high}})|b|/(|v^l|_{\mathtt{high}} - 1) = |b|/|v^l|_{\mathtt{high}}$, for each other letter b of v^l. If a is a neutral symbol and u_0 ends with some $R \in \Sigma_Q$, any sample that would be either R or a is replaced by $R \bullet a$.

The other case is that some R_{v_2} is computed at line 19 of Algorithm 3.7. In this case, v is equal to some $(v_1 \cdot R_{v_2}) \cdot u_0$ concatenation. For each suffix $(v_1 \cdot v_2)^l$ in $D_{(v_1 \cdot v_2)}$ containing R_{v_2}, we proceed in the same way with the **Concatenate** function, replacing any sample in v_2 with R_{v_2}. Now consider v_2^i the largest suffix of $D_{(v_1 \cdot v_2)}$ contained in v_2, and $v^l = R_{v_2} \cdot u_0$. We use the fact that **Concatenate** looks at $|v^l|_{\texttt{high}} \geq |u_0| + |R_{v_2}|$ for replacing samples. This means that we choose R_{v_2} as a sample for v^l with probability $(|v^l|_{\texttt{high}} - |u_0|)/|v^l|_{\texttt{high}} \geq |R_{v_2}|/|v^l|_{\texttt{high}}$, and therefore the property is verified.

We now prove property (3). If v^l has just been created, it contains only one letter of weight 1, and obviously $|v^l|_{\texttt{low}} = |v^l|_{\texttt{high}} = |v^l|$. In addition, unless some R_{v_2} has been computed at line 19 of Algorithm 3.7 when the last letter was read, then $|v^l|$ is only augmented by some exactly known $|a|$ or $|u_0|$ compared to the previous step. Therefore the difference $|v^l|_{\texttt{high}} - |v^l|_{\texttt{low}}$ does not change, and by induction it remains smaller than $2\varepsilon'|v^l|_{\texttt{low}}/3$ which can only increase. Now consider R_{v_2} computed at line 19 and $v^l = R_{v_2} \cdot u_0$. We again consider v_2^i for the largest suffix in the decomposition of $v_1 \cdot v_2$ that is contained within v_2, as used in Algorithm 3.10, and v_2^{i-1} is the suffix immediately preceding v_2^i in that decomposition.

If $|v_2^{i-1}|_{\texttt{high}} > (1+\varepsilon')|v_2^i|_{\texttt{low}}$, then from the **Simplify** function, the difference between those two suffixes cannot be more than one letter, and then $v_2^i = v_2$. Therefore, we have $|R_{v_2} \cdot u_0|_{\texttt{high}} = |v_2|_{\texttt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\texttt{low}} = |v_2|_{\texttt{low}} + |u_0|$. We conclude by induction on $|v_2|$.

We end with the case $|v_2^{i-1}|_{\texttt{high}} \leq (1+\varepsilon')|v_2^i|_{\texttt{low}}$. By definition, $|R_{v_2} \cdot u_0|_{\texttt{high}} = |v_2^{i-1}|_{\texttt{high}} + |u_0|$ and $|R_{v_2} \cdot u_0|_{\texttt{low}} = |v_2^i|_{\texttt{low}} + |u_0|$. Therefore the difference $|v^l|_{\texttt{high}} - |v^l|_{\texttt{low}}$ is at most $\varepsilon'|v_2^i|_{\texttt{low}}$. Since the test at line 18 of Algorithm 3.7 (modified by Algorithm 3.10) was satisfied, we know that $|v_2^i|_{\texttt{low}} \leq 2|u_0|$. This implies that $\varepsilon'|v_2^i|_{\texttt{low}} \leq 2\varepsilon'(|v_2^i|_{\texttt{low}} + |u_0|)/3 \leq 2\varepsilon'|v^l|_{\texttt{low}}/3$, which concludes the proof. $\square$

From this we prove that the S_{w_l} can actually generate a $(1+\varepsilon')$-suffix sampling on the suffix decomposition, and that this decomposition is not too large so it will fit in our polylogarithmic memory.

Lemma 3.2.26. *Let $v, \mathcal{W}$ be an unfinished peak with a sampling maintained by the algorithm. Then $\mathcal{W}^{\otimes 2}$ over-samples an $(1+\varepsilon')$-suffix sampling on v, and $\mathcal{W}$ has size at most* $144(\log|v|)(\log n)/\varepsilon + \mathrm{O}(\log n)$.

Proof. The first property is a direct consequence of property (1) and (2) in Proposition 3.2.25, as in the proof of Lemma 3.2.15.

The second is a consequence of the (modified) **Simplify** used in Algorithm 3.10: D_{temp} is defined as the set of suffixes below with $m < l$ such that $|v^m|_{\texttt{high}} \leq (1+\varepsilon')|v^l|_{\texttt{low}}$. Because **Simplify** deletes all but one elements from D_{temp}, it follows that $|v^{l-2}|_{\texttt{high}} > (1+\varepsilon')|v^l|_{\texttt{low}}$. Now, from property (3) of Proposition 3.2.25 we have that $|v^l|_{\texttt{low}} \geq |v^l|_{\texttt{high}} - 2\varepsilon'|v^l|_{\texttt{low}}/3 \geq (1-2\varepsilon'/3)|v^l|_{\texttt{high}}$. Therefore we have that $|v^{l-2}|_{\texttt{high}} > (1+\varepsilon')(1-2\varepsilon'/3)|v^l|_{\texttt{high}}$

By successive applications, we have $|v^{l-6}|_{\texttt{high}} > (1+\varepsilon')^3(1-2\varepsilon'/3)^3|v^l|_{\texttt{high}}$. Now, as $|v^l|_{\texttt{high}} > |v^l|$ and $|v^l| \geq |v^l|_{\texttt{low}} \geq (1-2\varepsilon'/3)|v^l|_{\texttt{high}}$, we obtain $|v^{l-6}|/(1-2\varepsilon'/3) > (1+\varepsilon')^3(1-2\varepsilon'/3)^3|v^l|$. Equivalently, $|v^{l-6}| > (1+\varepsilon')^3(1-2\varepsilon'/3)^4|v^l|$

Thus, the size of the suffix decomposition is at most $6\log_{(1+\varepsilon')^3(1-2\varepsilon'/3)^4}|v| \leq 6\log|v|/\log(1+\varepsilon'/3 + \mathrm{O}(\varepsilon'^2)) \leq 144(\log|v|)(\log n)/\varepsilon + \mathrm{O}(\log(n))$. $\square$

Robustness. We first extend the notion of ε-approximation of words for a finite automaton (Definition 3.2.5) to any VPA when words are in Λ_Q.

Definition 3.2.27. *Let $R \subseteq Q^2$. Then R (ε, Σ)*-approximates *a balanced word $u \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$ on $\mathcal{A}$, if for all $p, q \in Q$: (1) $(p,q) \in R$ when $p \xrightarrow{u} q$; (2) u is (ε, Σ)-close to some word v satisfying $p \xrightarrow{v} q$ when $(p,q) \in R$.*

Then, we state an analogue of Theorem 3.2.10 for words in Λ_Q instead of Λ_1. We present the result as an algorithm with an output R as in Theorem 3.2.6. We also need to adapt it to the sampling we have. Indeed the suffixes from which we sample do not exactly match the peaks we want to compress, but we know that for each peak at least one suffix is within a factor $(1+\varepsilon')$ from Lemma 3.2.26.

Corollary 3.2.28. *Let $\mathcal{A}$ be a* VPA *with $m \geq 2$ states and Σ-diameter $d \geq 2$. There is an algorithm that:*

1. *Take as input $\varepsilon', \eta > 0$ and T k-factors of $w_1, \ldots, w_T$ of some weighted word $v \in \Lambda_Q$, such that $T = 4kt$, $t = 2\lceil 4dm^3(\log 1/\eta)/\varepsilon' \rceil$ and $k = \lceil 4dm/\varepsilon' \rceil$;*
2. *Output a set $R \subseteq Q \times Q$ that (ε', Σ)-approximates v on $\mathcal{A}$ with bounded error η, when each factor w_i come from an independent k-factor sampling on $\widehat{v}$.*

Let v' be obtained from v by at most $\varepsilon'|v|$ balanced deletions. Then, the conclusion is still true if the algorithm is given an independent k-factor sampling on $\widehat{v'}$ for each w_i instead, except that R now provides a $(3\varepsilon', \Sigma)$-approximation. Last, each sampling can be replaced also by an over-sampling.

Proof. The argument is similar to the one of Theorem 3.2.10, and we use again as a subroutine the algorithm of Theorem 3.2.6 for $\widehat{\mathcal{A}}$ with restricted alphabet $\widehat{\Sigma}'$, where $\Sigma' = (\Sigma_+ \cup \Sigma_- \cup \Sigma_\infty)$. Remind that $\mathcal{A}$ is Σ'-closed and its Σ'-diameter is the Σ-diameter of $\mathcal{A}$.

For the case when we do not have exact k-factor sampling on v however, we need to compensate for the prefix of v of size $\varepsilon'|v|$ that may not be included in the sampling. This introduces potentially an additional error of weight $2\varepsilon'|v|$ on the approximation R. □

We are now ready to state the robustness of our algorithm. For $u \in \Sigma^n$, we apply all compressions from lines 14 and 19 of Algorithm 3.7 using the tester from Corollary 3.2.28 with $\varepsilon' = \varepsilon/(6\log n)$ and $\eta' = \eta/n$. This leads to a final $R_{\text{temp}} \in \Sigma_Q$.

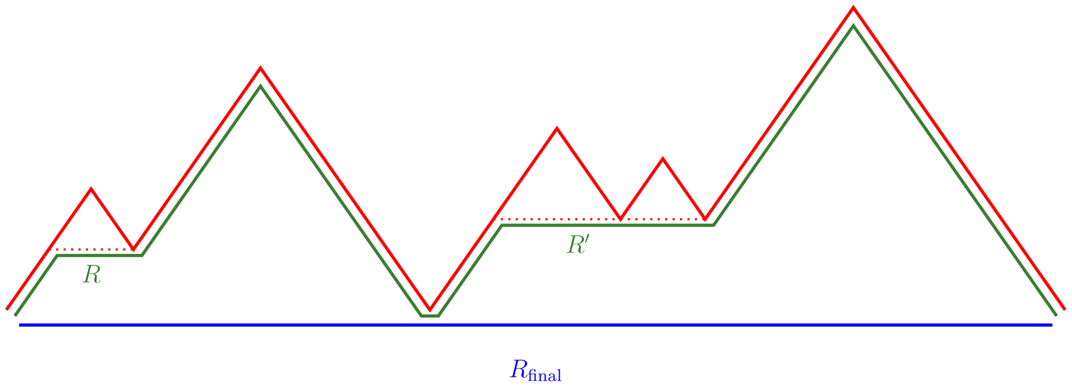


Figure 3.9: Constructing the words u_0, u_1 and u_2 as in Lemma 3.2.29 where $\text{Depth}(R_{\text{final}}) = 2$

Lemma 3.2.29. *Let $\mathcal{A}$ a* VPA *recognizing L and let $u \in \Sigma^n$. Let R_{final} the final value of R_{temp} in the Algorithm 3.7 with sketches. If $u \in L$, then $R_{\text{final}} \in L$; and if $R_{\text{final}} \in L$, then $\mathsf{bdist}_\Sigma(u, L) \leq \varepsilon n$ with probability at least $1 - \eta$.*

Proof. One way is easy. A direct inspection reveals that each substitution of a factor w by a relation R enlarges the set of possible w-transitions.Therefore $R_{\text{final}} \in L$ when $u \in L$.

For the other way, consider some word u such that $R_{\text{final}} \in L$. Since the tester of Corollary 3.2.28 has bounded error $\eta' = \eta/n$ and was called at most than n times, none of the calls fails with probability at least $1 - \eta$. From now on we assume that we are in this situation.

Let $h = \text{Depth}(R_{\text{final}})$. We will inductively construct sequences $u_0 = u, \ldots, u_h = R_{\text{final}}$ and $v_h = R_{\text{final}}, \ldots, v_0$ such that for every $0 \leq l \leq h$, $u_l, v_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$, $\mathsf{bdist}_\Sigma(u_l, v_l) \leq 3(h-l)\varepsilon'|u_l|$ and $v_l \in L$. Furthermore, each word u_l will be the word u with some substitutions of factors by relations R computed by the tester. Therefore, $\text{Depth}(u_l)$ is well defined and will satisfy $\text{Depth}(u_l) = l$. This will conclude the proof using that $\mathrm{Depth}(u) \leq \log_{3/2} n$ from Lemma 3.2.20. Indeed, since $h \leq \mathrm{Depth}(u)$, it will give us $\mathsf{bdist}_\Sigma(u, v_0) \leq 6\varepsilon' n \log n \leq \varepsilon n$.

We first define the sequence $(u_l)_l$ (see Figure 3.9 for an illustration). Starting from $u_0 = u$, let u_{l+1} be the word u_l where some factors in Λ_Q have been replaced by a $(3\varepsilon', \Sigma)$-approximation in Σ_Q. These correspond to all the approximations eventually performed by the algorithm that did not involve a symbol already in Σ_Q. Some approximations are eventually collapsed together into a single symbol by the $\bullet$ operation (in Figure 3.9 this is the case for R'). Observe that after this collapse, the symbol is still a $(3\varepsilon', \Sigma)$-approximation. In particular, $u_h = R_{\text{final}}$, $u_l \in (\Sigma_+ \cup \Sigma_- \cup \Sigma_Q)^*$ and $\mathrm{Depth}(u_l) = l$ by construction.

We now define the sequence $(v_l)_l$ such that $v_l \in L$. Each letter of v_l will be annotated by an accepting run of states for $\mathcal{A}$. Set $v_h = R_{\text{final}}$ with an accepting run from p_{in} to q_f for some $(p_{in}, q_f) \in R_{\text{final}} \cap (Q_{in} \times Q_f)$. Consider now some level $l < h$. Then v_l is simply v_{l+1} where some letters $R \in \Sigma_Q$ in common with u_{l+1} are replaced by some factors in $w \in (\Lambda_Q)^*$ as explained in the next paragraph. Those letters are the ones that are present in u_l but not u_{l+1}, and are still present in v_{l+1} (i.e. they have not been further approximated down the chain from u_{l+1} to u_h, or deleted by edit operations moving up from v_h to v_{l+1}).

Let $w \in (\Lambda_Q)^*$ be one of those factors and $R \in \Sigma_Q$ its respective $(3\varepsilon', \Sigma)$-approximation.

