External Memory Suffix Array Construction

Dimplomarbeit

am Fachbereich Informatik an der Universität des Saarlandes

von

Jens Mehnert

November 2004

Diese Arbeit wurde nach einem Thema von Prof. Dr. Peter Sanders am Max-Planck-Institut für Informatik in Saarbrücken angefertigt.

Key words and phrases. Suffix array construction, doubling, discarding, DC3, I/O optimal, external memory, difference covers.

Hiermit erkläre ich an Eides Statt, dass ich diese Diplomarbeit selbständig verfasst und nur die gegebenen Quellen benutzt habe. Ferner habe ich die Arbeit noch keinem anderen Prüfungsamt vorgelegt.

Saarbrücken, November 2004

Danksagung

An dieser Stelle gilt mein Dank Herrn Prof. Dr. rer. nat. Peter Sanders für seine Betreuung und Unterstützung während der Anfertigung dieser Diplomarbeit. Desweiteren danke ich Roman Dementiev (M.Sc.) für seine stetige Unterstützung, Depak Ajwani (M.Sc.) für das Gegenlesen der Arbeit und Herrn Dr. Juha Kärkkäinen für hilfreiche Vorschläge in theoretischen Aspekten.

Deutsche Kurzzusammenfassung

Diese Arbeit befasst sich mit der schnellen externen Konstruktion grosser *Suffix Arrays*. Die Datenstruktur hat ihren Anwendungszweck in der beschleunigten Suche von Mustern in grossen Datensätzen, wie etwa dem menschlichen Genom aber auch in der Datenkompression und vielen anderen Bereichen. Da interner Speicher vergleichsweise teuer ist, findet man in aktuellen Computern oftmals mehr als das Hundertfache an Festplattenspeicher. Wir nutzen diesen Speicher und konstruieren *Suffix Arrays* von bis zu $2^{32} - 1$ zeichen langen Strings, in wenigen Stunden auf einem günstigen Testsystem. Dazu werden unter anderem externe Designs neuer *Suffix Array* Konstruktionsalgorithmen vorgestellt, verbessert, implementiert und analysiert. Einer der verwendeten Algorithmen ist für alle möglich Eingaben asymptotisch optimal. Die wesentlichsten Verbesserungen resultieren aus einer Methode, genannt *pipelining*, welche die Festplattennutzung weiter reduziert und dadurch die Konstruktion beschleunigt.

Contents

Chapter 1. Introduction 7

Chapter 2. Related Work 10

Chapter 3. Pipelining 11
3.1. The Pipelining Structure 11
3.2. The Pipelining Concept In The C++ Programming Language 13

Chapter 4. Theoretical External Memory Models 18

Chapter 5. Definitions And Notation 20
5.1. String Order 20
5.2. Suffix Array 20
5.3. String Names 20
5.4. Trie And Suffix Tree 21
5.5. Longest Common Prefix 21
5.6. Alphabets 22
5.7. Pseudo Code Conventions 22

Chapter 6. Basic Sort Algorithms 23
6.1. The Radix Sort Algorithm 23
6.2. A Simple External Memory Merge Sort Algorithm 23

Chapter 7. Doubling 25
7.1. The Idea Of Doubling 25
7.2. The Optimised Index Comparison Function 26
7.3. A Pipelined Doubling Algorithm 26
7.4. From Pipelined Doubling To Pipelined X-Tupling 28
7.5. Finding The Optimal X For X-Tupling 28

Chapter 8. Discarding 31
8.1. Changing The Naming Strategy 31
8.2. Doubling Combined With Discarding 33
8.3. A Pipelined Discarding Algorithm 33
8.4. From Pipelined Discarding To Pipelined X-Discarding 36
8.5. Finding The Optimal X For X-Discarding 37

Chapter 9. DCX 39
9.1. The Idea Of The DC3 Algorithm 39
9.2. The DC3 Algorithm In Detail 39
9.3. A Pipelined DC3 Algorithm 40

9.4. A Pipelined DC3 Algorithm With Discarding 43
9.5. Thoughts About A Pipelined DCX Algorithm 45

Chapter 10. Suffix Array Checking 48

Chapter 11. Experiments 50
11.1. Our Test System Configuration 50
11.2. Characterising Our Input Data Instances 50
11.3. Results For The Non-Pipelined Doubling Algorithm 52
11.4. Results For The X-Tupling Algorithm 53
11.5. Results For The X-Discarding Algorithm 53
11.6. Results For The DC3 Algorithm 55

Chapter 12. String Preprocessing 56
12.1. Alphabet Increasing By Concatenating Characters 56
12.2. More Efficient Alphabet Increasing By A Base Change 57
12.3. Alphabet Increasing By Using Global Information 57
12.4. Alphabet Increasing By Reducing Repeating Characters 58

Chapter 13. Summary And Conclusion 59

Chapter 14. Future Work 61

Appendix A. Foundations 62
A.1. Symbol Table 62
A.2. Difference And Sum Cover Basics 64

Appendix B. Implementation Issues 65
B.1. Generator Pipeline Nodes 65
B.2. Naming Pipeline Nodes 66
B.3. Comparison Functions 68
B.4. Pipelined Loop Example 70

Appendix C. More About DCX 72
C.1. The DCX I/O Volume 72
C.2. Further Thoughts About Reducing IO Volume 72
C.3. Perfect Difference Covers 74

Appendix. Bibliography 75

CHAPTER 1

Introduction

A *suffix tree* **[31, 26]** is a well known tree data structure used for example for *pattern matching*. The pattern matching problem asks to find all starting positions of a pattern T of size m in a string S of size n. *Suffix Trees* can solve the pattern matching problem in $O(m+k)$ time, where k is the number of occurrences of T in S, and have many other applications. We know from **[11]** that suffix tree construction from a string S over an alphabet Σ is as complex as sorting the characters of S. Hence we can construct a *suffix tree* from a given string S for an *integer alphabet* Σ of size k ($k = n^c$ for a constant c) in $O(n)$ time **[11]** and for a *general alphabet* (5.6) in $O(n \log n)$ time.

The *Suffix array*, first introduced by Udi Manber and Gene Myers **[24]** in 1989, is another data structure which is functionally similar to a *suffix tree* and which can be used to solve the pattern matching problem in $O(m + \log n)$ time complexity. It contains the sorted suffixes of a string S represented by their starting position in S. We can construct a *suffix array* from a *suffix tree* **[11]** in linear time and thus the two structures have the same construction time complexity. A direct *suffix array* construction algorithm does not use a *suffix tree* which usually needs more space than a *suffix array*. Until 2003 it was an open problem if there is an *I/O* optimal direct *suffix array* construction algorithm **[7]**. Earlier direct *suffix array* construction algorithms have a construction time of $O(n \log n)$ even for alphabets with constant size. Examples are the algorithm from Manber and Myers **[24]**, a simple algorithm called *doubling* **[19, 3]** and *doubling combined with discarding* **[7, 1]**. In 2003 three direct *I/O optimal* linear time suffix array construction algorithms **[18, 21, 22]** were published independently. The algorithm introduced in **[18, 5]** is called *DC3* algorithm whose generalisation is connected to an algebraic structure called *difference cover*. For more details about *suffix trees* and *suffix arrays* see *Chapter* 5.

OBJECTIVE

The main objective of this thesis is to adapt the most recent *suffix array* construction algorithms to external memory *suffix array* construction and also optimise them for this purpose. Our algorithms have to construct large *suffix arrays* having only little memory space so that the *suffix array* cannot be constructed internally. The algorithms have to take advantages of multiple disks and they should also be fast in practice. For this purpose, we have designed, improved and implemented external *suffix array* construction algorithms from *DC3*, *doubling* and *doubling combined with discarding*.

For computers nowadays, *internal memory* is expensive with respect to *external memory* of equal size. A typical case for an average computer is that the *external memory* size is more than hundred times larger than the *internal memory* size. If we assume that internal and external *suffix array* construction

algorithms have a similar space complexity then we can handle much larger *suffix arrays* externally. Most of the algorithms, we have implemented, can handle input strings of size $2^{32}-1$ with an alphabet size of $2^{32}-1$. On the other hand a disadvantage of external memory is the slow random disk accesses. Nowadays a $250GB$ disk with the bandwidth $\approx 92MByte/s$ has a write/read seek time of $\approx 9-11ms$ on the average[1]. Hence external memory algorithms should avoid many random disk accesses and should instead write/read data in bigger chunks. Our *external memory* model consists of fast *internal memory* of size M and a block size B (measured in words) which is the unit for one *external memory* access to one disk. Hence we can read or write in one *I/O access* $D \cdot B$ words to the D disks. In our pipelined algorithms only the sorters and the external arrays need disk space for large input data. Thus we hide the *I/O accesses* in the terms $\text{sort}_{I/O}$ and $\text{scan}_{I/O}$ where $\text{sort}_{I/O}$ represents an *external memory* sort algorithm and $\text{scan}_{I/O}$ an *external memory* array. The external merge sort algorithm from [**9**] needs $\text{sort}_{I/O}(N) = O(\frac{N}{D \cdot B} \cdot \log_{M/B} \frac{N}{M})$ *I/O accesses* for sorting N words . Only the DCX algorithm needs external arrays besides the external sorters. The external arrays from [**9**] need about $\text{scan}_{I/O}(N) = O(\frac{N}{D \cdot B})$ *I/O* accesses (shorter *I/Os*) for storing N words to disk. Since we have different block sizes in different implementations, we count the *I/O volume*, which refers to the total number of bytes read or written by our algorithms, to compare our implementations.

3. Pipelining

Besides avoiding random disk accesses, we can also reduce the *I/O volume* of our algorithms with a principle called *pipelining*. In *Chapter* 3 we describe the basic method of *pipelining* which is the main source of improvement for all implemented algorithms in this work. There is also a description of how a programmer can adapt his own algorithms to the pipeline interface in the *C++* programming language. The adapted algorithms can be used with the external memory algorithm library STXXL [**9**]. Interfaces of STXXL are similar to the interfaces of the *Standard Template Library*[2]. STXXL has an implementation of *external merge sort* [**10**] having the mentioned *pipeline interface*.

7. Doubling

In *Chapter* 7 we present the first result of this thesis which is the *pipelined doubling* algorithm. *Doubling* without pipelining has an *I/O volume* which is about three times as large as the I/O volume of *pipelined doubling*. The next logical step was a generalisation of *pipelined doubling* called *X-Tupling* (7.4). The X in *X-Tupling* stands for the logarithm base in the *suffix array* construction time complexity which is for N input characters $O(N \log_X N)$. We need at most $log_X N$ iterations for *X-Tupling* and a larger X increases the *I/O volume* per iteration. Our *X-Tupling* is pipelined. We implemented *X-Tupling* for $X = 4$ since we show it to be a good tradeoff between a minimal *I/O volume* and the resulting *CPU* load. This choice of the parameter X (see *Section* 7.5) reduces the *I/O volume* of *X-Tupling* for the most input instances to about two third times the *pipelined doubling I/O volume*.

8. Discarding

A disadvantage of *X-Tupling* is that the number of iterations which the algorithm needs hinges on a few input string suffixes with the *longest common*

[1] `http://www.wdc.com/en/products/Products.asp?DriveID=42`

[2] `http://www.sgi.com/tech/stl/`

prefix (see *Section* 5.5). *Discarding* which was introduced in [**7**] is able to reduce the *I/O volume* sorted during each iteration. Our pipelined *suffix array* construction algorithm *X-Discarding* is based on this idea and on the improvement in [**1**] which reduces the *I/O volume* with respect to [**7**] because it is able to take sorter items out of the only loop of the algorithm and thus to reduce the *I/O volume* further. We have implemented *X-Discarding* for $X = 4$ for similar reasons as for the *X-Tupling* (see *Section* 8.5) .

9. DCX

The most important result of this thesis is a pipelined version of the *DCX* algorithm [**5**] for the parameter $X = 3$, where X is connected to an algebraical structure called *difference cover*. We have implemented *DC3* for reasons we discuss in *Section* 9.5. We have also designed *DCX* with *discarding* which can be of interest for data with large alphabets.

10. Checker

Since we work with large data, use two different kinds of memory and because humans make mistakes, the correctness of the computed *suffix array* has to be warranted, particularly because todays hard disks are error-prone. For this reason we also implemented a *suffix array checker* (*Chapter* 10) which is based on the results in [**5**] (more precisely their *Theorem 2*). The *I/O* cost of the pipelined version of this checker is so cheap that it only needs a fraction of the time needed for the *suffix array* construction.

11.Experiments

We tested all implemented algorithms with mostly real world data and one difficult synthetic data instance. The result on our test computer is that *DC3* always has the shortest running time which is about $10\mu s$ per input character for all test instances. Even *4-Discarding* which matches the *I/O volume* of *DC3* in the *genome* data instance needs about $20\mu s$ per input character.

In *Chapter* 12 we present methods to increase the alphabet size of strings without additional *I/O cost,* therefore resulting in less work for discarding algorithms. We show how to translate a string S_1 with an alphabet Σ_1 of size m_1 into a string S_2 with an alphabet Σ_2 of size $m_2 > m_1$ which has the same *suffix array*.

CHAPTER 2

Related Work

The first *I/O* optimal non direct suffix array construction algorithm was introduced by M. Farach-Colton and P. Ferragina and S. Muthukrishnan [**13**]. Besides the fact that they use suffix trees they state a part of their algorithm as difficult to implement.

Another suffix array construction algorithm due to Beaza Yates, G. Gonnet and T. Snider is called BGS algorithm [**14, 7**]. The algorithm needs $O(\frac{N}{M} \cdot scan(N))$ *I/Os* in $\Theta(\frac{N}{M})$ scan steps. We have not implemented this algorithm because we assume, that $\frac{N}{M}$ is large. Furthermore we cannot extend the *BGS* algorithm with most of our optimisation strategies. However, for small $\frac{N}{M}$ and for machines with slow *I/O* the algorithm might be a good choice.

Besides the STXXL external memory library we use [**9**], there are other external memory librarys as *LEDA-SM* and *TPIE* [**8, 4**]. Of the mentioned librarys only STXXL supports pipelining.

That there is interest in large suffix arrays, we can conclude from the following works about internal memory suffix array or suffix tree construction [**17, 16, 23, 27, 5**]. On of the objectives of these papers is to reduce the internal space requirement during the suffix array/tree construction, or the compression of the resulting data structures and thus the possibility to compute larger suffix arrays and suffix trees internally. Nevertheless, saving space can also slow down algorithms. If we compare the *DC3* algorithm in the test section with *4-Discarding,* we see that we cannot ignore the internal work requirement of the algorithms. Even in the cases, where the two algorithms have nearly the same *I/O* volume[1], the *DC3* algorithm is about twice as fast due to its smaller internal work requirement.

Other constructions of large suffix arrays from the human genome were done in [**23**] on a PC with $3GB$ of internal memory within 21 hours and in [**23**] on a supercomputer with $64GB$ internal memory within 7 hours [**29**]. In [**28**] they construct a suffix tree from the human genome within 30 hours on a $3GHz$ PC with $2GB$ internal memory.

We construct a suffix array of the human genome in less than 8.5 hours with four disks and one $2GHz$ CPU of our test system.

[1] the number of read and written bytes from/to disk

CHAPTER 3

Pipelining

We have little fast internal memory (RAM), slow external memory (hard disks) and we have a sequence of space consuming algorithms. Each algorithm in this sequence stores data to disk which the next algorithm must read again. This is not necessary because we could combine the sequence of algorithms to one complex algorithm. This complex algorithm might be hard to write, hard to debug and all the sub algorithms hard to reuse.

With pipelining, we connect the algorithms with an interface, so that they can pass-through their data from one to the next algorithm, without storing data in between. Thus the sequence of algorithms is coalesced to one big algorithm. This algorithm is still manageable because an algorithm do not need to know much about the structure of the others.

3.1. The Pipelining Structure

The pipelined algorithm structure, we will discuss here can be used to design more generic and faster algorithms which avoid storing data whenever possible. Pipelining is a well known principle, used in hardware as a pipelined *CPU* design. Our pipelining structure can be seen as a water pipe system without pump where water (data) runs top down the pipes (algorithms) and sometimes needs a water reservoir (algorithms which need hard disks in our case).

Pipelining algorithms can be described as acyclic graphs $G = (V,\ E)$ with degree restricted nodes. We call a connected acyclic graph in this context *pipe*. A pipelined algorithm can consist of more than one *pipe*. The most generic node types are *pipeline nodes* $v_p \in PV \subset V$, *file nodes* $f \in FV \subset V$ and *buffer nodes* $v_b \in BV \subset V$. *Pipeline nodes* can have an arbitrary in and out degree. *File* and *buffer nodes* have either an in degree or out degree of finite many nodes. Only file nodes can access external memory. *Buffer nodes* can sometimes replace *file nodes*, if our space requirement fits into internal memory. With these node types, we can model all needed composed node types in this work. We shortcut pipelined algorithms with new node types.

The internal memory requirement of a single node $v \in V$ can be computed in machine words with a function $b\ :\ V \mapsto \mathbb{N}_0$. The edge cost function $w\ :\ E \mapsto \mathbb{N}_0$ computes the number of machine words flowing through an edge.

An example for a composed node is the pipelined external merge sort[1]. This sorter can be described with two *pipes*, which are also node types. The sorter run formation *pipe* and the sorter merge *pipe*. For the sorter run formation *pipe*, we need to connect a *pipeline node* $p_1 \in PV$ to a *file node* $f_1 \in FV$ with an

[1] See *Section* 6.2.

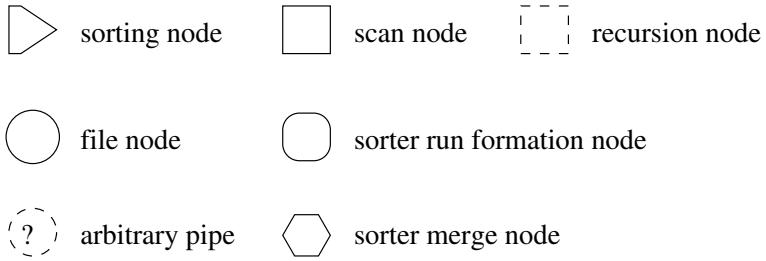

FIGURE 3.1.1. Node types

edge $(p_1,\ f_1) \in E$. The sorter merge pipe consists of a file node f_2 connected to the pipeline node p_2 with the edge $(f_2,\ p_2)$. Assume, that we want to sort a sequence of N words. This is done externally by first sorting $\lceil \frac{N}{M} \rceil$ partial sequences (runs) of size M internally and storing the sorted sequences to disk (sorter run formation). In our example the internal sorting is done in p_1 and the sorted sequence are stored to disk in f_1. It means that $b(p_1) = \Theta(M)$ and $w((p_1,\ f_1)) = N$. The second *pipe* reads the sorted subsequences from the new file node f_2 and merges them to the final sorted sequence in p_2. A more detailed description of an external merge sort algorithm follows in *Section* 6.2. In *Figure* 3.1.1 we have summarised the node types we use to describe the algorithms in this thesis. The sorting node is a shortcut for a sorter run formation node together with sorter merge node. A scan node is a *pipeline node* but in most cases in this work it has in and out degree of one. A recursion node represents the result computed by the pipeline in the recursion call.

Since our *pipes* do not support cycles, we need more than one pipe to model a loop. In *Figure* 3.1.2 we see how to model a loop in our pipelining model with three pipes. What we do is to split the cycle with help of file nodes. Multiple loops through a cycle are simulated by calling the pipeline which represents the loop body multiple times. The sorter run formation node represents the W file node in the *Figure* 3.1.2 and the sorter merge represents the R file node. For the sake of short descriptions, we shortcut the loop flow chart with sorter as shown in *Figure* 3.1.3. In part a) of the figure we see how the loop with sorter is implemented and b) shows the shortcut. The shortcut hides further details, but we can count the I/O costs using this representation. Notice that we cannot avoid the first sorter in the loop, because the data is already stored in the sorter run formation node before we know if we need to sort. For programmers, we give a loop example in the *Appendix* B.4.