By hypothesis R is still in v_{l+1} and corresponds to a transition (p, q) of the accepting run of v_{l+1}. We replace R by a factor w' such that $p \xrightarrow{w'} q$ and $\mathsf{bdist}_\Sigma(w, w') \leq 3\varepsilon'|w|$, and annotate w' accordingly. By construction, the resulting word v_l satisfies $v_l \in L$ and $\mathsf{bdist}_\Sigma(u_l, v_l) \leq 3(h - l)\varepsilon'|u_l|$. $\square$

3.3 Input/Output Streaming Algorithms

3.3.1 Reversing the Input stream

The naive algorithm for reversing the Input stream using p passes in total (i.e. $p \geq 2$) is to copy to memory the first $2n/p$ bits of the input stream, then write them on the output stream in the correct order. During the next pair of passes, we do the same thing for the next $2n/p$ bits of the input. This approach clearly uses $\mathrm{O}(n/p)$ memory space. We will show in Section 4.2.1 that if we restrict ourselves to the Read-Only/Burn-Only model it is optimal.

For other models, we however have algorithms that perform better than the naive one.

3.3.1.1 With either Read-Write Input or Read-Write Output

If we can read the output stream again, we can use it to store some of the input in order to maximize our use of memory. While every bit written on the wrong part of the input stream will have to go through our algorithm's memory in order to be written at the correct place eventually, if the bits that we must moved are placed at well chosen locations along the stream, we can move many of them in one pass with just $\mathrm{O}(\log n)$ bits of memory, each time moving one, discarding it from memory and storing the next one with its destination.

Theorem 3.3.1. *There is a deterministic algorithm such that, given n and $p \leq \sqrt{n}$, it is a p-pass RO/RW streaming algorithm for* Reverse *on Σ^n with space $\mathrm{O}(\log n + (n \log |\Sigma|)/p^2)$ and expansion* 1.

We first present Algorithm 3.11, which performs $\mathrm{O}(\sqrt{n})$ passes and uses memory space $\mathrm{O}(\log n + \log |\Sigma|)$. It works by copying blocks of size $\mathrm{O}(\sqrt{n})$ directly from the input stream to the output stream without reversing them (otherwise there would not be enough space in the memory), but writing different blocks in the correct order pairwise. In addition, during each pass on the output, it moves one element from each block already copied to the correct place. Since blocks have as many elements as there are passes left after they are copied, the output stream is in the correct order at the end of the execution.

The real algorithm for generic p and s will use Algorithm 3.11 with alphabet Σ set to be $\Sigma^{s/\log|\Sigma|}$.

```
Data Structure:
$p = \sqrt{2n}$ passes, $t \leftarrow 1$ // current pass
$i_1 \leftarrow p$ // last index of the current block of $X$
$R$ register for one letter
$l$ original index of $R$ // total memory $\mathrm{O}(\log n + \log|\Sigma|)$
Code:
While {$t \leq p$}
  If $t > 1$ then $(R,l) \leftarrow (Y[n-i_t], n-i_t)$
  If $i_t < n$ then
    $Y[n-i_t-(p-t), n-i_t] \leftarrow X[i_t-(p-t), i_t]$ // Order is unchanged
    $i_{t+1} \leftarrow i_t + p - t$
  For $m = t-1$ to 1
    Put $R$ in the right place $Y[l']$ // $l'$ computed from $l,m,p,n$
    $(R,l) \leftarrow (Y[l'], l')$
  $t \leftarrow t+1$
```

Algorithm 3.11: RO/RW streaming algorithm for Reverse

Proof of Theorem 3.3.1. We will prove that Algorithm 3.11 satisfies the theorem when $p = 2\sqrt{n}$. First, observe that every element of the input is copied on the output, as $\sum_{t=1}^{p}(p-t) = p(p+1)/2 = n + \sqrt{n} \geq n$.

Let $1 \leq t < p$. The block $X[i_t-(p-t)-1, i_t]$ is initially copied at line 10 on $Y[n-i_t-(p-t)-1, n-i_t]$. Therefore only $Y[n-i_t]$ is correctly placed. Let B_t be $\{n-i_t-(p-t), n-i_t-1\}$. Then B_t denotes the indices (on Y) of elements copied during the t-th iteration of the While loop that are incorrectly placed. For each $l \in B_t$, the correct place for $Y[l]$ is in $\{n-i_t+1, n-i_t+p-t\} = \{n-i_{t-1}-(p-t+1), n-(i_{t-1}+1)\}$, which is B_{t-1} (where by convention B_0 is defined with $i_0 = 0$). Therefore, in the $(t+1)$-th iteration of the while loop, the first $Y[l]$ we place correctly goes from $l = n - i_{t+1} + 1 = n - i_t - (p-t) \in B_t$ to some $l' \in B_{t-1}$. Then recursively the previous value of $Y[l']$ goes in B_{t-2}, and so on until we reach B_0 where nothing was written initially.

Thus, the For loop places correctly one element of each of $B_{t-1}, B_{t-2}, \dots, B_1$. Observe that B_m has at most $p - m$ elements incorrectly placed. Moreover, there are $(p - m)$ remaining iterations of the While loop after B_m is written. Therefore all elements are places correctly when Algorithm 3.11 ends.

For $p = 2\sqrt{n}$, the complexity can be seen by examining Algorithm 3.11. For the general case, the algorithm treats groups of $m = 4n/p^2$ letters as though they were just one letter of alphabet Σ^m, then runs Algorithm 3.11. This requires memory space $\mathrm{O}(\log(n/m) + (n/m)(\log|\Sigma^m|)/p^2) = \mathrm{O}(\log n + (n \log|\Sigma|)/p^2)$. □

We also prove a similar result in the Read-Write/Write-Only model. The proof structure is the same, and it uses Algorithm 3.12 instead as a subroutine.

Theorem 3.3.2. *There is a deterministic algorithm such that, given n and $p \leq \sqrt{n}$, it is a p-pass RW/WO streaming algorithm for* Reverse *on Σ^n with space* $\mathrm{O}(\log n + (n\log|\Sigma|)/p^2)$ *and expansion* 1.

```
Data Structure:
$p = \sqrt{2n}$ passes, $t \leftarrow 1$ // current pass
$i_1 \leftarrow n$ // first/last index of the current block of $X$
$R$ register for one letter
$l$ original index of $R$ // total memory $\mathrm{O}(\log n + \log|\Sigma|)$
Code:
While $\{t \leq p\}$
  $(R, l) \leftarrow (X[t], t)$
  $Y[n - i_t + 1, n - i_t + t] \leftarrow X[i_t, i_t + t - 1]$ // Order is unchanged
  $i_{t+1} \leftarrow i_t - t - 1$
  For $m = 1$ to $(p - t)$
    Put $R$ in $X[l']$ // $l'$ computed from $l, m, p, n$
    $(R, l) \leftarrow (X[l'], l')$
  $t \leftarrow t + 1$
```

Algorithm 3.12: RW/WO streaming algorithm for Reverse

Proof of Theorem 3.3.2. We will prove that Algorithm 3.12 satisfies the theorem when $p = 2\sqrt{n}$. First, we must define blocks more properly to explain what the algorithm does at line 12. X is divided into $p - 1$ blocks, the first one of length $p - 1$ and the last one of length 1. This covers the entirety of X. The last element in each block does not move, but every other element does in order to change the order of the block. The last index of a block, which is equal to some i_t, therefore becomes the first one by the time the block is copied. If R is in a block where the pivot is i_t, th algorithm will move it to its symmetrical position with regard to i_t.

When $X[i_t, i_t + t - 1]$ is copied in the t-th iteration, it therefore corresponds to the block that was originally in position $X[i_t - t + 1, i_t]$, and has been reversed. Therefore its correct position on Y is $Y[n - i_t + 1, n - (i_t - t + 1) + 1] = Y[n - i_t + 1, n - i_t + t]$. This means that each element is copied directly in its correct place by Algorithm 3.12.

For $p = 2\sqrt{n}$, the complexity can be seen by examining Algorithm 3.12. For the general case, as before the algorithm treats groups of $m = 4n/p^2$ letters as though they were just one letter of alphabet Σ^m, and then runs Algorithm 3.12. This requires memory space $\mathrm{O}(\log n + (n \log |\Sigma|)/p^2)$. $\square$

3.3.1.2 With Read-Write Input and Read-Write Output

In the case where both streams are Read-Write, an algorithm using a logarithmic number of passes can be much more powerful. While a logarithmic number of passes may seem like a lot, one can think of the streaming augmented with a sorting primitive model [38]. In this model, every use of the sorting primitive uses $\Theta(\log n)$ passes, presumably on three streams.

Theorem 3.3.3. *Algorithm 3.13 is a deterministic $O(\log n)$-passes RW/RW streaming algorithm for* Reverse *on Σ^n with space* $\mathrm{O}(\log n)$ *and expansion* 1.

Algorithm 3.13 proceeds by dichotomy. For simplicity, we assume that n is a power of 2, but the algorithm can easily be adapted while keeping $\lambda = 1$. At each step, it splits the input in two, copies one half to its correct place on the stream, then makes another pass to copy the other half, effectively exchanging them.

```
1  Data Structure:
2  $W_0 \leftarrow X$, $W_1 \leftarrow Y$ // Rename the streams as they will switch roles
3  $\alpha \leftarrow 0$ // Which stream is considered the input stream
4  $k \leftarrow n$ // Size of current blocks
5  Code:
6  While $k > 1$
7     $k \leftarrow k/2$
8     For $i = 1$ to $n/2k - 1$
9           $W_{1-\alpha}[(2i+1)k+1, (2i+2)k] \leftarrow W_\alpha[2ik+1, (2i+1)k]$ // One pass
10    For $i = 0$ to $n/2k - 1$
11          $W_{1-\alpha}[2ik+1, (2i+1)k] \leftarrow W_\alpha[(2i+1)k+1, (2i+2)k]$ // Another pass
12    $\alpha \leftarrow 1 - \alpha$n, erase $W_{1-\alpha}$ // Exchange the roles of the streams
13 $Y \leftarrow W_\alpha$ // Copy the final result on output tape
```

Algorithm 3.13: RW/RW streaming algorithm for Reverse

Proof of Theorem 3.3.3. Since the algorithm can read and write on both tapes, they perform very similar roles. We rename the input stream W_0 and the output stream W_1. By a simple recursion, we see that whenever a block $W_{1-\alpha}[tk, (t+1)k]$ is moved, it is moved in the place that $\mathrm{Reverse}(W_{1-\alpha}[tk, (t+1)k])$ will occupy. Therefore, the algorithm is correct.

Now we prove the bounds on s and p. Algorithm 3.13 never needs to remember a value, only the current index and current pass, so $s = \mathrm{O}(\log n)$. Since the length of blocks copied is divided by 2 at each execution of line 7, it ends after a logarithmic number of executions of the While loop. Each iteration of the While loop requires two passes, one for each of the two For loops (lines 8 and 10). Therefore the total number of passes is in $\mathrm{O}(\log n)$. $\square$

3.3.2 Sorting the Input stream

Sort is generally more useful, as it allows us to simulate streaming algorithms augmented with sorting primitives, but also more complex than Reverse. Even in the RW/RW model, we are not able to present a deterministic algorithm as efficient as Algorithm 3.13 for Reverse. With three streams, the problem becomes easy since we can Merge Sort two streams, and write the result of each step on the third one.

3.3.2.1 Merge Sort

We begin with an algorithm inspired from [11]. The algorithm works as a Merge Sort. We call B_i^t the i-th sorted block at the t-th iteration of the While loop, consisting of the sorted values of $X[2^t i+1, 2^t(i+1)]$. Since there is no third stream to write on when two blocks B_{2i-1}^t and B_{2i}^t are merged into B_i^{t+1}, we label each element with its position in the new block. Then both halves are copied on the same stream again so that they can be merged with the help of the labels. This improves the expansion (Definition 2.2.4) of [11] from n to $\log n$. However it is somewhat unsatisfying because when Σ is of constant size, our algorithm still has $\Omega(\log n)$ expansion.

Theorem 3.3.4. *Algorithm 3.14 is a deterministic* $\mathrm{O}(\log n)$*-pass RW/RW streaming algorithm for* Sort *on* Σ^n *with space* $\mathrm{O}(\log n)$ *and expansion* $\mathrm{O}((\log n)/\log|\Sigma|)$.

```
Data Structure:
$W_0 \leftarrow X$, $W_1 \leftarrow Y$ // Rename the streams as they will switch roles
$\alpha \leftarrow 0$ // Which stream is considered the input stream
$t \leftarrow 1$ // Current pass
$k \leftarrow 1$ // Size of the sorted blocks
Code:
Expand the input so that each element has a label of size $\log n$.
While $k < n$
  For $i = 1$ to $n/2k$
      Copy $B_{2i}^t$ on $W_{1-\alpha}$
  For $i = 1$ to $n/2k$
      For each $W_\alpha[j] \in B_{2i-1}^t \cup B_{2i}^t$
              $W_\alpha[j] \leftarrow$ index of $W_\alpha[j]$ in $B_i^{t+1}$
  For $i = 1$ to $n/2k$
      Copy $B_{2i}^t$ at the end of $W_\alpha$ after $B_{2i-1}^t$
  For $i = 1$ to $n/2k$
      For each $W_\alpha[j] \in B_{2i-1}^t$
              Write $W_\alpha[j]$ at its position on $W_{1-\alpha}$
  For $i = 1$ to $n/2k$
      For each $W_\alpha[j] \in B_{2i}^t$
              Write $W_\alpha[j]$ at its position on $W_{1-\alpha}$
  $\alpha \leftarrow 1-\alpha$ erase $W_{1-\alpha}$, $t \leftarrow t+1$, $k \leftarrow 2k$
$Y \leftarrow W_\alpha$
```

Algorithm 3.14: RW/RW streaming algorithm implementing Merge Sort

Proof. Since it is an implementation of the Merge Sort algorithm, Algorithm 3.14 is correct. Each iteration of the While loop corresponds to five passes on each tape, and therefore the total number of passes is in $\mathrm{O}(\log n)$. Since the algorithm only needs to remember the position of the heads, current elements and the counters k, t, it only uses memory $\mathrm{O}(\log n)$. Finally, since the label for each element uses at most space $\log n$ on the stream, the therefore the expansion is at most $(\log|\Sigma| + \log n)/\log|\Sigma| = \mathrm{O}((\log n)/\log|\Sigma|)$. □

3.3.2.2 Quick Sort

With a Quick Sort algorithm instead of a Merge Sort, we only need to store the current pivot (of size at most $\log n$), without labeling elements. However, Quick Sort comes with its own issues: the expected number of executions of the While loop is $\mathrm{O}(\log n)$, but unless we can select a good pivot it is $\Omega(n)$ in the worst case. For this reason, we use a randomized Las Vegas algorithm.

A block in Algorithm 3.15 is a set of elements that are still pairwise unsorted, i.e. elements that have the same relative positions to all pivots so far. The block B_i^t is the i-th lower one during the t-th iteration of the While loop, and P_i^t is its pivot. The block B_{2i-1}^{t+1} consists of all elements in B_i^t lower than P_i^t and all elements equal P_i^t with a lower index. The block B_{2i}^{t+1} is the complementary. Algorithm 3.15 marks the borders of blocks with the symbol $\sharp$.

Algorithm 3.15 selects each pivot P_i^t at random among the elements of B_i^t. While it may not do so uniformly with only one pass because $|B_i^t|$ is unknown, it has an upper bound $k \geq |B_i^t|$. Algorithm 3.15 selects $l \in \{1, \dots, k\}$ uniformly at random, then picks l_i the remainder modulo $2^{\lceil \log |B_i^t| \rceil}$ of l. This can be computed in one pass with $\mathrm{O}(\log n)$ space by updating l_i as the lower bound on $|B_i^t|$ grows. It uses $P_i^t = B_i^t[l_i]$. P_i^t is selected uniformly at random from a subset of size at least half of B_i^t, which guarantees that with high probability $min(|B_{2i-1}^{t+1}|, |B_{2i}^t|) = \Omega(|B_i^t|)$.

Theorem 3.3.5. *For all $0 < \varepsilon \leq 1/2$, Algorithm 3.15 is a randomized $\mathrm{O}((\log n)/\varepsilon)$-pass RW/RW Las Vegas streaming algorithm for Σ^n with failure ε, space $\mathrm{O}(\log n)$ and expansion $\mathrm{O}(1)$.*

```
Data Structure:
$W_0 \leftarrow X$, $W_1 \leftarrow Y$ // Rename the streams as they will switch roles
$\alpha \leftarrow 0$ // Which stream is considered the input stream
$t \leftarrow 1$ // Current pass
$K \leftarrow 1$ // Number of sorted blocks
Code:
While $K > 0$
  Abort if the total number of passes is $\Omega((\log n)/\varepsilon)$
  Expand $W_0$ adding $O(K)$ space for $\sharp$ and pivots
  For $i = 1$ to $K$
      Find $P_i^t$ at random
      $W_{1-\alpha}[i] \leftarrow P_i^t$
      Replace $P_i^t$ with a $\perp$ on $W_\alpha$
  For $i = 1$ to $K$
      Copy $W_{1-\alpha}[i] = P_i^t$ at the start of $B_i^t$ on $W_\alpha$
  For $i = 1$ to $K$
      Write all elements in $B_{2i-1}^{t+1}$ on $W_{1-\alpha}$
      Write $\sharp P_i \sharp$ on $W_{1-\alpha}$.
      Leave space for the rest of $B_{2i}^{t+1}$
  For $i = 1$ to $K$
      Write all elements in $B_{2i}^{t+1}$ in the space left on $W_{1-\alpha}$
  $\alpha \leftarrow 1 - \alpha$; Erase $W_{1-\alpha}$; $t \leftarrow t + 1$
  $K \leftarrow$ new number of unsorted blocks // using an additional pass
$Y \leftarrow W_\alpha$
```

Algorithm 3.15: RW/RW streaming algorithm implementing Quick Sort

Proof. Algorithm 3.15 is an implementation of the Quick Sort algorithm, and is therefore correct. Its correctness does not depend on the quality of the pivots. The bound on the memory and the expansion are direct. While it uses additional symbols $\perp$ and $\sharp$, they can be easily replaced by an encoding with symbols of Σ appearing in the input with only $\mathrm{O}(1)$ expansion. Because for each i and t, with high probability $min(|B_{2i-1}^{t+1}|, |B_{2i}^t|) = \Omega(|B_i^t|)$, with high probability Algorithm 3.15 terminates in $\mathrm{O}(\log n)$ passes. $\square$

Chapter 4

Lower Bounds

4.1 Recognizing Priority Queues with Timestamps in one pass

A tight lower bound on the space complexity of recognizing priority queues without timestamps in streaming comes from the lower bound on the streaming complexity of Dyck languages by [32]. By attributing to each parenthesis the priority $2h + k$, where h is the height of the parenthesis, and k is 0 for $''(''$ and $'')''$ and 1 for $''[''$ and $'']''$, we can reduce DYCK[2] to PQ.

Corollary 4.1.1 (Chakrabarti,Cormode, Kondapalli,McGregor)**.** *Every p-pass randomized streaming algorithm recognizing* PQ(N) *with bounded error* $1/3$ *requires memory space* $\Omega(\sqrt{N}/p)$ *for inputs of length* N.

However, the original algorithm by Chu, Kannan and McGregor [12] was for priority queues augmented with timestamps. In this model, extractions are of the form $(\mathtt{ext}(a), i)$, with the promise that $u[i] = \mathtt{ins}(a)$ matches this extraction. More precisely:

Definition 4.1.2 (PQ-TS)**.** *Let* $\Sigma = \{ins(a), ext(a) : a \in \{0, 1, \ldots, U\}\} \times \mathbb{N}$. *Let* $u \in \Sigma^n$. *Then* $u \in$ PQ-TS(U) *if and only if:*

- $u \in$ COLLECTION(Σ).
- $u[1, \ldots, n][1] \in$ PQ(U)
- $u[i][2] = i$ *when* $u[i][1] = ins(a)$

Note that the first and third properties guarantee that timestamps are actually what we could intuitively call timestamps. Indeed if $u[i] = (\mathtt{ext}(a), j)$, $u[j][1] = \mathtt{ins}(a)$, as there must exists an element $(\mathtt{ins}(a), j)$ for u to be in COLLECTION(Σ). Note that timestamps also have no collisions, which means that for $u \in$ PQ-TS, if $u[i][2] = u[i'][2] = j$, then either $j = i$ or $j = i'$. Because the first and third properties are trivial to check in streaming (the first only requires a single hashcode, the third not even that), recognizing PQ-TS cannot be harder than recognizing PQ.

We also remark that we cannot reduce DYCK[2] to PQ-TS in the same way: if parentheses have timestamps, then an algorithm can just make a hashcode with the type of parenthesis and the timestamp, and verify that an input is in the language with memory $\mathrm{O}(\log n)$. To prove a lower bound on the streaming complexity of recognizing PQ-TS, we therefore need to find a hard distribution on PQ that does not correspond to well-parenthesized words.

In [12], the authors give a sketch of what they believe to be a hard distribution for PQ-TS. However, as they note, proving the hardness of this distribution using communication complexity tools does not seem to be feasible. The reason for that is that the timestamps in the instances in the support of their distribution contain information, and cannot be strictly deduced from $u[1, \dots, n][1]$. We will instead give a hard distribution for which the timestamps can be computed by the algorithm from $u[1, \dots, n][1]$ using $\mathrm{O}(\log n)$ memory.

Theorem 4.1.3. *Every p-pass randomized streaming algorithm recognizing* $\text{PQ-TS}(3N/2)$ *with bounded error* $1/3$ *requires memory space* $\Omega(\sqrt{N}/p)$ *for inputs of length* N.

As in many proofs of lower bounds for streaming complexity before, the technique we used in our proof is to reduce a communication problem, where each player has part of the input, to the streaming problem (with our hard distribution). For any streaming algorithm using p passes and s memory space, the k players can simulate the algorithm on their part of the input and send the memory to the next player, which results in a randomized public-coins message-passing communication protocol with communication complexity at most ksp. By contraposition, a lower bound on the communication complexity of such a protocol implies a lower bound on the space complexity of a streaming algorithm for our problem.

One of the big difficulties with this technique is that it is often hard to prove lower bounds in communication complexity for more than 2 players, but in this case we would most likely need to use $\Theta(\sqrt{n})$ players to prove our bound. We use a technique from [32] here; they reduce a $\mathrm{O}(\sqrt{n})$-player communication problem consisting of $\mathrm{O}(\sqrt{n})$ nested independent 2-player communication problems to the streaming problem, then use a direct sum argument to bound the information complexity of that problem with the information complexity of a 2-player problem, and conclude by using the information complexity as a lower bound for communication complexity.

Here however, this approach is made even harder by the fact that we were not able to find a decomposition into nested independent 2-player problems with a satisfying lower bounds: instead we have 3-player problems, which both complicates the proof of the lower bound for information complexity of the small problem, and the direct sum to bound the information complexity of the global $3m$-player problem.

4.1.1 Reduction from communication protocols to the streaming algorithms

4.1.1.1 Hard distribution

For our hard distribution to have easily computable timestamps, we should make it so the insertions and extractions at each index are in disjoint sets that depend on as few variables other than the index as possible. If every extraction can take two values, and therefore be sometimes larger and sometimes smaller than another extraction, then we may have a hard set of instances. Consider the following set of instances of length $N = (2n + 2)m$:

RAINDROPS(m, n) (see Figure 4.1)

- For $i = 1, 2, \dots, m$, repeat the following pattern:
  - For $j = 1, 2, \dots, n$, insert either $v_{i,j} = 3(ni - j)$ or $v_{i,j} = 3(ni - j) + 2$
  - Insert either $b_i = 3(ni - (k_i - 1)) + 1$ or $b_i = 3(ni - k_i) + 1$, for some $k_i \in [2, n]$
  - Extract $v_{i,1}, v_{i,2}, \dots, v_{i,k_i-1}, b_i$ in decreasing order
- Extract everything left in decreasing order

Observe that because the k_i are not fixed, if u is a word formed this way, the index of $v_{i,j}$ or b_i cannot be deduced only from i. However, the timestamps can still be easily computed by a streaming algorithm:

Lemma 4.1.4. *If there exists a p-pass randomized streaming algorithm recognizing* PQ-TS$((2n+2)m)$ *with bounded error ε and memory space s, then there exists a p-pass randomized streaming algorithm recognizing the instances of* RAINDROPS(m, n) *which are in* PQ *with bounded error ε and memory space $s + \mathrm{O}(\log(mn))$.*

Proof. When reading an extraction that is part of the first patter (i.e. `ext`$(v_{i,j})$ with $j < k_i$), the algorithm knows the timestamp is $t - n - 1$, or $t - n - 2$, depending on whether it saw `ext`(b_i) already, with t the current index. During the final sequence when everything remaining is extracted, the timestamp of `ext`$(v_{i,j})$ at index t is $t - (m - i)(2n + 2) - n - 2$. Therefore the algorithm only needs to keep track of the index at which `ins`(b_i) is read until it reads `ext`(b_i) to be able to compute all the timestamps as extractions are read, and simulate the first algorithm on RAINDROPS(m, n). □

Using this result, we now only consider whether an instance u of RAINDROPS(m, n) is in PQ. There is only one potential error in each occurrence of the pattern that can make u not be in PQ. Indeed, $v_{i,1}, v_{i,2}, \dots, v_{i,k_i-1}, b_i$ are extracted in decreasing order, and the only insertion before that which might be larger than the minimal extraction in that sequence is v_{i,k_i}. This only happens if v_{i,k_i} is as large as it can be (i.e. $3(ni - k_i) + 2$) and b_i is as small as it can be (i.e. $b_i = 3(ni - k_i) + 1$).

Given such an instance as a stream, an algorithm for PQ-TS must decide if an error occurs with `ext`(b_i) and `ins`(v_{i,k_i}) as the witnesses, for some i. Intuitively, if the memory space is less than εn, for a small enough constant $\varepsilon > 0$, then the algorithm cannot remember all the values $(v_{i,j})_j$ when b_i is extracted, and therefore cannot check that $v_{i,k_i} = 3(ni - k_i) + 2$, which is a necessary condition for an error to exist in the i-th occurrence of the pattern. The next opportunity is during the last sequence of extractions. But then, the algorithm would have to remember all values $(b_i)_i$ to check that $b_i = 3(ni - k_i) + 1$, which is again impossible if the memory space is less than εm.

4.1.1.2 Reduction of a $3m$-player communication problem to the hard distribution

In order to formalize the intuition above, Lemma 4.1.5 first translates our problem into a communication one between $3m$ players as shown in Figure 4.2. Then we will analyze its complexity using information theory arguments.

Any insertion and extraction of an instance in RAINDROPS(m, n) can be described by its index and a single bit. Let $x_i \in \{0,1\}^n$ such that $v_{i,j} = 3(ni - j) + 2x_i[j]$. Similarly, let $d_i \in \{0,1\}$ such that $b_i = 3(ni - k_i) + 1 + 3d_i$. For simplicity, we write $\mathbf{x}$ instead of $(x_i)_{1 \le i \le m}$. Similarly, we use the notations $\mathbf{k}$ and $\mathbf{d}$. Then our related communication problem is:

WEAKINDEX(m, n)

- Input for players $(A_i, B_i, C_i)_{1 \le i \le m}$:
  - Player A_i has a sequence $x_i \in \{0,1\}^n$
  - Player B_i has $x_i[1, k_i - 1]$, with $k_i \in [2, \dots, n]$ and $d_i \in \{0,1\}$
  - Player C_i has $x_i[k_i, n]$
- Output: $f_m(\mathbf{x}, \mathbf{k}, \mathbf{d}) = \bigvee_{i=1}^{m} f(x_i, k_i, d_i)$, where $f(x, k, d) = [(d = 0) \land (x[k] = 1)]$
- Communication settings:
  - One round: each player sends a message to the next player according to the diagram $A_1 \to B_1 \to A_2 \to \cdots \to B_m \to C_m \to C_{m-1} \to \cdots \to C_1$.
  - Multiple rounds: If there is at least one round left, C_1 sends a message to A_1, and then players continue with the next round.