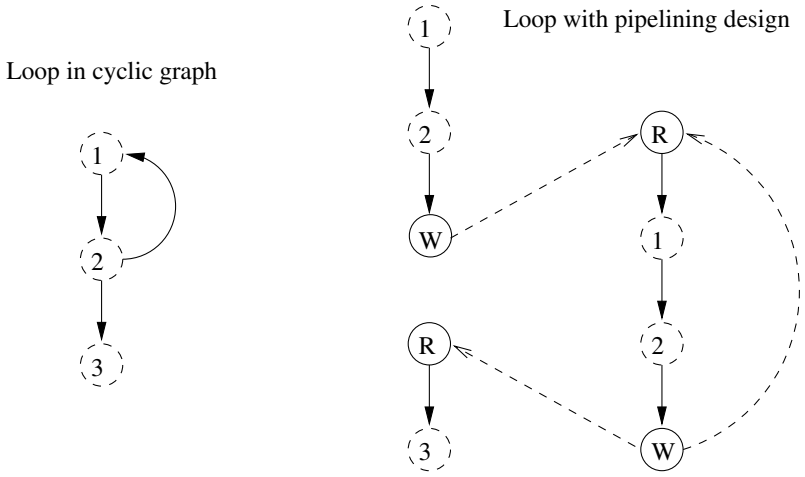

FIGURE 3.1.2. Loop model example with pipelining design

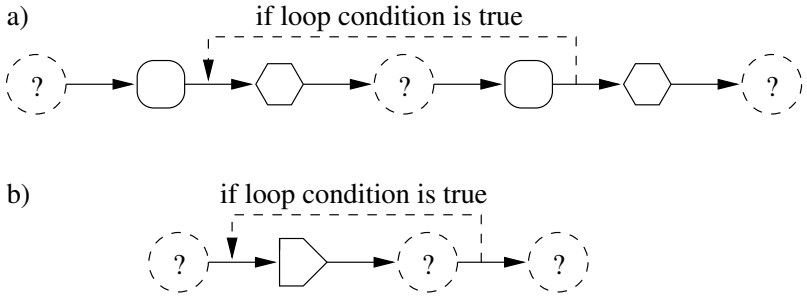

FIGURE 3.1.3. Loop with sorter

3.2. The Pipelining Concept In The C++ Programming Language

In the following we will describe how to write pipelined algorithms which can be combined with the algorithms from the STXXL library from Roman Dementiev **[9, 10]** . A short description of the details:

Description

The pipeline interface or pipe model behaves similar as *Forward Iterators*[2] of the *Standard Template Library* of *C++*. The difference is that an algorithm, which uses the pipeline interface, reads data from finitely many input pipes and computes one result in each step. As for *Forward Iterators* we are not allowed to step back in this sequence of results.

Associated types

Value type	the type returned by dereferencing a pipe

[2]http://www.sgi.com/tech/stl/ForwardIterator.html

Notation

$A,\ I_e$	types which are models of pipe
$A :: value\_type$	the value type of A
a	object of type A
i_e	object of type I_e
m	number of input pipes
$[t]$	a finite set of arbitrary types
$[o]$	finite many objects with arbitrary types

Definitions

A pipe is *past-end*, if the member function $a.empty()$ returns boolean value *true*. A pipe is *deferenceable*, if the value resulting from a dereferenced pipe is well-defined. A pipe is *incrementable*, if $++a$ is well-defined.

Valid expressions

Name	Expression	Return type
Constructor	$A\langle I_0, \ldots, I_m, [t]\rangle\ a(i_0, \ldots, i_m, [o])$	
Dereference	$*a$	*const value_type&*
Preincrement	$++a$	$A\&$
Check *past-end*	a.empty()	*bool*

Expression semantics

Name	Expression	Precondition	Postcondition
Constructor	$A\langle I_0, \ldots, I_m, [t]\rangle\ a(i_0, \ldots, i_m, [o])$		
Dereference	$*a$	a is *incrementable*	
Preincrement	$++a$	a is *dereferenceable*	a is *dereferenceable* or *past-end*
Check *past-end*	a.empty()		

Complexity guarantees

There are none. Nevertheless the operations $*a$ and a.empty() can have amortised constant time, if they only return temporary variables.

Notes

We suggest to compute new values during preincrement calls because dereferencing of the same value can happen multiple times.

Algorithm 1 Pipelined Algorithm Framework

```
template<class Input_Type_1,... , class Input_Type_n,...>
class piped_alg{
public:
 typedef output_type value_type;
private:

 /* References to input algorithms with pipeline conform structure */
 Input_Type_1& A1;
 ...
 Input_Type_n& An;

 /* Temporary variable for the current output element */
 value_type result;
 bool empty_condition;
 ...

public:

 /* Connect input pipes to this pipelined algorithm
  * and compute the first result.
  */
 piped_alg(Input_Type_1& A1_,..., Input_Type_n& An_,...)
 :A1(A1_),...,An(An_),...{
  ...
 }
 /* Only output current result */
 const value_type& operator*()const{
  return result;
 }
 /* Compute next element and store it in result. */
 pipe_alg& operator++(){
  ...
  return *this;
 }
 /* Checks, if none elements are left */
 bool empty(){
  return empty_condition;
 }
};
```

The algorithms framework *Algorithm* 1 gives some hints how to design pipelined algorithms. A simple example for a pipelined algorithm follows in *Algorithm* 2. The *bad_memory* class represents a pipeline node with in and out degree of one which implements the pipeline algorithm interface. The

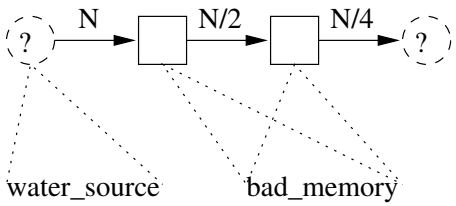

FIGURE 3.2.1. Pipeline algorithm example data flow.

only action the algorithm takes is to drop every second element. To keep the example short, we hide the incoming pipeline node (*water_source*) connected to the first *bad_memory* instance (aqueduct). The second instance of *bad_memory* (thirsty_people) is connected to the output of *aqueduct*. *Aqueduct* and *thirsty_people* form a pipe which lets every fourth input element through. *Figure* 3.2.1 shows the corresponding data flow for the example. For simplicity we assume for the flow, that the number of incoming *water_source* elements (variables of type *water_source_type*) is dividable by four.

Algorithm 2 Pipeline Algorithm Example

```
template<class Input>
class bad_memory{
public:
 typedef typename Input::value_type value_type;
private:
 Input A;
 value_type result;
 bool empty_flag;
 bad_memory(){}
public:
 bad_memory(Input& A_):A(A_),empty_flag(false){
  if(!A.empty()) result=*A;
  else empty_flag=true;
 }
 const value_type& operator*(){
  return result;
 }
 bad_memory& operator++(){
  ++A; // A was checked for not being empty
  if(!A.empty()) ++A;
  else empty_flag:=true;
  if(!A.empty()) result=*A;
  return *this;
 }
 bool empty(){
  return empty_flag;
 }
};
/* Now build a pipeline */
int main(){
 // The people of saarbruecken are always thirsty. So lets
 // transfer water to them.
 // Let water_source be an arbitrary input pipe
 // and water_source_type its type.
 ...
 typedef bad_memory<water_source_type> alg1_type;
 typedef bad_memory<alg1_type>       alg2_type;

 // Loose 1/2 of the elements of watersource
 alg1_type aqueduct(water_source);
 // And again 1/2
 alg2_type thirsty_people(aqueduct);
 // The output of thirsty_people is only about 1/4
 // of the elements of water_source
 while(!thirsty_people.empty()){
  // Get element from pipe
  alg2_type::value_type drink = *thirsty_people;
  // Compute next element
  ++thirsty_people;
 }
 return 0;
}
```

CHAPTER 4

Theoretical External Memory Models

Since we work with external memory algorithms, we cannot argue with the RAM model alone. Our algorithms use two kinds of memory with different behaviour, which must be taken into account to compare different external memory algorithms.

An early model of this kind came from Aggarval and Vitter [**2**]. Their external memory model has a fast internal memory of size M and external memory of unlimited size. One read or write disk access (*I/O access*) moves B elements[1] from disk to internal memory or stores them to disk. A disk can have several heads, which can move independently. If a disk has $P \geq 1$ heads, it can transfer $P \cdot B$ elements in one *I/O access* in this disk model. A redesign of this model in 1994 from Vitter and Shriver [**30**] allowed multiple disks D with single heads and multiple *CPUs* P. Each disk can move B elements, what means that D disks can move $D \cdot B$ elements in one *I/O access*. They assume, that the block size is fixed and smaller data can only be accessed in internal memory. For $B > 1$, for example, we cannot read a single element from disk, because we always have to read B elements in one I/O access. The performance parameters used with this model are:

(1) The number of *I/O accesses*
(2) The number of *CPU* operations (*RAM* model)
(3) The number of disk blocks used

The main problem with this model is, that it does not distinguish between random disk accesses and bulk disk accesses. A bulk disk access means reading consecutive blocks, which is much faster than seeking the position of every block. The reason for this is, that disks are partitioned in tracks and their disk heads (read/write heads) move physically between these tracks. This is a mechanical action, which takes on the average about $\approx 10ms$ time per seek for nowadays disks. The precise seek time depends on the distance a head has to move. The *I/O* model of Farach et al. [**12**] is more realistic, because it cares for bulk *I/Os*. They define an bulk *I/O* as $c \cdot \frac{M}{B}$ consecutive *I/Os* for a constant c.

Besides the Stxxl implementations of external sort algorithms and an external array, which we use for the suffix array algorithms, there is only very little use of random accesses. More precisely, only the *DC3* algorithm needs non bulk *I/Os* for finding the beginning and the middle of the stored external arrays in each iteration[2]. Since pipes output sequences, we can read/write in consecutive

[1] as unit for example machine words

[2] *DC3* needs $\Theta(h)$ bulk *I/Os*, if h is the number of *DC3* sub problems, of which we have $\Theta(\log_{\frac{2}{3}} N)$ for an input string of length N

accesses from/to file nodes. Because of this, it suffices to work with the model of Vitter and Shriver. We hide the *I/O volume* of the sorter and the external array in terms $\text{sort}(N) = O(N \cdot \log_{M/B} \frac{N}{M})$ and $\text{scan}(N) = O(N)$. The number of *I/O* accesses is given by $\text{sort}_{I/O}(N) = O(\frac{N}{D \cdot B} \cdot \log_{M/B} \frac{N}{M})$ and $\text{scan}_{I/O}(N) = O(\frac{N}{D \cdot B})$. For the internal work, we write $\text{sort}_{internal}$ and $\text{scan}_{internal}$ and for the disk space complexity sort_{space} and scan_{space} . Note, that $\text{sort}_{space}(N)$ represents in the context of disk space complexity the maximum number of words, simultaneous stored on the disks, during the external sort algorithm sorts N words. We assume, that sort and scan have the following properties (analogous for $\text{sort}_{I/O}$, $\text{scan}_{I/O}$, sort_{space}, scan_{space}, $\text{sort}_{internal}$ and $\text{scan}_{internal}$):

- $\text{sort}(N_1) + \text{sort}(N_2) \leq \text{sort}(N_1 + N_2)$
- $\text{scan}(N_1) + \text{scan}(N_2) \leq \text{scan}(N_1 + N_2)$

The running time of the suffix array construction algorithms is strongly dominated by the running time of the sorters, which is affected by the time complexity of the used comparison functions. This is the case, because all scan pipes have linear time complexity, where the sorter needs $O(N \log N)$ time for sorting N items. For instance, we reduced the running time of our implementation of doubling by a factor of 2/3 on our test system, simply by optimising the comparison function of one sorter. Note, that all scan nodes have linear time and *I/O complexity* in this work.