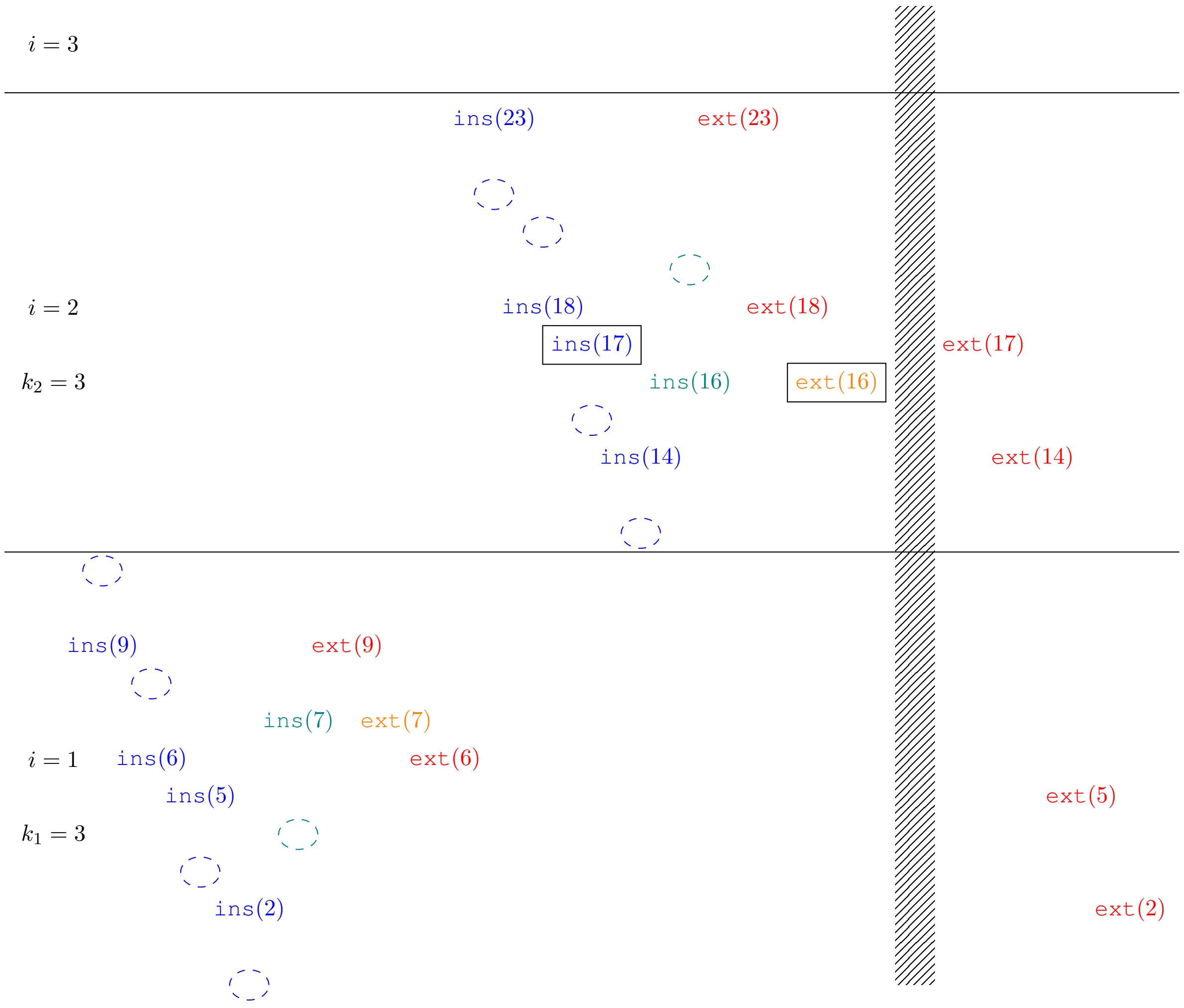


Figure 4.1: First two occurrences of the pattern in an instance of RAINDROPS$(m, 4)$ with one error in the second occurrence ($b_2 = 16$ is extracted before $v_{2,k_2} = 17$). The hatched part corresponds the factor of u with all other occurrences of the pattern, not represented here. For each insertion, the dashed ellipse below or above represents the other possible value for that $v_{i,j}$ or b_i. Insertions and extractions of b_i are respectively represented in cyan and orange. Note that in the first occurrence of the pattern, $b_i > v_{i,k_i-1}$ and therefore ext(b_i) comes before ext(v_{i,k_i-1}).

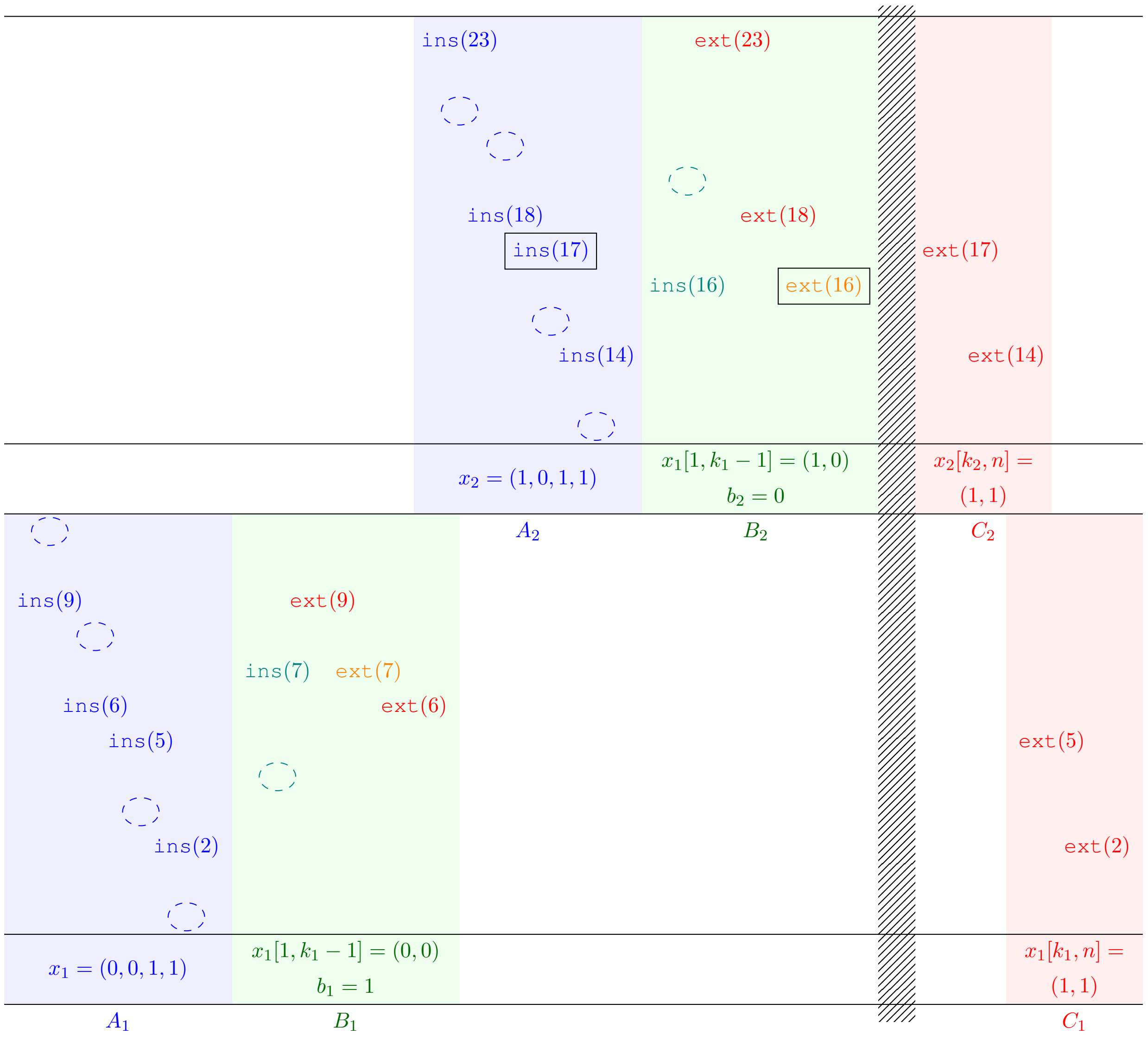


Figure 4.2: Cutting the same instance of RAINDROPS$(m,4)$ as in Figure 4.1 into $3m$ pieces to make it an instance of WEAKINDEX(m,n). Players' input are within each corresponding region.

We named our problem WEAKINDEX because of the resemblance to AUGMENTEDINDEX used in [32] and [26] to prove the lower bound on DYCK$[2]$. The main difference here is that instead of having only two players per level (in the reduction from AUGMENTEDINDEX to DYCK$[2]$, Alice would have the same input as here, but also be in control of the final set of closing parentheses), there are three. Note that neither Alice nor Charlie know d, so they cannot by themselves determine the value of f. However if we had "Charlice" as super-player taking both Alice's and Charlie's place, she would know k which would make her much more powerful than Alice in the original AUGMENTEDINDEX, as she would be able to send $x[k]$ to Bob using only one bit of communication.

We now prove the reduction from WEAKINDEX(n, m) to PQ$(3mn)$ using the standard method.

Lemma 4.1.5. *If there exists a p-pass randomized streaming algorithm for deciding if an instance of* RAINDROPS(m, n) *is in* PQ$(3mn)$ *with memory space $s(m, n)$ and bounded error ε, Then there is a p-round randomized protocol for* WEAKINDEX(n, m) *with bounded error ε such that each message has size at most $s(m, n)$.*

Proof. Assume that there exists a p-pass randomized streaming algorithm with memory space $s(m, n)$, that decides if an instance of RAINDROPS(m, n) belongs or not to PQ-TS$(3nm)$. Each instance of RAINDROPS(m, n) can be encoded by an input of WEAKINDEXnm, where each of the $3m$ players has one part of it.

Each player simulates alternatively the algorithm, until it reaches a part of input belonging the next player. At this points, the first player sends the current memory to the next player, who continues the simulation. Since the algorithm uses at most memory space $s(m, n)$, the current memory can be encoded using $s(m, n)$ bits. Each pass corresponds to one round of communication between players, which finishes the proof. □

4.1.2 Lower bound of the communication complexity of RAINDROPS(m, n)

4.1.2.1 Direct sum and reduction of a 3-player communication problem to the $3m$-player problem

We now show that we can reduce a single instance of WEAKINDEX$(n, 1)$ with 3 players to the general problem WEAKINDEX(n, m) with $3m$ players. In order to do so we use the direct sum property of the information cost. For this we need to choose a collapsing distribution where f is always 0. Let μ_0 be the uniform distributions on the $(x, k, d) \in 0, 1^n \times [2, n] \times 0, 1$ verifying $f(x, k, d) = 0$.

The resulting communication problem will have asymmetrical constraints, as shown on Figure 4.3. For each i, A_i can just send a message to B_i. However if B_i wants to send a message to C_i, that message has to transit through all intermediate players, in particular its information content has to go from B_m to C_m. Similarly, any information that C_1 wants to send to A_i has to transit from C_1 to A_1.

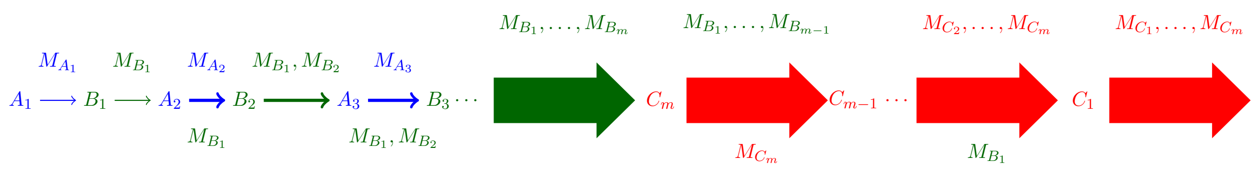


Figure 4.3: While for each i, A_i may send a relatively large message to B_i, all messages from all B_i to all C_i transit through the same point, and the same is true of all messages from all C_i to all A_i. Therefore in the 3-player problem, the information sent by Bob to Charlie and sent by Charlie to Alice has to be less than a constant amount. This figure only represents the first pass: otherwise there would be thick arrows everywhere since each A_i also has to send the messages from $C_{i'}$ to $A_{i'}$ for $i' > i$.

Lemma 4.1.6. *If there is a p-round randomized protocol P for* $\textsc{WeakIndex}(n, m)$ *with bounded error ε and messages of size at most $s(m, n)$, then there is a $(p + 1)$-round randomized protocol P' for* $\textsc{WeakIndex}(n, 1)$ *with bounded error ϵ, and transcript Π' satisfying $|\Pi'| \leq 3(p + 1)s(m, n)$ and $\max\{\mathrm{I}_{\mu_0}(D : \Pi'_B|X, K), \mathrm{I}_{\mu_0}(K, D : \Pi'_C|X)\} \leq \frac{p+1}{m}s(m, n)$.*

Given a protocol P, we show how to construct another protocol P' for any instance (x, k, d) of $\textsc{WeakIndex}(n, 1)$. In order to avoid any confusion, we denote by A, B and C the three players of P', and by $(A_i, B_i, C_i)_i$ the ones of P.

Protocol P'

- Using public coins, all players generate uniformly at random $j \in \{1, \dots, m\}$, and $x_i \in \{0, 1\}^n$ for $i \neq j$
- Players A, B and C set respectively their inputs to the ones of A_j, B_j, C_j
- For all $i > j$, Player B generates, using its private coins, uniformly at random $k_i \in [2, n]$, and then it generates uniformly at random d_i such that $f(x_i, k_i, d_i) = 0$
- For all $i < j$, Player C generates, using its private coins, uniformly at random $k_i \in [2, n]$, and then it generates uniformly at random d_i such that $f(x_i, k_i, d_i) = 0$
- Players A, B and C run P as follows. A simulates A_j only, B simulates B_j and $(A_i, B_i, C_i)_{i>j}$, and C simulates C_j and $(A_i, B_i, C_i)_{i<j}$.

Observe that A starts the protocol if $j = 1$, and C starts otherwise. Moreover C stops the simulation after p rounds if $j = 1$, and after $p + 1$ rounds otherwise.

Proof. We show that P' satisfies the conditions of Lemma 4.1.6. For all $i \neq j$, $f(x_i, k_i, a_i) = 0$. Therefore $f_m(\mathbf{X}, \mathbf{k}, \mathbf{d}) = f(x_j, k_j, a_j) = f(x, k, a)$, and P' has the same bounded error than P.

Let us now show that P' satisfies the conditions on its transcript under the distribution μ_0. Let Π, Π' be the respective transcripts of P, P'. For convenience, we write $\Pi_{C_{m+1}} = \Pi_{B_m}$ and $\Pi_{B_0} = \Pi_{C_m}$. As always, the public coins of a protocol are included in its transcript.

First, each player of P' sends 3 messages by round, and there are $(p + 1)$ rounds. Since each message has size at most $s(m, n)$, we derive that the length of Π' is at most $3(p + 1)s(m, n)$.