Since most running time and disk space related issues are hidden in the sorter, we have to analyse the sorter costs for each algorithm more precisely. We also cannot measure the time complexity in terms of scan because a scan node can be an arbitrary pipe. Hence, we can only give O notation for the running time.

For avoiding multiple merge passes, or using more internal memory with our external merge sort, we use different block sizes in our implementations. Hence, we cannot measure the *I/O* complexity in *I/O* accesses, but we can measure the *I/O volume* of the different algorithms in bytes (write+read).

CHAPTER 5

Definitions And Notation

For symbols and smaller definitions see *Appendix* A.1.

5.1. String Order

Let Σ be an ordered alphabet meaning that for $a, b \in \Sigma$ with $a \neq b$, we have $a < b$ or $b < a$ but not both. If X and Y are strings over Σ^N, then X is larger ($X > Y$), equal ($X = Y$) or smaller ($X < Y$) than Y. Definition of the *string order*:

- $X = Y \iff \forall i \in \{0, \ldots, N-1\} : X[i] = Y[i]$
- $X > Y \iff \exists i \in \{0, \ldots, N-1\} : X[0, i-1] = Y[0, i-1] \wedge X[i] > Y[i]$
- $X < Y \iff \neg(X = Y) \wedge \neg(X > Y)$

It is easy to see, that this definition is well defined on strings of equal length. We will also use shortcuts as $X \leq Y$ and $X \geq Y$ which means $X < Y \vee X = Y$ and $X > Y \vee X = Y$.

Assuming w.l.o.g. S is shorter in length than T, we can compare S with $T[0,\ |S|-1]$. With this assumption, we can get the order between S and T for less ($<$) and greater ($>$), but not equality. If we reserve a special character $\$ \in \Sigma$ for the end position of the strings then equality cannot happen.

5.2. Suffix Array

Let $ be the smallest characters of an alphabet Σ. Consider the string S over $\Sigma \backslash \{\$\}$ of size N and the string $T := S \circ \$$ of size $N+1$, then two different suffixes $T_i := T[i,\ N]$ and $T_j := T[j,\ N]$ with $0 \leq i, j < N$ have order less or greater. The suffix array SA of S is a permutation of the integers $0 \ldots N-1$ with the restriction $i < j \iff T_{SA[i]} < T_{SA[j]}$. An example can be found in *Figure* 5.2.1. The inverse suffix array SA^{-1} is defined as $SA^{-1}[i] = j$ if $SA[j] = i$.

5.3. String Names

Let $X := \langle X_i \rangle_{0 \leq i < N}$ a sequence of strings of arbitrary length and let $U := \langle U_i \rangle_{0 \leq i < M \leq N}$ be the sub sequence of unique strings in X. The sequence V, which consist of M different integers, is a string name sequence of U if $V[i] < V[j] \iff U[i] < U[j]$. If $V[i]$ is a string name for $U[i]$ then $V[i]$ is also a string name for all $X[j]$ with $X[j] = U[i]$.
The important property of all the *suffix array* construction algorithms we will describe later is, that the integer *string names* used there, in the process of computation, will never get larger than the input string size.

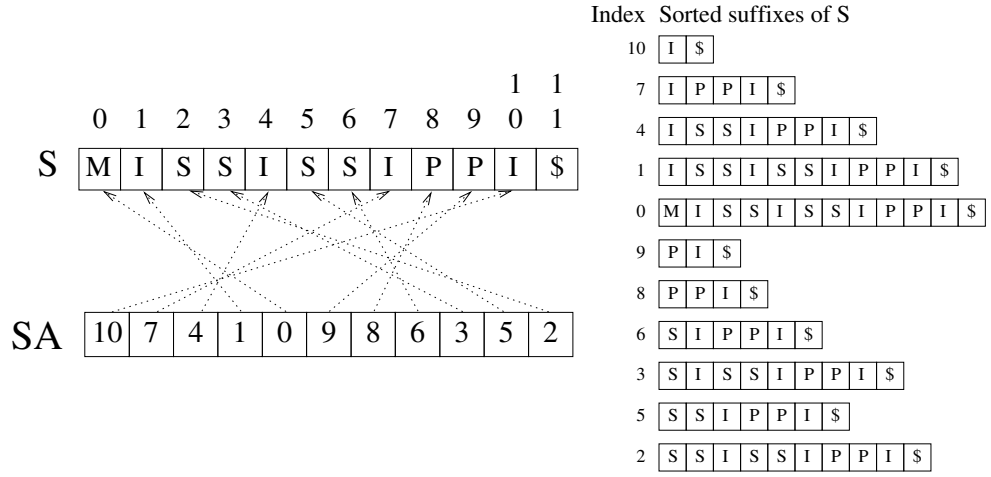

FIGURE 5.2.1. Suffix array of the string "MISSISSIPPI"

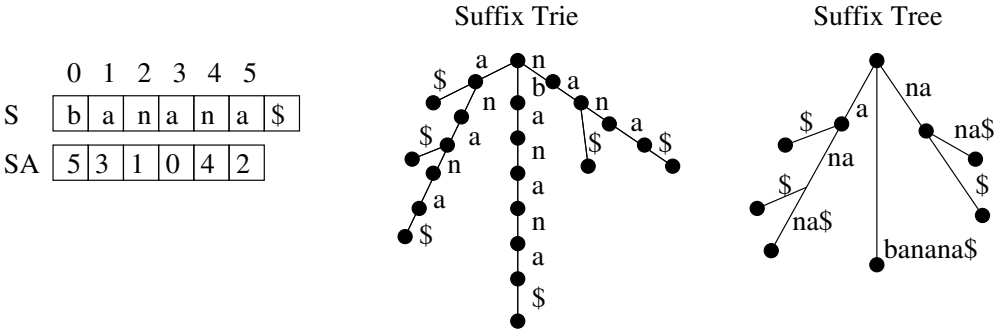

FIGURE 5.4.1. Suffix array, suffix tree and suffix trie of the string "banana"

5.4. Trie And Suffix Tree

A *trie* is a tree data structure with a root (definitions as in [**1**]). The edges are labelled with characters and the nodes represent the string of concatenated edge characters beginning at the root to the node. A *trie* for a set of strings is a *minimal trie* if its leafs represent all the strings in the set without repetition. A *compact trie* is derived from a *minimal trie* by replacing each maximal brancheless path by a single edge, labelled by the concatenated replaced edge labels. A *suffix tree* of a string S is the *compact trie* of all suffixes of S.

5.5. Longest Common Prefix

The *longest common prefix length* of strings T and S is defined as the maximum $r \in \mathbb{N}_0$ with $T[0, r-1] = S[0, r-1]$.

We define $\mathrm{lcp}(i,\ j)$ $(i < j)$ as the length of the *longest common prefix* of the suffixes $S_{SA[i]}$ and $S_{SA[j]}$. For $j > N-1$ and $i < 0$ we define $\mathrm{lcp}(i, j) := 0$.

In some works the *lcp array* $LCP[i] := \mathrm{lcp}(i, i+1)$ for $0 \leq i < N$ is part of the *suffix array* definition. For $i < 0$ or $i \geq N$, we define $LCP[i] := 0$.

If we want to find $\max_j \text{lcp}(i,\ j)$ $(i \neq j)$ for a suffix S_i we simply compute $\text{maxlcp}_i := \max(LCP[i-1],\ LCP[i])$. If $\text{maxlcp}_i = n$ then S_i is different from all S_j $j \neq i$ with their prefixes of length $\leq n+1$.

With the *maximum lcp* of the input string $\text{maxlcp} := \max_{0 \leq i < N} \text{lcp}(i,\ i+1)$ we can distinguish two arbitrary suffixes S_i and S_j $(i \neq j)$ with their prefixes of length $\leq 1 + \text{maxlcp}$.

If each suffix S_i causes $\leq \log_b(1 + \text{maxlcp}_i) \cdot c$ work for a constant c in some of our suffix array construction algorithms then the *logarithmic lcp* to the base b $\text{loglcp}_b := \frac{1}{N} \sum_{0 \leq i < N} \log_b(1 + \text{maxlcp}_i)$ is a measure for the average work for computing the rank of a suffix in these algorithms.

5.6. Alphabets

A *constant alphabet* has a fixed size $|\Sigma| = O(1)$, an *integer alphabet* has size $|\Sigma| = N^{O(1)}$ if N is the input size of our algorithm. In a *general alphabet*, we assume that only characters can be compared in constant time (alphabet definitions as in **[18]**).

5.7. Pseudo Code Conventions

The pseudo code in this work should be understandable by everyone who knows basic concepts in programming languages (loop types, functions...), basic data structures (list, array...) and has basic knowledge in school math (function, set...).

Furthermore the pseudo code is optimised for being short. This is done by eliminating brackets by indention of the code (at loop bodys, functions...), by ignoring obvious special cases (as supporting arbitrary input instances for the algorithms), by hiding the implementation of trivial functions and by math. We will describe the notation we use:

$I := \langle i_0,\ i_1, \ldots,\ i_{N-1} \rangle = \langle i_i \rangle_{0 \leq i < N}$ is a sequence of $N \in \mathbb{N}_0$ sequence items $i_0, \ldots, i_{N-1}$. Sequence item i_e for $0 \leq e < N$ can be accessed by $I[e]$ and we assume $I[e] := \$$ for $e \geq N$ in our algorithms. Sequence items can be tuples, which are also represented as sequences. In this work all nested sequences are sequences of tuples. The items of a tuple are indexed starting with one. For example $P := \langle \langle a_{i,1},\ a_{i,2}, \ldots, a_{i,x} \rangle \rangle_{0 \leq i < N} = \langle a_{i,1},\ a_{i,2}, \ldots, a_{i,x} \rangle_{0 \leq i < N}$ is a sequence with N sequence items, which are sequences of x sequence items, called x-tuples in this context. If $e_0,\ e_1, \ldots, e_{M-1}$ for $M \leq x$ are tuple item positions of the tuples in P, then we can build a new sequence $K := P_{[e_0,\ldots,e_{M-1}]} = \langle a_{i,e_0}, \ldots, a_{i,e_{M-1}} \rangle_{0 \leq i < N}$ from P. In this context $K[i]$ for $0 \leq i < M$ represents the tuple $\langle a_{i,e_0}, \ldots, a_{i,e_{M-1}} \rangle$ and $K[i]_{[y]}$ the tuple item a_{i,e_y} for $0 < y \leq M$. If a_{i,e_y} is again a tuple, we can access its x-th tuple item with $K[i]_{[y][x]}$.

If we want to sort a sequence I by its first tuple component, we write "sort(I) by $I_{[1]}$". The sorted result is a new sequence I, sorted by its first tuple component in ascending order. The most generical sort case is to sort a sequence I by a sequence L of equal length N, written as "sort(I) by L". This means, that we sort the tuple sequence $H := \langle L[i],\ I[i] \rangle_{0 \leq i < N}$ by its first component and that we extract $H_{[2]}$ from the sorted sequence as result.

CHAPTER 6

Basic Sort Algorithms

In this chapter we describe two basic sort algorithms. Radix sort is used in **[18]** for internal memory *suffix array* construction. For external memory concerns we use the external merge sort described in **[10]**. We will describe here a simplified external memory merge sort.

6.1. The Radix Sort Algorithm

Let the alphabet size be $|\Sigma| = m < \infty$ and we want to sort a sequence $S := \langle a_i \rangle_{0 \le i < N}$ of strings of fixed length k. A *stable sorter* maintains the relative order between equal strings after sorting S. An example is $S1 = \langle a,\ b,\ b,\ a \rangle$ with $m = 2$ and $k = 1$. The only valid output of a *stable sorter* is in this case the sorted sequence $T1 = \langle S1[0],\ S1[3],\ S1[1],\ S1[2] \rangle$.

One radix pass sorts a input sequence stably by a single character of the sequence items. In the example above we have stably sorted $S1$ by the first and only character.

S can be sorted with radix sort in k passes of which each pass sorts S stably by a different character of the sequence items. There are two common ways to choose the order of the radix passes. Most significant digit *radix sort* (MSD) does k radix passes from the first to the last character of the sequence items. The least significant digit *radix sort* (LSD) radix pass order is vice versa. An example of LSD, where $S2$ is the input sequence and $T2$ the sorted sequence of strings of length k=3 is as follows:

	S2						**T2**
i							
0	$\langle c, a, t \rangle$		$\langle s, a, (d) \rangle$		$\langle s, (a), d \rangle$		$\langle (c), a, t \rangle$
1	$\langle c, u, t \rangle$	$\Rightarrow$	$\langle c, a, (t) \rangle$	$\Rightarrow$	$\langle c, (a), t \rangle$	$\Rightarrow$	$\langle (c), u, t \rangle$
2	$\langle s, a, d \rangle$		$\langle c, u, (t) \rangle$		$\langle r, (a), t \rangle$		$\langle (r), a, t \rangle$
3	$\langle r, a, t \rangle$		$\langle r, a, (t) \rangle$		$\langle c, (u), t \rangle$		$\langle (s), a, d \rangle$

Radix sort can be implemented with running time $O(kN)$.

6.2. A Simple External Memory Merge Sort Algorithm

Let M be the size of main memory, B the block size as defined in *Chapter* 4. We want to sort a sequence $S := \langle a_i \rangle_{0 \le i < N}$ over an *integer alphabet* Σ with $N \gg M$. For simplicity let $(N/M) \in \mathbb{N}$, $(M/B) \in \mathbb{N}$ and $(N/M) \le (M/B)$. First split S into $m := N/M$ equal sized subsequences R_i with $0 \le i < m$. Then read each R_i into the main memory, sort it and store it in blocks of size B to the disk. The sorted sub sequences R_i are called *sorted runs*.

Now load for each *sorted run* R_i the block with the smallest elements into the main memory. The main memory now contains the smallest elements of S. This is our invariant. We can now merge a part of the final sorted sequence from these blocks, until some blocks of the *sorted runs* start with smaller elements than our smallest elements in the main memory. In this case, we have to load these blocks from the disks and continue the merge process. Note that we only have to consider the "smallest block" of each *sorted run* in such a case. Assuming that we never run out of main memory, we can merge the final sorted sequence by following this strategy.

The external merge sort, which we use in our algorithms, is described in detail in **[10]**.

CHAPTER 7

Doubling

7.1. The Idea Of Doubling

Doubling is a suffix array construction algorithm, which can compute the suffix array of a string S over an *integer alphabet* $\Sigma\backslash\{\$\}$ of size N in $O(N \log N)$ time. For doing this, it needs $O(\log N)$ doubling steps. A doubling step computes names for a sequence of N pairs of alphabet characters. Note, that the computed names are also elements of Σ and that \$ is smaller than all other elements. The idea of doubling is it to compute names for all substrings $S[i,\ i + 2^h - 1]$ of S in doubling step h, for $0 \leq i < N$ and $h \geq 0$. This is done by using the names computed in doubling step h for computing the names of doubling step $h + 1$. More precisely, if $n_{h,i}$ is the name for the substring $S[i,\ i + 2^h - 1]$, we can compute the name $n_{h+1,i}$ of $S[i,\ i + 2^{h+1} - 1]$ by computing names for all

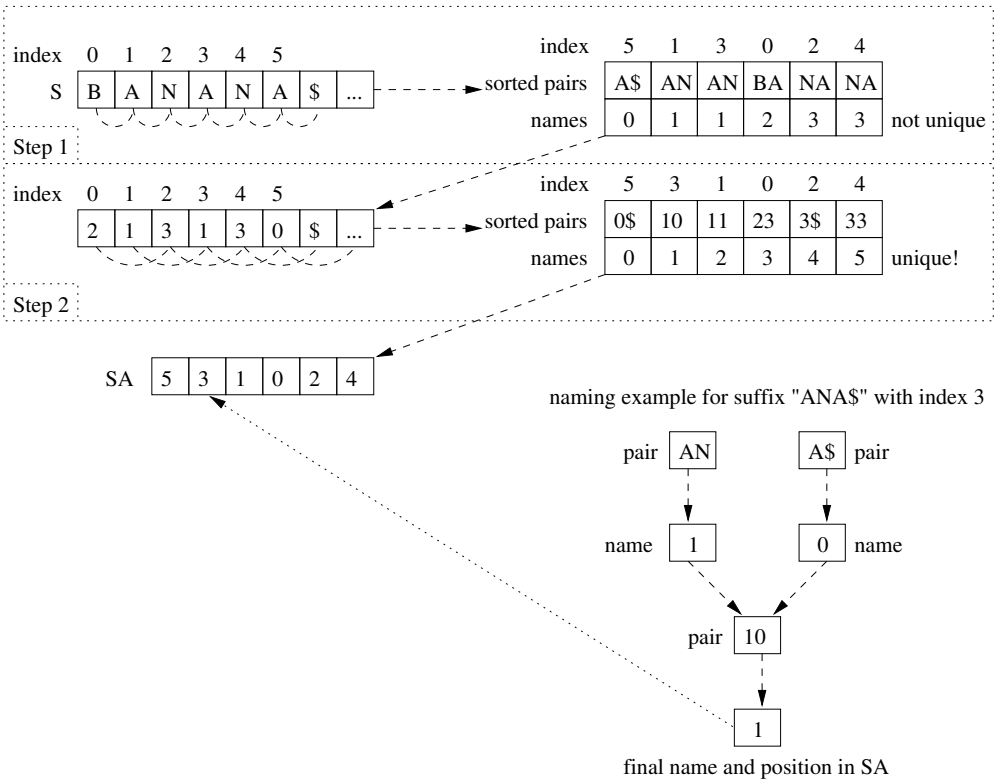

FIGURE 7.1.1. Doubling example for the input string "banana".

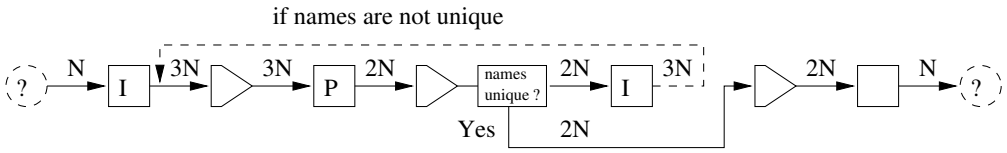

FIGURE 7.3.1. Pipelined doubling flow chart. The scan node labels refer to the sequences in the pseudo-code of *Algorithm* 3. An edge weight denotes the number of words flowing through an edge, if data is passed from one pipeline node to another one.

tuples $\langle n_{h,i},\ n_{h,i+2^h}\rangle_{0\le i<N}$. Since \$ is the smallest element of Σ, there exists a $k \geq 0$ for which all names $n_{k,i}$ are unique. In this case, we have computed unique names for all sub strings $S[i, i+N-1]$. If we sort $I := \langle i,\ n_{k,i}\rangle_{0\le i<N}$ with respect to the second component then $I_{[1]}$ is the suffix array of S.

The example in *Figure* 7.1.1 needs only two doubling steps for computing unique names. The important property of *doubling* is, that during all iterations a computed name needs no more than $\lceil \log_2 N \rceil$ bits.

7.2. The Optimised Index Comparison Function

In pipelined algorithms random array accesses are replaced by a sort step followed by a scan step. In any scan step of the pipelined doubling algorithm we will describe in *Section* 7.3, we need to get the name tuples $\langle i,\ n_i\rangle$ and $\langle i+2^h,\ n_{i+2^h}\rangle$ at distance 2^h to build the new triple $\langle i, n_i, n_{i+2^h}\rangle$. The costly solution is to use two pipes. One starts reading at position 0 and the other at position 2^h (for small distances the internal memory can be used). Thus, we have to store a part of the data to disk because the input data comes out of a *pipe*. Luckily there is a way to arrange the tuples in such a way, that we can build a new triple by having three consecutive tuples from the sorter pipe in memory.

All integer sequences of the form $a_{n,e} := e + n \cdot 2^x$ (for $e \in \{0 \dots 2^x - 1\}$ and $x \in \mathbb{N}$) are independent in having no common elements for fixed e. Thus, we can split each index sequence $b_i := i$ into 2^x sequences of the form $a_{n,e}$. This means that we can rearrange the name tuples, such that $\langle i,\ n_i\rangle$ and $\langle i+2^h,\ n_{i+2^h}\rangle$ are adjacent.

Instead of sorting the name tuples by the indices, we now sort them by $\langle i \mod 2^x, \lfloor \frac{i}{2^x} \rfloor \rangle$. The first component brings the indexes $i = a_{n,e}$ for a fixed e together and the second achieves the order with respect to n. Since rotation $i \longleftrightarrow \langle i \mod 2^x, \lfloor \frac{i}{2^x} \rfloor \rangle$ is a bijection, we can avoid complex comparison functions and do two index transformations (two bit rotations) before and after a sorter. If we replace the basis two with an arbitrary basis $b \geq 2$, we can also produce $b+1$-tuples $\langle i, n_{i+0\cdot b^h}, n_{i+1\cdot b^h}, \dots, n_{i+(b-1)b^h}\rangle$ with $O(b+1)$ internal memory. This trick is used in all subsequent algorithms.