Then, in order to prove that there is only a small amount of information in the transcripts of Bob and Charlie, we show a direct sum of some appropriated notion of information cost. Consider first the transcript of Player C_1. Because of the restriction on the size of his messages, we know that $|\Pi_{C_1}| \leq (p + 1)s(m, n)$. From this we derive a first inequality on the amount of information this transcript can carry, using that the entropy of a variable is at most its number of bits:

$$\mathrm{I}_{\mu_0}(\mathbf{K}, \mathbf{D} : \Pi_{C_1}|\mathbf{X}) \leq |\Pi_{C_1}| \leq (p + 1)s(m, n).$$

We now use the chain rule in order to get a bound about the information carried by P' on a single instance.

$$\begin{aligned}
\mathrm{I}_{\mu_0}(\mathbf{K},\mathbf{D}:\Pi_{C_1}|\mathbf{X}) &= \sum_{j=1}^{m}\mathrm{I}_{\mu_0}((K_i,D_i)_{i\le j}:\Pi_{C_1}|\mathbf{X},(K_i,D_i)_{i<j}) \quad \text{(by chain rule)}\\
&\ge \sum_{j=1}^{m}\mathrm{I}_{\mu_0}(K_j,D_j:\Pi_{B_{j-1}}|\mathbf{X},(K_i,D_i)_{i<j}) \quad \text{(by DPI (Lemma 2.4.6))}\\
&\ge \sum_{j=1}^{m}\mathrm{I}_{\mu_0}(K_j,D_j:\Pi_{B_{j-1}}|\mathbf{X}) \quad \text{(by Fact 2.4.7)}\\
&= m\times\mathrm{I}_{\mu_0}(K_J,D_J:\Pi_{B_{J-1}}|\mathbf{X},J) \quad \text{(by conditioning on } J)\\
&= m\times\mathrm{I}_{\mu_0}(K_J,D_J:\Pi_{B_{J-1}},J,(X_i)_{i\ne J}|X_J) \text{ (independence of } J,(X_i)_{i\ne J})\\
&= m\times\mathrm{I}_{\mu_0}(K,D:\Pi'_C|X) \quad \text{(since } J,(X_i)_{i\ne J} \text{ are public coins of } P').
\end{aligned}$$

We then do similarly for Player B_m and therefore conclude the proof. First the size bound on messages of B_m gives $\mathrm{I}_{\mu_0}(\Pi_{B_m}:\mathbf{D}|\mathbf{X},\mathbf{K})\le(p+1)s(m,n)$. Then as before we get:

$$\begin{aligned}
\mathrm{I}_{\mu_0}(\mathbf{D}:\Pi_{B_m}|\mathbf{X},\mathbf{K}) &= \sum_{j=1}^{m}\mathrm{I}_{\mu_0}(D_j:\Pi_{B_m}|\mathbf{X},\mathbf{K},(D_i)_{i>j})\ge\sum_{j=1}^{m}\mathrm{I}_{\mu_0}(D_j:\Pi_{C_{j+1}}|\mathbf{X},\mathbf{K},(D_i)_{i>j})\\
&\ge m\times\mathrm{I}_{\mu_0}(D_J:\Pi_{C_{J+1}},J,(X_i)_{i\ne J}|X_J,K_J)=m\times\mathrm{I}_{\mu_0}(D:\Pi'_B|X,K).
\end{aligned}$$

□

4.1.2.2 Information complexity of the 3-player problem

We now prove a trade-off between the bounded error of a protocol for a single instance of WEAKINDEX$(n,1)$ and its information cost. The proof involves some of the tools of [26] but with some additional obstacles to apply them. The average encoding theorem tells us that because of the small mutual information between the transcript and the input under our distribution μ_0, if we change the inputs of Alice and Charlie without affecting Bob's input and while staying in the support of μ_0, then the transcript does not change too much (Lemma 4.1.7). Lemma 4.1.8 says that the same is true if we change Bob's input without affecting Alice and Charlie or leaving the support of μ_0. However since f is always 0 on the support of μ_0 that is not conclusive by itself.

We will then want to apply the cut-and-paste Lemma to show that two inputs where f takes different values have almost always the same transcript. One difficulty lies in the fact that the cut-and-paste lemma only applies to 2-player protocols, but we have 3 players. For this we group Alice and Charlie together as one player. Note that we could not have done this from the start or we would not have obtained the result of Lemma 4.1.7. The whole process is summed up by Figure 4.4.

In the following lemmas, we denote by $\Pi(x,k,a)$ the random variable describing the transcript Π of our protocol for (x,k,a) an input of WEAKINDEX$(n,1)$.

Lemma 4.1.7. *Let P be a randomized protocol for* WEAKINDEX$(n,1)$ *with transcript* Π *satisfying* $|\Pi|\le\alpha n$ *and* $\mathrm{I}_{\mu_0}(K,D:\Pi_C|X)\le\alpha$. *Then*

$$\mathbb{E}_{x[1,l-1],l}\,\mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,1),\Pi(x[1,l-1]1X[l+1,n],l,1))\le 22\alpha,$$

where $l\in[\frac{n}{2}+1,n]$ and $x[1,l-1]$ are uniformly distributed.

Proof. We will first show that we can change the value of k for a given x without affecting too much the messages sent by Alice and Charlie. In other words, $\Pi_{A,C}(x, j, 1)$ and $\Pi_{A,C}(x, l, 1)$ will be roughly the same for $j < l$. We will then show that if $k = j$, with $l > j$ we can change $x[l]$ without affecting the $\Pi_{A,C}$ much either. This will allow us to deduce the inequality on $\Pi_{A,C}$. The inequality on Π will follow from the fact that if neither Bob's input nor the message that Bob receives change, then the message Bob sends should also stay the same.

From the second hypothesis and the data processing inequality (Lemma 2.4.6) we get that $\mathrm{I}_{\mu_0}(K, D : \Pi_{A,C}|X) \leq \alpha$, which after applying the average encoding lemma (Lemma 2.4.16) leads to $\mathbb{E}_{(x,k,d)\sim\mu_0} \mathrm{h}^2(\Pi_{A,C}(x, k, d), \Pi_{A,C}(x, K, D)) \leq \kappa\alpha$, with $\kappa = \frac{\ln 2}{2}$. We now restrict μ_0 by conditioning on $D = 1$. Then (X, K) is uniformly distributed. Moreover, since $D = 1$ with probability $2/3$ on μ_0, we get $\mathbb{E}_{x,k} \mathrm{h}^2(\Pi_{A,C}(x, k, 1), \Pi_{A,C}(x, K, 1)) \leq \frac{3}{2}\kappa\alpha$. Let J, L be uniform integer random variables respectively in $[2, \frac{n}{2}]$ and $[\frac{n}{2} + 1, n]$. Then the above implies $\mathbb{E}_{x,j} \mathrm{h}^2(\Pi_{A,C}(x, j, 1), \Pi_{A,C}(x, K, 1)) \leq 3\kappa\alpha$ and $\mathbb{E}_{x,l} \mathrm{h}^2(\Pi_{A,C}(x, l, 1), \Pi_{A,C}(x, K, 1)) \leq 3\kappa\alpha$. Using that $(u+v)^2 \leq 2(u^2+v^2)$, with the triangle inequality we get

$$\mathbb{E}_{x,j,l} \mathrm{h}^2(\Pi_{A,C}(x, j, 1), \Pi_{A,C}(x, l, 1)) \leq 12\kappa\alpha. \tag{4.1}$$

Using the convexity of h^2, we finally obtain for $b = 0, 1$:

$$\mathbb{E}_{x[1,l-1],j,l} \mathrm{h}^2(\Pi_{A,C}(x[1, l-1]bX[l+1, n], j, 1), \Pi_{A,C}(x[1, l-1]bX[l+1, n], l, 1)) \leq 24\kappa\alpha. \tag{4.2}$$

Now the chain rule allow us to measure the information about a single bit in $\Pi_{A,C}$ as

$$\begin{aligned} \mathrm{I}(X[L] : \Pi_{A,C}(X, J, 1)|X[1, L-1]) &= \mathbb{E}_{l\leftarrow L} \mathrm{I}(X[l] : \Pi_{A,C}(X, J, 1)|X[1, l-1]) \\ &= \frac{2}{n} \times \mathrm{I}(X[\tfrac{n}{2}+1, n] : \Pi_{A,C}(X, J, 1)|X[1, \tfrac{n}{2}]). \end{aligned}$$

Since the entropy of a variable is at most its bit-size, we get that the last term is upper bounded by $|\Pi_{A,C}|$, which is at most αn by the first hypothesis. Then combining equality 4.1 with the average encoding lemma (Lemma 2.4.16) and the triangle inequality leads to

$$\mathbb{E}_{x[1,l-1],j,l} \mathrm{h}^2(\Pi_{A,C}(x[1, l-1]0X[l+1, n], j, 1), \Pi_{A,C}(x[1, l-1]1X[l+1, n], j, 1)) \leq 14\kappa\alpha. \tag{4.3}$$

Combining equalities 4.2 and 4.3 gives

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi_{A,C}(x[1, l-1]0X[l+1, n], l, 1), \Pi_{A,C}(x[1, l-1]1X[l+1, n], l, 1)) \leq 62\kappa\alpha.$$

This is almost what we want for the theorem as $62\kappa < 22$, we just need to know that Π_B does not change too much either. Let R_B be the random coins of B. Since they are independent from all variables, including the messages, the previous inequality is still true when we concatenate R_B to $\Pi_{A,C}$. Then Π_B is uniquely determined from R_B once $K, D, X[1, K-1]$ are fixed, which is the case in that inequality. Therefore replacing R_B by Π_B can only decrease the distance, concluding the proof. $\square$

Lemma 4.1.8. *Let P be a randomized protocol for* WEAKINDEX$(n, 1)$ *with transcript Π satisfying $\mathrm{I}(D : \Pi_B|X, K) \leq \alpha$. Then*

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1, l-1]0X[l+1, n], l, 0), \Pi(x[1, l-1]0X[l+1, n], l, 1)) \leq 12\alpha,$$

where $l \in [\frac{n}{2}+1, n]$ and $x[1, l-1]$ are uniformly distributed.

Proof. Using the data processing inequality (Lemma 2.4.6) on the hypothesis gives $\mathrm{I}_{\mu_0}(D : \Pi | X, K)) \leq \alpha$. Therefore by average encoding (Lemma 2.4.16), $\mathbb{E}_{(x,k,d)\sim\mu_0} \mathrm{h}^2(\Pi(x,k,d), \Pi(x,k,D)) \leq \kappa\alpha$.

Let L be a uniformly random variable in $[\frac{n}{2}+1, n]$. Then $\mathbb{E}_{(x,l,d)\sim mu_0} \mathrm{h}^2(\Pi(x,l,d), \Pi(x,l,D)) \leq 2\kappa\alpha$. Using the convexity of h^2 and the fact that $X[l]$ is a uniform random bit, we derive

$$\mathbb{E}_{x[1,l-1],l,d} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,d), \Pi(x[1,l-1]0X[l+1,n],l,D)) \leq 4\kappa\alpha.$$

Since $D = 0$ with probability $1/2$ when $X[l] = 0$ and $K = l$, we finally get the two inequalities

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,0), \Pi(x[1,l-1]0X[l+1,n],l,D)) \leq 8\kappa\alpha,$$

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,1), \Pi(x[1,l-1]0X[l+1,n],l,D)) \leq 8\kappa\alpha,$$

leading to the conclusion using the triangle inequality and that $(u+v)^2 \leq 2(u^2+v^2)$. □

We end this section with the a lemma which combines both previous ones and applies the cut-and-paste property, where Players A, C are grouped.

Lemma 4.1.9. *Let P be a randomized protocol for* WEAKINDEX$(n,1)$ *with bounded error ϵ, and transcript Π satisfying $|\Pi| \leq \alpha n$ and $\max\{I(D : \Pi_B | X, K), I(K, D : \Pi_C | X)\} \leq \alpha$. Then $\alpha \geq (1-2\varepsilon)^2/100$.*

Proof. Let L be a uniform integer random variable in $[\frac{n}{2}+1, n]$. Remind that we enforce the output of P to be part of Π. Therefore, any player, and in particular B, can compute f with bounded error ε given Π. Since $f(x[1,l-1]0X[l+1,n],l,0) = 0$ and $f(x[1,l-1]1X[l+1,n],l,1) = 1$, the error parameter ε must satisfies

$$\mathbb{E}_{x[1,l-1],l} \|\Pi(x[1,l-1]0X[l+1,n],l,0) - \Pi(x[1,l-1]1X[l+1,n],l,0)\|_1 \geq 2(1-2\varepsilon).$$

The rest of the proof consists in upper bounding the left-hand side of this inequality by $20\sqrt{\alpha}$.

Applying the triangle inequality and that $(u+v)^2 \leq 2(u^2+v^2)$ on the inequalities of Lemmas 4.1.7 and 4.1.8 gives

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,0), \Pi(x[1,l-1]1X[l+1,n],l,1)) \leq 68\alpha.$$

We then apply the cut-and-paste property by considering a different protocol where (A, C) is a single player and sends the message $\Pi_{A,C}$. Such a protocol would clearly not be the best for its input (otherwise we would have considered a 2-player protocol from the start), but it can be easily created from P and applying the property to it gives:

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,1), \Pi(x[1,l-1]1X[l+1,n],l,0)) \leq 68\alpha.$$

Combining again with the inequality from Lemma 4.1.8 gives

$$\mathbb{E}_{x[1,l-1],l} \mathrm{h}^2(\Pi(x[1,l-1]0X[l+1,n],l,0), \Pi(x[1,l-1]1X[l+1,n],l,0)) \leq 80\alpha.$$

Last, we use the connection between the Hellinger distance and the ℓ_1-distance from Fact 2.4.15 and the convexity of the square function to conclude with:

$$\mathbb{E}_{x[1,l-1],l} \|\Pi(x[1,l-1]0X[l+1,n],l,0), \Pi(x[1,l-1]1X[l+1,n],l,0)\|_1 \leq \sqrt{160\sqrt{2}\alpha} < 20\sqrt{\alpha}.$$

□

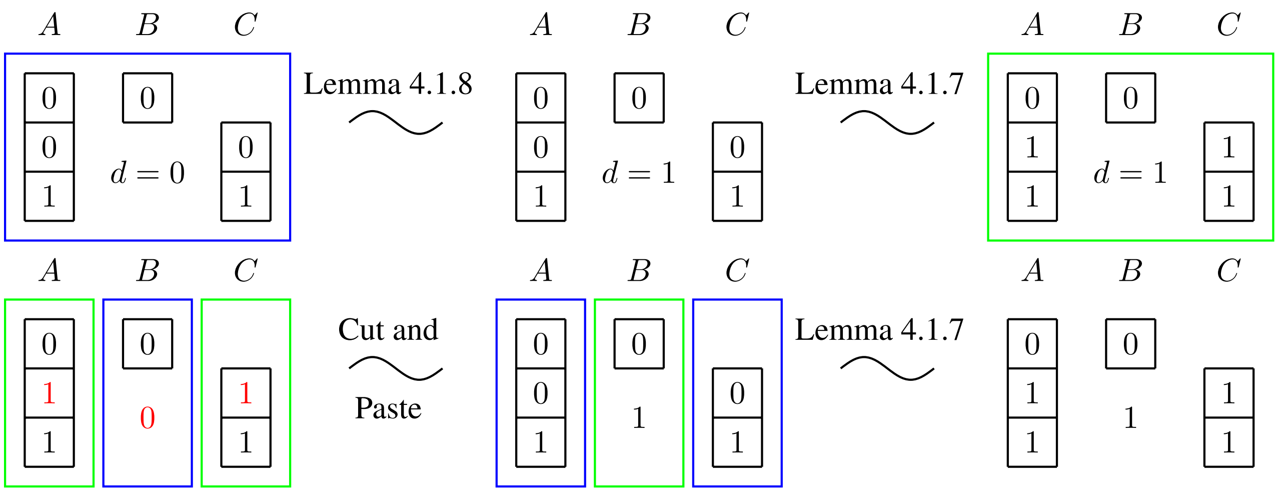


Figure 4.4: Illustration of the proof of Lemma 4.1.9. The green and blue borders show how the cut-and-paste property is applied. In the end, the lower-right instance and the lower-left instance should have close transcripts, although one verifies $f(x, k, d) = 0$ and the other $f(x, k, d) = 1$, and Bob's input is the same in both cases. This creates a contradiction.