7.3. A Pipelined Doubling Algorithm

Doubling was first introduced in [**3**]. To save the description overhead of the

Algorithm 3 Pipelined Doubling

```
unique := false

Function doubling(T)
  N := |T|
  I := ⟨i, ⟨T[i], T[i + 2^0]⟩⟩_{0≤i<N}
  h := 1
  while(!unique) do
    // ∀⟨i, n_i⟩ ∈ P : n_i names for T[i, i + 2^h − 1]
    sort(I) by I_[2]
    // naming pipe
    P := name(I)
    sort(P) by ⟨P[i]_[1] mod 2^h, P[i]_[1] div 2^h⟩_{0≤i<N}
    if (!unique)
      // update pipe
      I := ⟨i, ⟨P[i]_[2], P[i + 1]_[2]⟩⟩_{0≤i<N}
    h++
  sort(P) by P_[2]
  return P_[1]

Function name(S)
  counter := 0
  result := ⟨S[0]_[1], counter⟩
  tmp := S[0]_[2]
  unique := true
  for i := 1 to |S| − 1 do
    if tmp = S[i]_[2] then
      result.push_back(⟨S[i]_[1], counter⟩)
      unique := false
    else counter++; result.push_back(⟨S[i]_[1], counter⟩)
    tmp := S[i]_[2]
  return result
```

pipeline structure, we describe *doubling* with *Algorithm* 3 and the pipeline structure with its corresponding flow chart in *Figure* 7.3.1. In *doubling* and in *discarding* algorithms only the sorters consume disk space. By elements, we mean sequence items.

We use the optimised comparison function in the pseudo-code. In h iterations and for an input string of size N this measure saves us an *I/O volume* of $O(h \cdot \mathrm{scan}(N))$. Without this scan part we get an *I/O volume* of $O(h \cdot \mathrm{sort}(N))$ for the *pipelined doubling* algorithm. For internal work, we get $O(h \cdot (\mathrm{scan}_{internal}(N) + \mathrm{sort}_{internal}(N)))$, which is strongly dominated by the sorters. As we mentioned earlier, the first sorter in the loop must be divided into two parts. The first part stores the sorted runs to disk at the loop body bottom and the second part reads and merges the data at the loop body, or in the last sorting step after the loop. If the second component of all tuples in the

sequence P is unique, we cannot avoid entering the next sorter. This condition can be checked by the function name(I). After a tuple is checked for uniqueness, it leaves the naming pipe and enters the *sorter run formation node* of the next sorter. The worst case input is a sequence of N equal characters, because the first two suffixes have $N-1$ common characters.

Since we hide the *I/O* costs in terms of sort and scan, we have to do a more detailed analysis about their constant factors. This can be done by counting the edge costs for sort and scan in the flow chart, and we also have to consider some special cases. For the loop body, we have to multiply the edge costs with the needed number of iterations. Note, that our constant factors are real upper bounds, which means that an algorithm can be implemented with this *I/O volume* or better. We cannot avoid the first iteration, because we cannot check the uniqueness of arbitrary input strings simply by scanning through them. Thus, we count an additional iteration to our upper *I/O volume* bound which is $\leq \text{sort}(2N) + (1+h) \cdot \text{sort}(5N)$.

THEOREM 7.3.1. *Doubling with the optimised index comparison function can be implemented with an I/O volume of* $\leq \text{sort}(2N) + (1 + \lceil \log_2(1 + \text{maxlcp}) \rceil) \cdot \text{sort}(5N)$.

PROOF. We can argue more precisely about the number of iterations an input string causes. In iteration h doubling creates names for substrings of size 2^h. A substring is unique if its size is larger than *maxlcp*. Thus we need $k := \lceil \log_2(1 + \text{maxlcp}) \rceil$ iterations if $k > 0$ for creating unique names because $2^{k-1} \leq \text{maxlcp} < 2^k$ and one iteration, if k = 0 (this is the case, if all input string characters are different). □

For computing the disk space complexity of doubling, we assume that the sorters free disk space for sort items, which have left the sorter pipe. This means that we only have to count the disk space complexity of the most expensive sorter, which sorts tuples of size three words. Hence, we have a disk space complexity of $\text{sort}_{space}(3N)$ words.

7.4. From Pipelined Doubling To Pipelined X-Tupling

A generalisation of doubling is *X-Tupling,* described in *Algorithm* 4, in which the name tuple component has X elements. Doubling is *2-Tupling.* A generalisation to *X-Tupling* is a straightforward change of the *doubling* algorithm, but it is not clear, if *X-Tupling* for $X > 2$ can help to reduce the *I/O volume.* On the one hand the number of iterations of the main loop is reduced to $\leq 1 + \lceil \log_X(1 + \text{maxlcp}) \rceil$ for *X-Tupling.* But on the other hand the naming tuple size increases the *I/O volume* of sorting in each iteration.

The space complexity for *X-Tupling* is $\text{sort}_{space}((X+1)N)$ words because the most expensive sorter has to sort tuples of size $X+1$.

7.5. Finding The Optimal X For X-Tupling

The choice of X in X-Tupling is a tradeoff between increasing the *I/O volume* per iteration and decreasing the number of iterations. If a larger naming tuple decreases the overall *I/O volume,* then there must be an upper bound on the

Algorithm 4 *X-Tupling*

$unique := false$

```
Function xtupling(T, x)
  N := |T|
  P := ⟨i, ⟨T[i + 0 · x^0], ..., T[i + (x − 1)x^0]⟩⟩_{0≤i<N}
  h = 1
  while(!unique) do
    // ∀⟨i, n_i⟩ ∈ I : n_i name of T[i, i + x^h − 1]
    sort(P) by P_[2]
    // naming pipe
    I := name(P)
    sort(I) by ⟨I[i]_[1] mod x^h, I[i]_[1] div x^h⟩_{0≤i<N}
    if(!unique)
      // update pipe
      P := ⟨i, ⟨I[i + 0]_[2], ..., I[i + (x − 1)]_[2]⟩⟩_{0≤i<N}
    h := h + 1
  sort(I) by I_[2]
  return I_[1]
```

optimal tuple size. Else we would take tuples of input string size plus one. For a string S of size N and a fixed X the *I/O volume* per iteration is $\mathrm{sort}(2N) + \mathrm{sort}((X+1)N)$ (sorters for scanning and naming tuples). The *I/O volume* of our sorter is linear with respect to the number of input tuples.

For simplicity, we also assume that the maximum number of iterations is $\log_X(1+N)$ and not $\lceil \log_X(1+N) \rceil$. With this assumption the optimal X does not depend on the number of input characters and thus the optimal X is a constant. We compute the maximal possible *I/O volume* over all iterations.

$$\begin{aligned} & \log_X(1+N) \cdot (3+X) \cdot \mathrm{sort}(N) \\ = \; & \mathrm{sort}(N) \cdot \frac{\ln(1+N)}{\ln X} \cdot (3+X) \\ = \; & \mathrm{sort}(N) \ln(1+N) \cdot \frac{3+X}{\ln X} \end{aligned}$$

To compute the optimal base, we need the extreme values of $f(X) = \frac{3+X}{\ln X}$, which we can get from equation $f'(X) = \frac{1}{\ln X} - \frac{X+3}{X \cdot \ln^2 X} = 0$. We get the only minimum at $e^{1+LambertW(3e^{-1})}$ (Lambert W-Function[1]) with the value ≈ 4.971. The *LambertW-* or *Omega-Function* is the inverse function of $g(X) = X \cdot e^X$. $\mathrm{LambertW}(X)$ is real for $X \geq -\frac{1}{e}$. We are only interested in integer values for X with $X \geq 2$, where $f(2) \approx 7.21$ is no minimal value. We also see in our calculation that the needed *I/O volume* of the sorter is not important for calculating the optimal tuple size, if it is only linear in the number of tuple components.

[1] `http://mathworld.wolfram.com/LambertW-Function.html`

THEOREM 7.5.1. *The I/O optimal naming tuple size for X-Tupling with respect to our optimisation model is five.*

PROOF.

X	2	3	4	5	6	7
$f(X)$	7.21	5.46	5.05	4.97	5.02	5.14

□

Our calculation and the table tells us, that the optimal base for the assumed model is five. One good argument for taking four as base is that the most common CPU architectures are fast for powers of two. Another point is, that one *4-tupling* iteration does the construction work of two *2-Tupling* iterations and sometimes, we waste *I/O volume* because of this (the possible waste increases with the size of the base). An example is the string $S :=$ "abcdefghijklmnop" with $|S| = 16$. We need in this case one iteration for *2-Tupling* and *4-Tupling* but *4-Tupling* needs an additional *I/O volume* of $\text{sort}(2|S|)$.

Analogous to *pipelined doubling,* the *I/O volume* of *4-Tupling* is $\text{sort}(2N) + m \cdot \text{sort}(7N)$ if m is the number of iterations.

CHAPTER 8

Discarding

Discarding is an improvement of doubling. The disadvantage of doubling is that the number of iterations depends on few suffixes with maxlcp. In each iteration doubling computes new unique name components of tuples, which are staying in the loop until all names are unique. Thus for doubling the number of I/Os per iteration is equal.

The idea of discarding is to reduce the I/O complexity for tuples with unique name component. This is done in two steps. First we change the naming strategy, so that a unique name component of updating tuples denotes the final position of the corresponding suffix in the suffix array. As second step we have to change the updating strategy to make it possible to discard updating tuples from one or more sorters in one iteration.

8.1. Changing The Naming Strategy

We will show the naming strategy of discarding, by extending doubling, without describing a discarding algorithm. Nevertheless, the discarding algorithms described later use this naming strategy but their updating strategies differ. Thus, discarding will be described with these algorithms. *Figure* 8.1.1 shows a simple example for a later described discarding algorithm.