4.1.3 Final proof of Theorem 4.1.3

We are now ready to combine all previous lemmas to prove that recognizing PQ-TS with a streaming algorithm is not easier than recognizing PQ.

Proof of Theorem 4.1.3. Let n, N be positive integers such that $N = (2n + 2)n$. Assume that there exists a p-pass randomized streaming algorithm that recognizes PQ-TS$(3N/2)$, with memory space αn and with bounded error ε, for inputs of size N. Then, by Lemma 4.1.4, there is a p-pass randomized streaming algorithm that decides whether an instance of $\textsc{Raindrops}(n, n)$ is in PQ$(3n^2)$ = PQ$(3N/2)$ using memory space $\alpha n + \beta \log n$ and with bounded error ε for some constant β. For n large enough, this is smaller than $2\alpha n$. Lemma 4.1.5 then implies that there exists a p-round randomized protocol P for $\textsc{WeakIndex}(n, n)$ such that each message has size at most $2\alpha n$. By Lemma 4.1.6, one can derive from P another $(p + 1)$-round randomized protocol P' for $\textsc{WeakIndex}(n, 1)$ with bounded error ε, and transcript Π' satisfying $|\Pi'| \leq 6(t + 1)\alpha n$ and $\max \{\mathrm{I}(D : \Pi'_B|X, K), \mathrm{I}(K, D : \Pi'_C|X)\} \leq 2(p + 1)\alpha$. Then by Lemma 4.1.9, $6(p + 1)\alpha \geq (1 - 2\varepsilon)^2/100$, that is $\alpha = \Omega(1/p)$, concluding the proof. □

4.2 Reversing an Input stream in the Input/Output model

In this section we consider a slightly different model from Section 3.3: The algorithm processes a stream containing the input u from left to right, and most copy u on an output stream it processes from right to left. The reason we consider this model with streams in two directions instead of the equivalent model where both streams are read from left to right and the goal is to output $\mathrm{Reverse}(u)$ is that this allows for a greater simplicity in notations. For example, we can refer clearly to an index at which the heads meet.

We call X the input stream and Y the output stream, and also use those names to describe their content. Since they can change over time, we will often refer to X^t and Y^t the contents after pass t. To avoid confusion, the original input (which is equal to X^0) will be called u, as in other sections.

4.2.1 With Read-Only Input and Burn-Only Output

Unsurprisingly, the naive algorithm in this setting is optimal, although the proof is far from trivial. More surprisingly, we were unable to prove it in the general Read-Only/Write-Only model and had to use the more restrictive model where the output stream is Burn-Only.

Theorem 4.2.1. *Let $0 \leq \varepsilon \leq 1/10$. Every p-pass randomized RO/BO algorithm with the input stream read from left to right and the output stream written on from right to left for copying the input u with error ε requires space $\Omega(n/p)$.*

To prove this theorem, we had to develop new techniques. Indeed the normal technique using a reduction from some communication problem cannot work. If we tried something of the sort, logically each player would have a part of u, and would write a part of the output v that is different, and we could try to show that if we want $v = u$ then all players have to send their input to another player. However, that would not imply the theorem, as a streaming algorithm with two streams cannot be simulated by a communication protocol. Consider the situation illustrated in Figure 4.5. Here, Alice is able to write directly on Bob's part of the output without using the algorithm's memory. This sort of trick is also the reason why there are more efficient algorithms than the naive ones in the Read-Only/Read-Write and Read-Write/Write-Only models (see Theorems 3.3.1 and 3.3.2).

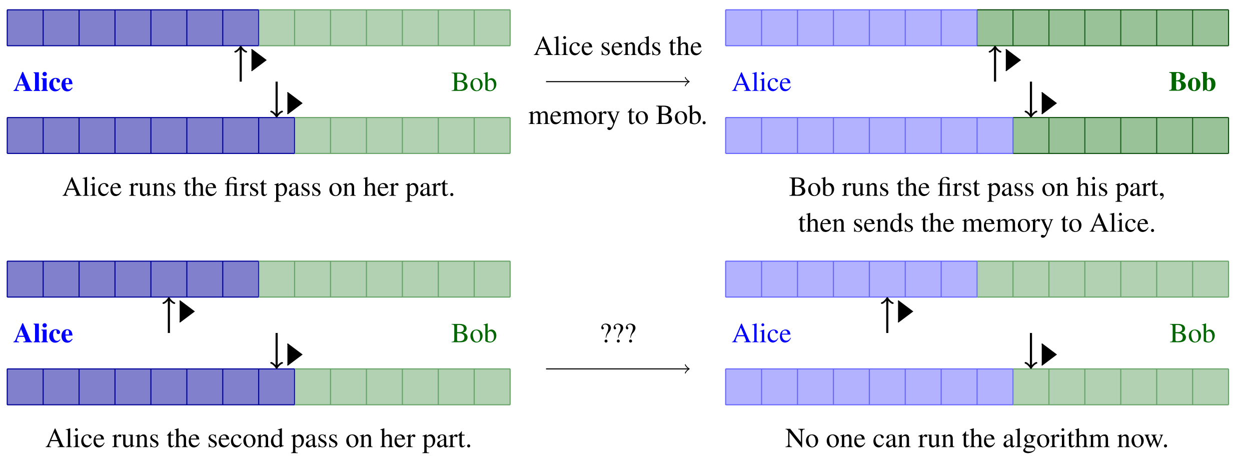

Alice
Bob
Alice sends the memory to Bob.
Alice
Bob
Alice runs the first pass on her part.
Bob runs the first pass on his part, then sends the memory to Alice.
Alice
Bob
???
Alice
Bob
Alice runs the second pass on her part.
No one can run the algorithm now.

Figure 4.5: A streaming algorithm with two streams and multiple passes cannot be simulated by a communication protocol. The first pass determines where the cut-off for the second stream is in relation to the cut-off for the first stream. Then in the second pass, the algorithm can be processing Alice's part of the first stream and Bob's part of the second stream. Information can transit this way between the two players without using the algorithm's memory, i.e. without communication.

Instead, we directly apply information theory tools. If the input is a uniformly distributed random variable X, and Z^t is what is written at pass t, we will want to show that for most i the mutual information between $X[i]$ and $Z^t[i]$ is small. But if that mutual information is small, then the algorithm has a high probability of either not writing anything at index i on the output stream, or writing something wrong. We can then conclude by using the fact that the output stream is Burn-Only, and therefore that if at any pass it contains a wrong symbol, then the output will be wrong.

In the proof below, we actually use the entropy of $X[i]$ conditioned on $Z^t[i]$ instead of mutual information. This is because we need to condition on some other variable that affects the entropy of $X[i]$: otherwise $\mathrm{H}(X[i])$ would just be one and using the entropy or the mutual information would be equivalent.

Proof of Theorem 4.2.1. Consider a deterministic algorithm which is correct on a fraction at least $1-\varepsilon$ of the inputs, and satisfying the other constraints in the theorem. By Yao's principle, if there exists a randomized algorithm with error probability at most ε on every input, then there exists a deterministic algorithm which is correct on a fraction at least $1-\varepsilon$ of the inputs. Assume without loss of generality $s \geq \log n$ (we will show that s is much larger anyway). For simplicity, we assume that passes are synchronized : whenever a pass on one stream ends, the head on the other stream ends its own pass, and then eventually moves back to its original position. This costs us at most a factor 2 in p.

Let the input stream X be uniformly distributed in $\{0,1\}^n$. For each pass $1 \leq t \leq p$, let $Z^t \in \{0,1,\perp\}^n$ be the data written on the output stream Y: if nothing is written at pass t and index i, then $Z^t[i] = \perp$. Note that because the algorithm cannot overwrite a letter, for each i there is at most one t such that $Z^t[i] \neq \perp$. Last, let L^t be the index where the reading head and the writing head meet during pass t. Since passes are synchronized, L^t is unique (but possibly depends on the input and random choices). For $1 \leq i \leq n$, $1 \leq t \leq p$, let M_i^t be the state of the memory after the algorithm reads $X[i]$ on pass t.

For $1 \leq t \leq p$, since s bounds the size of memory, we have :

$$s \geq \mathrm{I}(X[1,L^t] : M^t_{L^t} | X[L^t+1,n])) \text{ and } s \geq \mathrm{I}(X[L^t+1,n] : M_n^{t+1} | X[1,L^t])).$$

Using the definition of mutual information, we get the following inequalities:

$$s \geq \mathrm{H}(X[1,L^t]|X[L^t+1,n]) - \mathrm{H}(X[1,L^t]|M^t_{L^t}, X[L^t+1,n]),$$

$$s \geq \mathrm{H}(X[L^t+1,n]|X[1,L^t]) - \mathrm{H}(X[L^t+1,n]|M_n^{t-1}, X[1,L^t]).$$

We define the following probabilities : $q_i^t(l) = \Pr(Z^t[i] \neq \perp | L^t = l)$, $q_i^t = \Pr(Z^t[i] \neq \perp)$, $\varepsilon_i^t(l) = \Pr(Z^t[i] \neq \perp, Z^t[i] \neq X[i] | L^t = l)$ and $\varepsilon_i^t = \Pr(Z^t[i] \neq \perp, Z^t[i] \neq X[i])$. By definition, they also satisfy $\varepsilon_i^t = \mathbb{E}_{l\sim L^t}(\varepsilon_i^t(l))$ and $q_i^t = \mathbb{E}_{l\sim L^t}(q_i^t(l))$. Because there is no rewriting, the events $Z^t[i] \neq \perp$ for $1 \leq t \leq p$ are mutually exclusive. Therefore by hypothesis[1], for all i, $\sum_{t=1}^p \varepsilon_i^t = \Pr(\exists t, Z^t[i] \neq \perp, Z^t[i] \neq X[i]) \leq \varepsilon$. Lemmas 4.2.2 and 4.2.3 give us these inequalities:

$$2s \geq n - \sum_{i=1}^n \mathrm{H}(X[i]|Z^t[i], L^t) - O(\log n),$$

$$\mathrm{H}(X[i]|Z^t[i], L^t) \leq 1 - q_i^t \left(1 - \mathrm{H}(\varepsilon_i^t/q_i^t)\right).$$

Combining them yields :

$$2s \geq \sum_{i=1}^n q_i^t \left(1 - \mathrm{H}\left((\varepsilon_i^t/q_i^t)\right)\right) - O(\log n).$$

[1]Note that we only need the hypothesis that each bit of Y is wrong with probability at most ε, and not the stronger hypothesis that $Y \neq X$ with probability at most ε.

Let $\alpha_i = \sum_{t=1}^{p} q_i^t$. Then $\alpha_i = \Pr[Y[i] \neq \perp]$ satisfies $\alpha_i \geq 1 - \varepsilon$ by hypothesis. Now summing over all passes leads to $2ps \geq \sum_{i=1}^{n} \alpha_i \sum_{t=1}^{p} (q_i^t/\alpha_i)(1 - \mathrm{H}(\varepsilon_i^t/q_i^t)) - \mathrm{O}(p \log n)$.

The concavity of H gives us $\sum_{t=1}^{p} (q_i^t/\alpha_i)\mathrm{H}(\varepsilon_i^t/q_i^t) \leq \mathrm{H}(\varepsilon/(1-\varepsilon))$. This means, replacing α_i and ε_i^t by their upper bounds, that $2ps \geq n(1-\varepsilon)(1 - \mathrm{H}(\varepsilon/(1-\varepsilon))) - \mathrm{O}(p \log n)$. Since Theorem 4.2.1 has $\varepsilon \leq 0.1$ as an hypothesis, our algorithm verifies $ps \geq \Omega(n)$. □

Lemma 4.2.2. *For any given pass t,*

$$2s \geq n - \sum_{i=1}^{n} \mathrm{H}(X[i]|Z^t[i], L) - O(\log n).$$

Proof. In this proof, we write $Z[i]$ for $Z^t[i]$ since there is generally no ambiguity. We similarly omit the t on other notations. The data processing inequality (Fact 2.4.6) gives us the following inequality : $\mathrm{H}(X[1,L]|M_L, X[L+1,n], L) \leq \mathrm{H}(X[1,L]|Z[1,L], L)$. We can rewrite it as

$$\mathrm{H}(X[1,L]|M_L, X[L+1,n], L) \leq \mathbb{E}_{l\sim L}(\mathrm{H}(X[1,l]|Z[1,l], L=l)).$$

Using the chain rule and removing conditioning, we get

$$\mathrm{H}(X[1,L]|M_L, X[L+1,n], L) \leq \mathbb{E}_{l\sim L}(\sum_{i=1}^{l} \mathrm{H}(X[i]|Z[i], L=l)).$$

Similarly,

$$\mathrm{H}(X[L+1,n]|M_n^{t-1}, X[1,L], L) \leq \mathbb{E}_{l\sim L}(\sum_{i=1}^{l} \mathrm{H}(X[i]|Z[i], L=l)).$$

Using that both M_L (where $M_L = M_{L^t}^t$) and M_n^{t-1} are of size at most s bits, we get

$$2s \geq \mathrm{I}(X[1,L] : M_{L^t}^t | X[L+1,n]) + \mathrm{I}(X[L+1,n] : M_n^{t-1}|X[1,L]).$$

Then we conclude by combining the above inequalities and using Fact 2.4.9 as follows:

$$\begin{aligned}
2s &\geq \mathbb{E}(L) - \mathrm{H}(L) + n - \mathbb{E}(L) - \mathrm{H}(L) \\
&\quad -\mathrm{H}(X[1,L]|M_L^t, X[L+1,n]) - \mathrm{H}(X[L+1,n]|M_n^{t-1}, X[1,L]) \\
&\geq n - \mathbb{E}_{l\sim L}\left(\sum_{i=1}^{l} \mathrm{H}(X[i]|Z[i], L=l) + \sum_{i=l+1}^{n} \mathrm{H}(X[i]|Z[i], L=l)\right) - \mathrm{O}(\log n) \\
&= n - \sum_{i=1}^{n} \mathrm{H}(X[i]|Z[i], L) - O(\log n).
\end{aligned}$$

□

Lemma 4.2.3. *For any pass t,* $\mathrm{H}(X[i]|Z^t[i], L_t) \leq 1 - q_i^t(1 - \mathrm{H}(\varepsilon_i^t/q_i^t))$

Proof. As above, we omit the t in the proof, as there is no ambiguity.

The statement has some similarities with Fano's inequality. Due to the specificities of our context, we have to revisit its proof as follows. First we write

$$\begin{aligned}
\mathrm{H}(X[i]|Z[i], L) &\leq \mathbb{E}_{l\sim L}(\mathrm{H}(X[i]|Z[i], L = l)) \\
&\leq \mathbb{E}_{l\sim L}(q_i(l)\mathrm{H}(X[i]|Z[i], L = l, Z[i] \neq \perp) \\
&\qquad +(1 - q_i(l))\mathrm{H}(X[i]|L = l, Z[i] = \perp) \\
&\leq \mathbb{E}_{l\sim L}\left(q_i(l)\mathrm{H}\left(\frac{\varepsilon_i(l)}{q_i(l)}\right) + 1 - q_i(l)\right) \\
&= 1 - q_i + \mathbb{E}_{l\sim L}\left(q_i(l)\mathrm{H}\left(\frac{\varepsilon_i(l)}{q_i(l)}\right)\right). \qquad (4.4)
\end{aligned}$$

By replacing the entropy with its definition, we can see that for any $1 \geq q \geq \varepsilon > 0$, we have $q\mathrm{H}\left(\frac{\varepsilon}{q}\right) = \mathrm{H}(q - \varepsilon, \varepsilon, 1 - q) - \mathrm{H}(q)$, where $\mathrm{H}(x, y, z)$ is the entropy of a random variable R in $\{0, 1, 2\}$ with $\Pr(R = 0) = x$, $\Pr(R = 1) = y$ and $\Pr(R = 2) = z$. Let R_i be such that $R_i = 0$ if $X[i] = Z[i]$, $R_i = 2$ if $Z[i] = \perp$ and $R_i = 1$ otherwise. Note that $(Z[i] = \perp)$ is a function of R_i. Therefore:

$$\begin{aligned}
\mathbb{E}_{l\sim L}\left(q_i(l)\mathrm{H}\left(\frac{\varepsilon_i(l)}{q_i(l)}\right)\right) &= \mathbb{E}_{l\sim L}(\mathrm{H}(R_i|L = l) - \mathrm{H}((Z[i] = \perp)|L = l)) \\
&= \mathrm{H}(R_i|L) - \mathrm{H}((Z[i] = \perp)|L) \\
&= \mathrm{H}(R_i) - \mathrm{H}((Z[i] = \perp)) + \mathrm{I}((Z[i] = \perp)|L) - \mathrm{I}(R_i|L).
\end{aligned}$$

By the data processing inequality, $\mathrm{I}((Z[i] = \perp) : L) \leq \mathrm{I}(R_i : L)$, so

$$\mathbb{E}_{l\sim L}(q_i(l)\mathrm{H}(\varepsilon_i(l)/q_i(l)))) \leq q_i\mathrm{H}(\varepsilon_i/q_i)).$$

Combining this with inequality 4.4 gives us the lemma. □

4.2.2 With either Read-Write Input or Read-Write Output

We first prove the lower bound in the situation where the input stream is Read-Only and the output stream Read-Write. For this, we employ techniques similar to the ones we used in the proof of Theorem 4.2.1. However, here we will not consider individual cells on the stream but instead blocks of size $k = \sqrt{ns}$, where s is the memory space. we call the blocks content X_i (for the input stream) and Y_i (for the output stream). This allows us to bound the amount of information each block receives during each pass, i.e. the difference in the mutual information between X_i and Y_i at pass $t - 1$ and at pass t. Indeed, during each pass, the heads will cross in some block i. In this block, Y_i may potentially receive all the content of X_i (although not in the correct order), as shown on the left side of Figure 4.6. However, for every $j \neq i$, all information that Y_j received about X_j must either already have been in Y (but in another block), and have been stored in memory since that block of Y was last processed, or come directly from X_j and have been stored in memory while the heads crossed in block i, as shown on the right side of Figure 4.6. This lets us bound the information Y_j gained about X_j during pass t with s. We then obtain our lower bound on s by summing over all blocks and all passes.

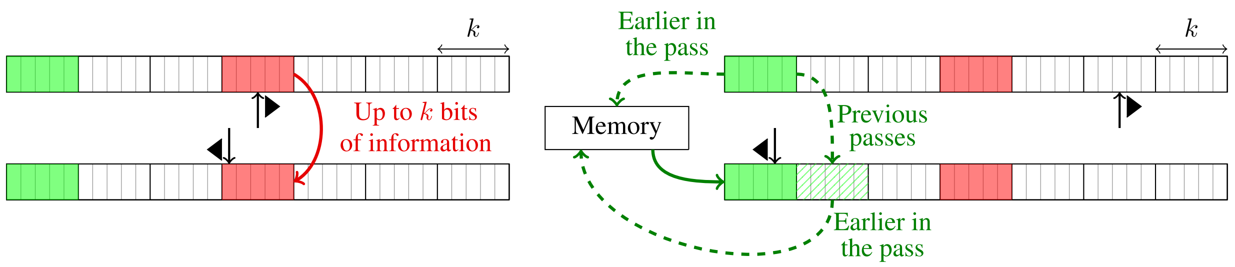


Figure 4.6: During each pass of a RO/RW algorithm, a lot of information can be exchanged at the block where the heads meet (red block), but for all other blocks (green block) the total information gained in Y_j about X_j can be bounded by s the size of the memory.

Theorem 4.2.4. *Let $0 < \varepsilon \leq 1/3$. Every p-pass randomized λ-expansion RO/RW algorithm with the input stream read from left to right and the output stream processed from right to left for copying the input u with error ε requires space $\Omega(n/p^2)$.*

Proof. Consider a deterministic algorithm which is correct on a fraction at least $1 - \varepsilon$ of the inputs, and satisfying the other constraints in the theorem. Assume without loss of generality $s \geq \log n$. As in the proof of Theorem 4.2.1, we assume passes are synchronized at the cost of a factor at most 2 in p. We also keep similar notations : X is the input stream uniformly distributed in $\{0,1\}^n$, and for each $1 \leq t \leq p$, $Y^t \in \{0,1,\perp\}^n$ is the data currently on output stream Y at pass t. Unlike with Z^t in the previous section, this includes the data previously written, as in this model we can read it and modify it. Let $1 \leq k \leq n$ be some parameter. We now think on X, Y^t as sequences of k blocks of size n/k, and consider each block as a symbol. We call X_i the i-th block of X. If $\lambda > 1$, then everything written on the output stream (the only one that can grow) after the n-th bit is considered to be part of the Y_k^t. We write X_{-i} for X without its i-th block, and $X_{>i}$ (resp. $X_{<i}$) for the last $(k-i)$ blocks of X (resp. the first $(i-1)$ blocks). For each $1 \leq t \leq p$, let $L_t \in \{0,\dots,k-1\}$ be the block where the input head and the output head meet during the t-th pass. Since passes are synchronized, L_t is unique and is the only block where both heads can be simultaneously during the t-th pass. Let M_i^t be the memory state as the output head enters the i-th block during t-th pass.

Consider a pass t and a block i. We would like to have an upper bound on the amount of mutual information between Y_i^t and X_i that is gained during pass t (with regards to information known at pass $t-1$). Let $\Delta_{i,j}^t = \mathrm{I}(X_i : Y_i^t | L_t = j, X_{-i}) - \mathrm{I}(X_i : Y_i^{t-1} | L_t = j, X_{-i})$ for some i and j. Of course, if $i = j$, without looking inside the block structure we only have the trivial bound $\Delta_{i,i}^t \leq H(X_i) = n/k$. It is however easier to bound other blocks. Assume without loss of generality that $i < j$. We use the data processing inequality $I(f(A) : B|C) \leq I(A : B|C)$ with Y_i^t as a function of M_i^t and Y_i^{t-1}. This gives us

$$\Delta_{i,j}^t \leq \mathrm{I}(X_i : X_{>i}, M_i^t, Y_i^{t-1} | L_t = j, X_{-i}) - \mathrm{I}(X_i : Y_i^{t-1} | L_t = j, X_{-i}).$$

We can remove $X_{>i}$ which is contained in the conditioning. Applying the chain rule, we cancel out the second term and are left with

$$\Delta_{i,j}^t \leq \mathrm{I}(X_i : M_i^t | L_t = j, X_{-i}, Y_i^{t-1}) \leq \mathrm{H}(M_i^t) \leq s.$$

The same holds if $j < i$ instead. If $j = i$, then $\Delta_{i,j}^t \leq n/k$ because $\mathrm{H}(X_i) = n/k$.

We fix j, sum over i and get $\sum_{i=0}^{k-1} \Delta_{i,j}^t \leq n/k + ks$. The expectation over $j \sim L_t$ is

$$\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^t | L_t, X_{-i}) - \mathrm{I}(X_i : Y_i^{t-1} | L_t, X_{-i}) \leq n/k + ks.$$

From Fact 2.4.8, we get the following inequalities:

$$\mathrm{I}(X_i : Y_i^t | L_t, X_{-i}) \geq \mathrm{I}(X_i : Y_i^t | X_{-i}) - \mathrm{H}(L_t),$$
$$\mathrm{I}(X_i : Y_i^{t-1} | L_t, X_{-i}) \leq \mathrm{I}(X_i : Y_i^{t-1} | X_{-i}) + \mathrm{H}(L_t).$$

Therefore,

$$\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^t | X_{-i}) - \mathrm{I}(X_i : Y_i^{t-1} | X_{-i}) \leq n/k + ks + k \log k.$$

Summing over t yields

$$\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^p | X_{-i}) \leq p(n/k + ks + k \log k).$$

By hypothesis $s \geq \log n$, $\varepsilon \leq 1/3$ and $\mathrm{I}(X_i : Y_i^p | X_{-i}) \geq (1 - \mathrm{H}(\varepsilon))n/k$. If $k = \sqrt{n/s}$, it follows that $s = \Omega(n/p^2)$. □

We now prove a similar theorem for the Read-Write/Read-Only model. The proof is very similar to the one of Theorem 4.2.4, as we consider again the information gained in Y_j about X_j during pass t. The main difference is what happens when j is different from i the block where the heads meet: here it is possible to copy on Y_j the entirety of block X'_j, which could conceivable contain the entire content of X_j, as shown in Figure 4.7. However, we also bound the information X'_j can have gained on X_j based on the number of passes, which lets us as before sum over all passes and all blocks to obtain the lower bound.

Theorem 4.2.5. *Let $0 < \varepsilon \leq 1/3$. Every p-pass randomized λ-expansion RW/WO algorithm with the input stream read from left to right and the output stream processed from right to left for copying the input u with error ε requires space $\Omega(n/p^2)$.*

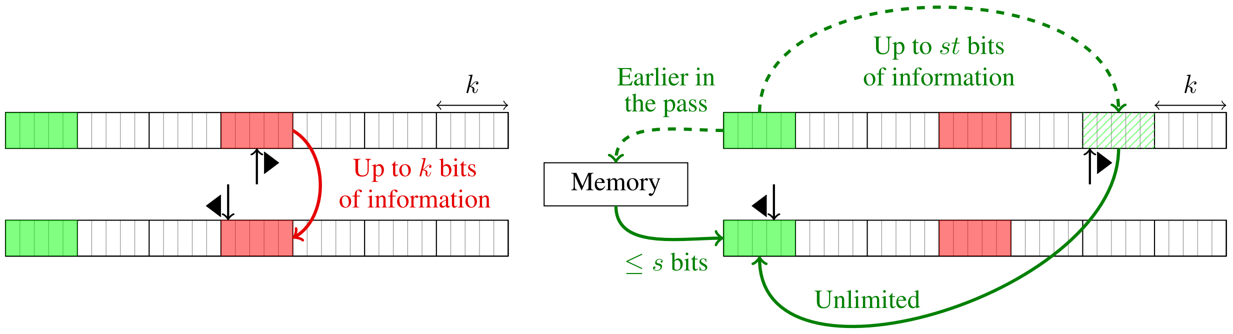


Figure 4.7: As with an RO/RW algorithm, during each pass of an RW/WO algorithm, a lot of information can be exchanged at the block where the heads meet (red block). For all other blocks however (green block), Y_j could also gain a lot of information from another block on the input stream X'_j (hatched green block). We show that X'_j does not contain too much information on X_j based on the number of passes already done.

Proof. Consider a deterministic algorithm which is correct on a fraction at least of the inputs, and satisfying the other constraints in the theorem. Assume without loss of generality $s \geq \log n$. As before we assume passes are synchronized at the cost of a factor at most 2 in p. We also keep similar notations : X is the input stream uniformly distributed in $\{0,1\}^n$, and for each $1 \leq t \leq p$, $X^t \in \{0,1\}^n$ is the content of the input stream and $Y^t \in \{0,1,\perp\}^n$ is the content of the output stream Y at pass t. Let $1 \leq k \leq n$ be some parameter. As before, we think of X, X^t, Y^t as sequences of k blocks of size n/k, and X_i denotes the i-th block of X. If $\lambda > 1$, then everything written on the input stream after the n-th bit is considered part of X_k^t. We write X_{-i}, $X_{>i}$ and $X_{<i}$ as before. We also define $X_i^{\leq t}$ as the $(t+1)$-uple $(X_i, X_i^1, \ldots, X_i^t)$, i.e. the history of the i-th block until pass t.

As in previous proofs, for each $1 \leq t \leq p$, let $L_t \in \{0, \ldots, k-1\}$ be the block where the input head and the output head meet during the t-th pass. L_t is unique and is the only block where both heads can be simultaneously during the t-th pass. Let M_i^t be the memory state as the output head enters the i-th block during t-th pass.