Let T be a character[1] string of size N and SA its suffix array. Further $n_{h,i}$ is the name of the substring $T[i,\ i+2^h]$, computed in iteration h. During the naming step of the doubling algorithm, we compute, in iteration h tuples $A := \langle i,\ n_{h,i}\rangle_{0\le i<N}$ from naming triples $B := \langle i,\ \langle n_{h-1,i},\ n_{h-1,i+2^{h-1}}\rangle\rangle_{0\le i<N}$. If $\mathrm{eq}_h(n) := \{i \mid n = n_{h,i}\}$ is an equivalence class for indices of tuples with equal names in iteration h then doubling computes integer names

$$n_{h,i} = \#\{\mathrm{eq}_h(n) \mid \mathrm{eq}_h(n) \neq \emptyset \wedge\ n < n_{h,i}\}$$

for all i in this iteration. This means a name $n_{h,i}$ denotes the number of different smaller names, computed in iteration h. Thus a unique name $n_{h,i}$ of a tuple $\langle i,\ n_{h,i}\rangle$ contains no reliable information about the final position of the suffix T_i in SA. With doubling, we do not get this information till all names are unique, which also means that each equivalence class $\mathrm{eq_h}(n)$ for $0 \le n < N$ contains one element.

Next, we extend the tuples A of the doubling algorithm by a new variable $a_{h,i}$ to the triples $C := \langle i,\ a_{h,i},\ n_{h,i}\rangle_{0\le i<N}$. The variable $a_{h,i}$ is also a name for the substring $T[i,\ i+2^h]$ but we will see its advantage compared to $n_{h,i}$. Define

[1] These characters are also integer names.

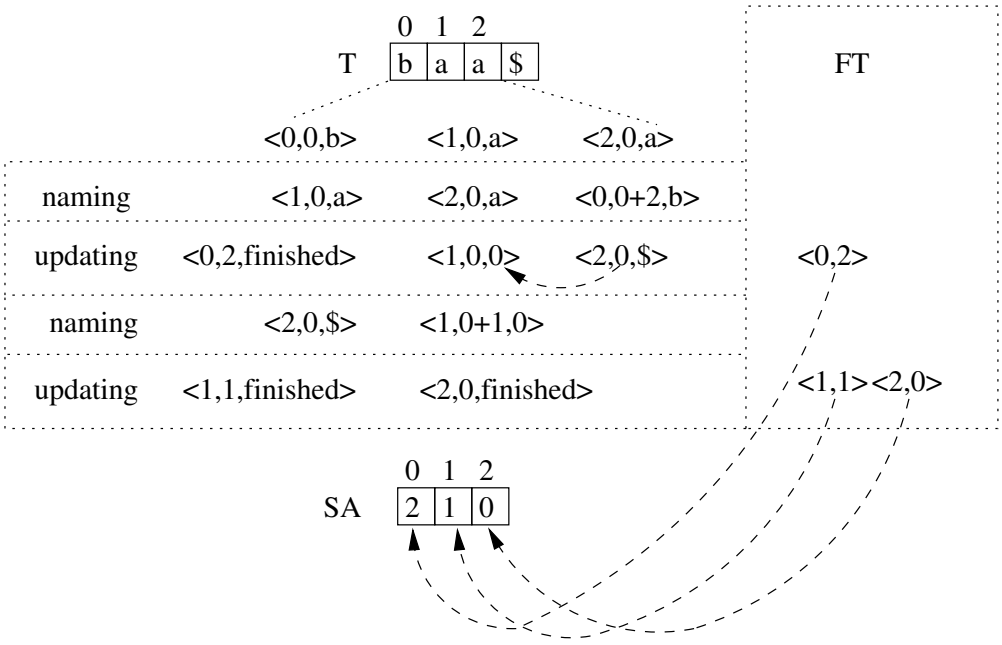

FIGURE 8.1.1. Discarding example for the string "baa"

$C := \langle i,\ 0,\ T[i]\rangle$ for iteration $h = 0$ and

$$\mathrm{M}_h(i) := \#\{\langle a, n\rangle \in \langle a_{h,j},\ n_{h,j}\rangle_{0\le j<N} \mid a = a_{h,i} \wedge n < n_{h,i}\}.$$

In each iteration, we compute $D := \langle i, \mathrm{M}_h(i) +\ a_{h,i},\ n_{h,i}\rangle_{0\le i<N}$ from C. Since $C[i]_{[2]} = 0$ for all i in iteration $h = 0$, the function M_0(i) computes for characters $T[i]$ the precise number of smaller characters of T. This also means that for a unique character $T[j]$ the suffix T_j has the final position M_0(j) in SA. For all following iterations $\mathrm{M}_h(j) = 0$ because $D[j]_{[2]}$ is unique. If $T[k]$ is not a unique character then $D[k]_{[2]}$ denotes the smallest position in SA, where T_k can be. For all suffixes starting with $T[k]$, we know now that their position in SA is in $\{\mathrm{M}_0(k), \ldots,\ \mathrm{M}_0(k) + \mathrm{eq}_0(T[k])\}$. Consider now an arbitrary iteration $h > 0$. If $C[l]_{[2]}$ is not unique, we know that $n_{h-1,l}$ was not unique in iteration $h-1$ and that T_l has in SA a position in $\mathrm{P}_{h-1}(l) := \{C[l]_{[2]}, \ldots,\ C[l]_{[2]} + \mathrm{eq}_{\mathrm{h}-1}(n_{h-1,l})\}$. In the case that $n_{h,l}$ is unique, $C[l]_{[2]} + \mathrm{M}_h(l)$ computes the precise position of T_i in SA and $\mathrm{P}_h(l)$ contains one element.

The first reason why the described naming strategy prepares the ground for discarding is that we can determine final suffix positions in SA in earlier iterations. Secondly, for updating a triple $C[i]$ in iteration h, we only need to know $|\mathrm{eq}_h(n_{h,i})|$ other triples of C.

The naming step of *Figure* 8.1.1 should be understandable now. New with respect to doubling is the list of finished tuples FT. If a triple during the updating step has a unique second component, we put a new tuple in FT, consisting of the first two triple components. This triple can be discarded after this step. After all triples are discarded, we get the final suffix array from the second components of the sorted tuples in FT.

Algorithm 5 Old Discarding

In addition to T, we now have two lists FT (finished triples) and UT (unfinished triples). We begin with the triples $\langle 0,\ T[i],\ i\rangle$, the empty list FT in step 1 and stage $k = 0$ of the algorithm. A finished triple has the form $\langle i,\ pos,\ -1\rangle$ with $SA[i] = pos$. An unfinished triple has the form $\langle i,\ a,\ b\rangle$ where the first two components are the name of this triple, $a, b > -1$.

(1) **Sort** UT with respect to the last two components. If UT is empty goto step 6.

(2) **Scan** UT and update finished status and names of triples. A triple $t = \langle i,\ a,\ b\rangle$ is changed to $\langle i,\ a,\ -1\rangle$, if if the predecessor triple and the successor triple have different ($a \neq a'$ or $b \neq b'$ for a triple $\langle *,\ a',\ b'\rangle$) names with respect to t. Now we compute new names for all triples in UT in the following way: If $M(\langle i, a, b\rangle) := \{\langle x,\ y,\ z\rangle \in UT | a = y \wedge b > z\}$ we replace each triple $t = \langle i,\ a,\ b\rangle$ with $\langle i,\ a + |M(t)|,\ b\rangle$.

(3) **Sort** UT according the first component of the triples.

(4) **Merge** UT and FT into UT according to the third component of the triples. FT will be emptied.

(5) **Scan** UT rightwards and we replace each triple $\langle i,\ a,\ b\rangle$, which is not finished, with $\langle i,\ a,\ z\rangle$, where at z comes from the triple $\langle i + 2^j,\ z,\ *\rangle$. Finished triples are moved to FT. Increment j and goto step 2.

(6) FT is sorted according to the first component of its triples. If we scan FT rightwards for the third component, we get SA.

8.2. Doubling Combined With Discarding

Doubling combined with discarding was introduced in [**7**]. Since this idea was improved in [**1**], we will only give a textual description with *Algorithm* 5 and use the pipeline strategy with the improved one.

As doubling, the algorithm has two sorters in the loop body but the number of input tuples of the sorter in step three can be decreased. This is done by splitting its input tuples in finished tuples FT and unfinished tuples UT. A finished tuple is unique. Let N_h be the number of tuples in UT after iteration h and let N be the number of input characters, then the *I/O volume* in iteration h is $O(\text{sort}(N)+\text{sort}(N_{h-1})+\text{scan}(N)))$. A more detailed analysis of the improved version will follow with *pipelined 2-Discarding*. Then we will see, that we neither need merge step four nor do we need a sorter, which sorts N sort items in each iteration.

8.3. A Pipelined Discarding Algorithm

There is a way to get rid of the triples with unique name component (called finished tuples FT) after an iteration and never merge them back, that was first introduced in [**1**] and used to create the *Algorithm* 6. If we only need the unfinished tuples UT, we do not need to merge UT with FT for one of the sorters, as for the discarding algorithm in [**7**]. The *I/O volume* of both sorters in iteration h is reduced to $O(\text{sort}(N_{h-1}))$. If we want to update the triple $t = \langle i, x, y\rangle$ in iteration h, we also need the triples $t' = \langle i + 2^{h-1}, x', y'\rangle$ and

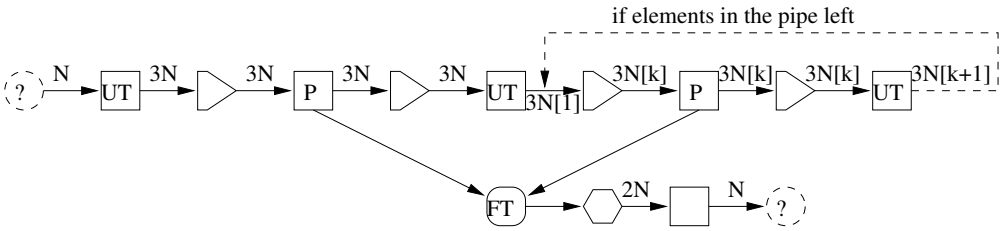

FIGURE 8.3.1. Pipelined discarding flow chart.

$t'' = \langle i + 2^h, x'', y'' \rangle$ in internal memory and differentiate between the following cases:

- If $t'' \in UT$, then $t = \langle i, x, x'' \rangle$.
- If $t' \in UT$ but $t'' \notin UT$, then t'' was already finished before iteration h. Thus the new triple is $t = \langle i, x, y' \rangle$. Note, that y' is precisely the needed x'' of the discarded triple t''.
- If $t' \notin UT$, then it was finished before iteration h and t must be one of the new marked finished tuples. We do not need this case, because we mark finished tuples in an earlier stage of the algorithm. Secondly, this case cannot find all finished tuples as for example in the first iteration, where no tuples are discarded.