Consider a pass t and a block i. As with Theorem 4.2.4, we would like to have an upper bound on the amount of mutual information between Y_i^t and X_i that is gained during pass t, assuming $L_t \neq i$. Let $\Delta_{i,j}^t = \mathrm{I}(X_i : Y_i^t | L_t = j, X_{-i}^{\leq t-1}) - \mathrm{I}(X_i : Y_i^{t-1} | L_t = j, X_{-i}^{\leq t-1})$ for some $j > i$. By the data processing inequality, we have $I(f(A) : B|C) \leq I(A : B|C)$. Therefore,

$$\Delta_{i,j}^t \leq \mathrm{I}(X_i : X_{>i}^{t-1}, M_i^t, Y_i^{t-1} | L_t = j, X_{-i}^{\leq t-1}) - \mathrm{I}(X_i : Y_i^{t-1} | L_t = j, X_{-i}^{\leq t-1}).$$

We can remove $X_{>i}^{t-1}$ which is contained in the conditioning. Applying the chain rule, we cancel out the second term and are left with

$$\Delta_{i,j}^t \leq \mathrm{I}(X_i : M_i^t | L_t = j, X_{-i}^{\leq t-1}, Y_i^{t-1}) \leq \mathrm{H}(M_i^t) \leq s.$$

The same holds if $j < i$ instead. If $j = i$, then $\Delta_{i,j}^t \leq n/k$ because $\mathrm{H}(X_i) = n/k$.

We fix j, sum over i and get $\sum_{i=0}^{k-1} \Delta_{i,j}^t \leq n/k + ks$. As before, by taking the expectation over $j \sim L_t$ and then using Fact 2.4.8, we can remove the condition $L_t = j$. This gives us

$$\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^t | X_{-i}^{\leq t-1}) - \mathrm{I}(X_i : Y_i^{t-1} | X_{-i}^{\leq t-1}) \leq n/k + ks + k \log k.$$

We cannot sum over t yet because the conditioning $X_{-i}^{\leq t-1}$ depends on t. However, because X_{-i}^t is a function of X_{-i}^{t-1}, the memory state at the beginning of the pass and the memory as the head on the input tape leaves the i-th block, applying Fact 2.4.8 again yields

$$\mathrm{I}(X_i : Y_i^t | X_{-i}^{\leq t}) - \mathrm{I}(X_i : Y_i^t | X_{-i}^{\leq t-1}) \leq \mathrm{H}(X_{-i}^t | X_{-i}^{\leq t-1}) \leq 2s.$$

This is a consequence of the output stream being write-only, which we had not used until now.

Therefore $\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^t | X_{-i}^{\leq t-1}) - \mathrm{I}(X_i : Y_i^{t-1} | X_{-i}^{\leq t-1}) \leq n/k + 3ks + k \log k$. Summing over t yields

$$\sum_{i=0}^{k-1} \mathrm{I}(X_i : Y_i^p | X_{-i}) \leq p(n/k + 3ks + k \log k).$$

By hypothesis $s \geq \log n$, $\varepsilon \leq 1/3$ and $\mathrm{I}(X_i : Y_i^p | X_{-i}) \geq (1 - \mathrm{H}(\varepsilon))n/k$. If $k = \sqrt{n/s}$, it follows that $s = \Omega(n/p^2)$. □

Chapter 5

Perspectives

This work does not pretend to be an exhaustive study of the domain of streaming algorithms. Even within the models consider, several problems remain open.

Language recognition with multiple streams

In Sections 3.3 and 4.2, we considered the model of an algorithm receiving the input on one stream and writing its output on the other stream. While several algorithms presented in Section 3.3 use the input stream as a working space as well, the goal is still to compute some function ($\mathrm{Reverse}$ or Sort), the size of which is equal to the input size.

An alternative model to consider would be one where the algorithms simply outputs YES or NO, such as in the case of language recognition. For example, one can combine Algorithm 3.13 (from Section 3.3.1.2) and Algorithms 3.4 and 3.5 (from Section 3.1) to obtain an algorithm on two Read-Write streams capable of recognizing PQ(n) in $\mathrm{O}(\log n)$ unidirectional passes using $\mathrm{O}(\log^2 n)$ memory[1]. The question is whether we can do better than this, or achieve the same complexity using a more restrictive model (such as only one of the streams being Read-Write). Because of the lower bounds we have, we could not achieve this by reversing the input stream and using Algorithm 3.5.

Another question deals with the difference between the RO/RW and RW/WO models. While Algorithms 3.11 and 3.12 in those two models have the same complexity, Algorithm 3.12 changes the input in a way that does not appear to be easily reversible. This means it loses the option of comparing the input u and $\mathrm{Reverse}(u)$ while still being able to read the entirety of u. It would be interesting to find a problem where this difficulty is relevant, and try to prove a gap between the two models for that problem.

Trade-off between expansion and randomness

In Section 3.3.2 we give two algorithms for sorting the input stream with two Read-Write streams. They have similar complexities, but the first one is deterministic and has potentially $\Omega(\log n)$ expansion, while the second one is a randomized Las Vegas algorithm. Proving that it is not possible to deterministically sort with no expansion and two streams in $\mathrm{O}(\log n)$ passes and $\mathrm{O}(\log n)$ memory would show an interesting trade-off. It would also be interesting to find problems exhibiting a bigger gap.

Improvements on streaming property testing

In Section 3.2, we presented a streaming property tester for visibly pushdown languages under the edit distance. A possible improvement would be to try to prove the same result under the Hamming distance. We know that the tester for regular languages from [35] that we use as a subroutine can work for the Hamming

[1]Similarly, we can recognize DYCK[2] or even DISJOINTNESS with the same complexity.

distance, however our method of suffix sampling would no longer be adapted as we can no longer simply ignore a prefix of unknown size.

Another question is whether the result extends beyond visibly push-down languages, for example to deterministic context-free languages. DCFLs are languages recognized by a stack automaton, and unlike with VPLs some symbols may cause the automaton either to push or pop depending on the state. Our algorithm makes extensive use of the fact that each letter is either a push, pop, or neutral symbol in a VPL, but on the other hand we were unable to find a likely candidate for a DCFL that would be hard to test in streaming, and give us a lower bound. Ideally, we would want the push/pop signification of letters in a later part of the words to completely change based on some property of the first part that is hard to know in advance.

Finally, we believe the complexity of our tester $\mathrm{O}(\log^7 n/\varepsilon^4)$ can be improved. Notably, the instances that give us the biggest problems and are the reason for this high complexity tend to have a lot of small push and pop sequences alternating. But such an instance would have small maximal stack height, potentially $\mathrm{O}(\log n)$, and the question of its membership in the language can be decided exactly using $\mathrm{O}(\log n)$ memory by simply running the stack automaton. If we could find a way to exploit this trade-off, we could potentially have a better complexity in n.

Combining other models

Another subject of interest related to property testing (but not streaming) is to combine the query model with the trial-and-error model. On the toy problem of finding the minimum in a list, the query model and the trial-and-error model (i.e. asking an oracle whether an element is the minimum. If not, the oracle returns a smaller element.) each require time $\Omega(n)$. However, by combining both a randomized algorithm can achieve expected time $\mathrm{O}(\sqrt{n})$ by querying comparisons between $\sqrt{n}$ random elements, and then asking trial-and-error queries starting from the minimum. We have tried to find other problems where the two types of queries together would be significantly more powerful than either alone.

Bibliography

[1] Noga Alon, Michael Krivelevich, Ilan Newman, and Mario Szegedy. “Regular languages are testable with a constant number of queries”. In: *SIAM Journal on Computing* 30.6 (2001), pp. 1842–1862.

[2] Noga Alon, Yossi Matias, and Mario Szegedy. “The Space Complexity of Approximating the Frequency Moments”. In: *Journal of Computer and System Sciences* 58.1 (1999), pp. 137–147.

[3] Rajeev Alur and Parthasarathy Madhusudan. “Adding nesting structure to words”. In: *Journal of the ACM* 56.3 (2009), p. 16.

[4] Rajeev Alur and Parthasarathy Madhusudan. “Visibly pushdown languages”. In: *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing* (2004), pp. 202–211.

[5] Andris Ambainis, Kaspars Balodis, Aleksandrs Belovs, Troy Lee, Miklos Santha, and Juris Smotrovs. “Separations in Query Complexity Based on Pointer Functions”. In: *arXiv preprint arXiv:1506.04719* (2015).

[6] Xiaohui Bei, Ning Chen, and Shengyu Zhang. “On the complexity of trial and error”. In: *Proceedings of the forty-fifth annual ACM symposium on Theory of computing* (2013), pp. 31–40.

[7] Mihir Bellare, Shafi Goldwasser, Carsten Lund, and Alexander Russell. “Efficient probabilistically checkable proofs and applications to approximations”. In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing* (1993), pp. 294–304.

[8] Manuel Blum, Michael Luby, and Ronitt Rubinfeld. “Self-testing/correcting with applications to numerical problems”. In: *Proceedings of the twenty-second annual ACM symposium on Theory of computing* (1990), pp. 73–83.

[9] Clément Canonne and Ronitt Rubinfeld. “Testing probability distributions underlying aggregated data”. In: *Automata, Languages, and Programming* (2014), pp. 283–295.

[10] Amit Chakrabarti, Graham Cormode, Ranganath Kondapally, and Andrew McGregor. “Information cost tradeoffs for augmented index and streaming language recognition”. In: *SIAM Journal on Computing* 42.1 (2013), pp. 61–83.

[11] Jianer Chen and Chee-Keng Yap. “Reversal complexity”. In: *SIAM Journal on Computing* 20.4 (1991), pp. 622–638.

[12] Matthew Chu, Sampath Kannan, and Andrew McGregor. “Checking and spot-checking the correctness of priority queues”. In: *Automata, Languages and Programming* (2007), pp. 728–739.

[13] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[14] Funda Ergün, Sampath Kannan, S Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. “Spot-checkers”. In: *Proceedings of the thirtieth annual ACM symposium on Theory of computing* (1998), pp. 259–268.

[15] Joan Feigenbaum, Sampath Kannan, Martin Strauss, and Mahesh Viswanathan. “Testing and spot-checking of data streams”. In: *Algorithmica* 34.1 (2002), pp. 67–80.

[16] Philippe Flajolet, Jean Françon, and Jean Vuillemin. “Sequence of operations analysis for dynamic data structures”. In: *Journal of Algorithms* 1.2 (1980), pp. 111–141.

[17] Nathanaël François, Rahul Jain, and Frédéric Magniez. “Unidirectional Input/Output Streaming Complexity of Reversal and Sorting”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques* (2014), p. 654.

[18] Nathanaël François and Frédéric Magniez. “Streaming Complexity of Checking Priority Queues”. In: *30th International Symposium on Theoretical Aspects of Computer Science* (2013), p. 454.

[19] Nathanaël François, Frédéric Magniez, Michel de Rougemont, and Olivier Serre. “Streaming Property Testing of Visibly Pushdown Languages”. In: *arXiv preprint arXiv:1505.03334* (2015).

[20] Michael L. Fredman and Robert Endre Tarjan. “Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms”. In: *Journal of the ACM* 34.3 (1987), pp. 596–615.

[21] Peter Gemmell, Richard Lipton, Ronitt Rubinfeld, Madhu Sudan, and Avi Wigderson. “Self-testing/correcting for polynomials and for approximate functions”. In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing* (1991), pp. 32–42.

[22] Oded Goldreich and Dana Ron. “On learning and testing dynamic environments”. In: *Foundations of Computer Science, 2014 IEEE 55th Annual Symposium on* (2014), pp. 336–343.

[23] Oded Goldreich and Dana Ron. “Property testing in bounded degree graphs”. In: *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing* (1997), pp. 406–415.

[24] Martin Grohe, André Hernich, and Nicole Schweikardt. “Lower bounds for processing data with few random accesses to external memory”. In: *Journal of the ACM* 56.3 (2009), p. 12.

[25] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. “Introduction to Automata Theory, Languages, and Computation”. In: (2006).

[26] Rahul Jain and Ashwin Nayak. “The space complexity of recognizing well-parenthesized expressions in the streaming model: the Index function revisited”. In: *arXiv preprint arXiv:1004.3165* (2010).

[27] Sampath Kannan, Claire Mathieu, and Hang Zhou. “Graph Verification and Reconstruction via Distance Oracles”. In: *arXiv preprint arXiv:1402.4037* (2014).

[28] Richard M Karp, Scott Shenker, and Christos H Papadimitriou. “A simple algorithm for finding frequent elements in streams and bags”. In: *ACM Transactions on Database Systems* 28.1 (2003), pp. 51–55.

[29] Philip N Klein and John H Reif. “Parallel Time O(\logn) Acceptance of Deterministic CFLs on an Exclusive-Write P-RAM”. In: *SIAM Journal on Computing* 17.3 (1988), pp. 463–485.

[30] Christian Konrad and Frédéric Magniez. “Validating XML documents in the streaming model with external memory”. In: *ACM Transactions on Database Systems* 38.4 (2013), p. 27.

[31] László Lovász, Moni Naor, Ilan Newman, and Avi Wigderson. “Search problems in the decision tree model”. In: *SIAM Journal on Discrete Mathematics* 8.1 (1995), pp. 119–132.

[32] 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.

[33] Andrew McGregor. “Graph stream algorithms: A survey”. In: *ACM SIGMOD Record* 43.1 (2014), pp. 9–20.

[34] Kurt Mehlhorn. *Pebbling mountain ranges and its application to DCFL-recognition*. Springer, 1980.

[35] Antoine Ndione, Aurélien Lemay, and Joachim Niehren. “Approximate membership for regular languages modulo the edit distance”. In: *Theoretical Computer Science* 487 (2013), pp. 37–49.

[36] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. “Testing membership in parenthesis languages”. In: *Random Structures & Algorithms* 22.1 (2003), pp. 98–138.

[37] Ronald L Rivest and Jean Vuillemin. “On recognizing graph properties from adjacency matrices”. In: *Theoretical Computer Science* 3.3 (1976), pp. 371–384.

[38] Jan Matthias Ruhl. “Efficient algorithms for new computational models”. PhD thesis. Massachusetts Institute of Technology, 2003.

[39] Miklos Santha. “On the Monte carlo boolean decision tree complexity of read-once formulae”. In: *Random Structures & Algorithms* 6.1 (1995), pp. 75–87.

[40] Jean Vuillemin. “A data structure for manipulating priority queues”. In: *Communications of the ACM* 21.4 (1978), pp. 309–315.

[41] Andrew Chi-Chih Yao. “Some complexity questions related to distributive computing (preliminary report)”. In: *Proceedings of the eleventh annual ACM symposium on Theory of computing* (1979), pp. 209–213.