In the first iteration, we only need condition one for tuple updating and one sorter is less expensive. Thus we have separated the first iteration from the main loop. For one of the sorters in the loop, we use the optimised index comparison function again, such as other optimisations known from the pipelined *doubling*. Again, only the sort algorithms store data to the disks. In the naming step, we store sorted subsequences (*sorted runs*) of finished tuples to FT, which we only have to merge in a final step after the last iteration. Note that we also need to store the information whether a tuple is marked for discarding in the triple structure. Our implementation encodes this as a bit in one of the three tuple components.

The worst case *I/O volume* is $O(\log_2 N \cdot \mathrm{sort}(N) + \mathrm{scan}(N))$. We cannot determine the constant factors of scan and sort with respect to the number of iterations as for *X-Tupling* because we cannot know the number of discarded tuples per iteration. This depends on the input. On the other hand, we can talk about a single iteration. The *I/O volume* in one iteration is $6 \cdot \mathrm{sort}(N_h)$, where N_h is the number of tuples left in iteration h. Note, that for the same number of tuples, doubling is cheaper per iteration with an *I/O volume* of $5 \cdot \mathrm{sort}(N_h)$ in this case.

Discarding discards all tuples, representing the suffixes S_i of the input string S with maxlcp_i in iteration $\lceil \log_2(1+\mathrm{maxlcp}_i) \rceil$ (for $\mathrm{maxlcp}_i > 0$), because of the uniqueness of their names, as described in the *doubling* chapter. This means that these suffixes cause no further costs after this iteration. Without considering the internal details of the scan and sort procedures, we assume that each single tuple causes the same work over each iteration.

Algorithm 6 *Pipelined discarding*

```
Function 2-discarding(T)
  N := |T|;  FT := ⟨⟩
  TMP := ⟨i, T[i]⟩_{0≤i<N}
  sort(TMP) by TMP_[2]
  UT := ⟨TMP[i]_[1], ⟨0, TMP[i]_[2]⟩⟩_{0≤i<N}
  P := name2(UT, 2)
  sort(P) by P_[1]
  UT := simple_discard(P, 2)
  h := 1
  while(UT ≠ ⟨⟩) do
    \\ ∀⟨i, ⟨a_i, n_i⟩⟩ ∈ P : a_i names for T[i, i + 2^h − 1]
    sort(UT) by UT_[2]
    P := name2(UT, 2)    // naming
    sort(P) by ⟨P[i]_[1] mod 2^{h−1}, P[i]_[1] div 2^{h−1}⟩_{0≤i<N}
    UT := discard(P, h)  // updating
    h++
  sort(FT) by FT_[2]
  return FT_[1]

Function name2(S, x)
  counter := 0
  store := 0
  result := ⟨S[0]⟩
  for i := 0 to |S| do
    if S[i − 1]_[2][1] = S[i]_[2][1] then
      if S[i − 1]_[2][2] ≠ S[i]_[2][2] then store := counter;
      counter++
      result.push_back(⟨S[i]_[1], ⟨S[i]_[2][1] + store, S[i]_[2][x]⟩⟩)
    else counter := 0; store := 0
      result.push_back(S[i])
    if (S[i − 1]_[2] ≠ S[i]_[2]) ∧
       (S[i]_[2] ≠ S[i + 1]_[2]) then
      FT.push_back(⟨S[i]_[1], S[i]_[2][1]⟩); set_finished(S[i])
  return result

Function simple_discard(S, x)
  result := ⟨⟩
  for i := 0 to |S| − 1 do
    if is_finished(S[i]) = false then
      result.push_back(
        ⟨S[i]_[1], ⟨S[i]_[2][1], ..., S[i + x − 1]_[2][1]⟩⟩)
  return result

Function discard(S, h)
  result := ⟨⟩
  for i := 0 to |S| − 1 do
    if is_finished(S[i]) = false then
      M := {S[i + 1]_[1], S[i + 2]_[1]}
      if (S[i]_[1] + 2^h) ∈ M then // t'' exists
        result.push_back(⟨S[i]_[1], S[i]_[2][1], S[i + 2]_[2][1]⟩)
      else // t' exists but not t''
        result.push_back(⟨S[i]_[1], S[i]_[2][1], S[i + 1]_[2][2]⟩)
  return result
```

THEOREM 8.3.1. *The I/O volume of pipelined discarding is* $O((1+\text{loglcp}_2)\cdot \text{sort}(6N)+\text{scan}(2N))$.

PROOF. The last sorter only needs to merge the sorted subsequences and thus we count it as scan. Since $\lceil \log_2(1+\text{maxlcp}_i)\rceil$ describes the number of iteration in which a tuple has to stay in the sorters and because the costs are linear we have for the whole work:

$$\frac{1}{N}\sum_{0\le i<N}\lceil \log_2(1+\text{maxlcp}_i)\rceil \le 1+\frac{1}{N}\sum_{0\le i<N}\log_2(1+\text{maxlcp}_i) = 1+\text{loglcp}_2$$

□

The disk space complexity is $\text{sort}_{space}(5N)$ words because it is possible that all tuples are in the sorter for FT and also in one of the triple sorters.

8.4. From Pipelined Discarding To Pipelined X-Discarding

Since *pipelined doubling* (*2-Doubling*) was improved by increasing the naming tuple size, we can do the same with *pipelined discarding* (*2-Discarding*). If we compare the *I/O volume* in the iterations of *X-Discarding,* described in *Algorithm 7,* with *X-Tupling,* the only difference is that one of the sorters needs to sort triples instead of tuples. The case distinction of the update triples of *2-Discarding* has to be expanded in a simple way. For the triple $t=\langle i,\langle u_i,v_i\rangle\rangle$ at position $0\le i<N$ in the discarding step of iteration h, we consider the longest series of not discarded triples at positions $i,\ i+X^h,\ i+2\cdot X^h,\ \ldots,\ i+e\cdot X^h$ with $0\le e<X-1$. From the triple $t''=\langle i+e\cdot X^h,\langle u_{i+e\cdot X^h},\ v_{i+e\cdot X^h}\rangle\rangle$ the case distinction continues as in $2-Discarding$ because we need the triple $t'=\langle i+X^{h-1}+e\cdot X^h,\langle u_{i+X^{h-1}+e\cdot X^h},\ v_{i+X^{h-1}+e\cdot X^h}\rangle\rangle$ if $e<X-2$. Hence, we need to have the triples

- $t=\langle i,\langle u_i,v_i\rangle\rangle$,
- $t'_e=\langle i+X^{h-1}+e\cdot X^h,\langle u_{i+X^{h-1}+e\cdot X^h},\ v_{i+X^{h-1}+e\cdot X^h}\rangle\rangle$ and
- $t''_e=\langle i+e\cdot X^h,\langle u_{i+e\cdot X^h},\ v_{i+e\cdot X^h}\rangle\rangle$ for $0\le e<X-1$

in the internal memory. From these triples, we build the new $(X+1)$-tuple $s:=\langle i,\langle u_i,\ u_{i+X^h},\ldots,\ u_{i+e\cdot X^h},\ v_{i+X^{h-1}+e\cdot X^h},\ldots\rangle$. Note, that we get $s:=\langle i,\langle u_i,\ u_{i+X^h},\ldots,\ u_{i+(X-2)X^h}\rangle\rangle$ for $e=X-2$.
Since the smallest index after the index i of t is the index $i+X^{h-1}$ of t'_0, we are forced to sort P by $\langle P[i]_{[1]}\ \bmod x^{h-1}, P[i]_{[1]}\ div\ x^{h-1}\rangle_{0\le i<N}$ after the naming step. This means, that we cannot sort the triples leaving the naming step in a way, so that we have precisely the triples needed for the discarding step in memory. Thus we have in the updating step at most $(X-1)\cdot X+1$ tuples simultaneously in memory.

Algorithm 7 *X-Discarding*

```
Function xdiscarding(T, x)
  N := |T|
  FT := ⟨⟩
  TMP := ⟨i, T[i]⟩_{0≤i<N}
  sort(TMP) by TMP_[2]
  UT := ⟨TMP[i]_[1], ⟨0, TMP[i]_[2]⟩⟩_{0≤i<N}
  P := name2(UT, x)
  sort(P) by P_[1]
  UT := simple_discard(S, x)
  h := 1
  while(UT ≠ ⟨⟩) do
    \\ ∀⟨i, n_i⟩ ∈ P : n_i names for T[i, i + x^h − 1]
    sort(UT) by UT_[2]
    P := name2(UT, x)
    sort(P) by ⟨P[i]_[1] mod x^{h−1}, P[i]_[1] div x^{h−1}⟩_{0≤i<N}
    UT := discard2(P, h, x)
    h++
  sort(FT) by FT_[2]
  return FT_[1]

Function discard2(S, h, x)
  result := ⟨⟩
  for i := 0 to |S| − 1 do
    if is_finished(S[i]) = false then
      M := {S[i + 1]_[1], S[i + 2]_[1], ..., S[i + x^2 − x]_[1]}
      f : (i, e) ↦ i + (e + 1)x^h
      g : (i, e) ↦ i + x^{h−1} + e · x^h
      r := max(e : ∀0 ≤ i ≤ e ∧ f(i, e) ∈ M)
      if r = x − 2 then
        result.push_back(⟨S[i]_[1], ⟨S[i]_[2][1],
        S[f(i, 0)]_[2][1], ..., S[f(i, x − 2)]_[2][1]⟩⟩)
      else
        result.push_back(⟨S[i]_[1], S[i]_[2][1], ⟨
          S[f(i, 0)]_[2][1], ..., S[f(i, r)]_[2][1],
          S[g(i, r + 1)]_[2][2], *, ..., *⟩⟩)
  return result
```

8.5. Finding The Optimal X For X-Discarding

Since *X-Doubling* and *X-Discarding* have a very similar structure, we should get a similar equation. The size of a naming tuple is $2 + (X - 1)$ words and the scanning tuple is 3 words. Since an element causes a constant *I/O volume* in each iteration, we can continue as in *X-Tupling,* but now we have to argue

with loglcp_X. For arbitrary input strings of size N, we get with *X-Discarding* an *I/O volume* of

$$\begin{aligned} & \frac{1}{N} \sum_{0 \le i < N} \log_X (1 + \text{maxlcp}_i) \, (4 + X) \text{sort}(N) \\ = \quad & \frac{1}{N} \sum_{0 \le i < N} \frac{\ln (1 + \text{maxlcp}_i)}{\ln X} (4 + X) \text{sort}(N) \\ = \quad & \frac{4 + X}{\ln X} \cdot \text{sort}(N) \cdot \frac{1}{N} \sum_{0 \le i < N} \ln (1 + \text{maxlcp}_i) \\ = \quad & \frac{4 + X}{\ln X} \cdot \text{sort}(N) \cdot \text{loglcp}_e \end{aligned}$$

and for the minimum $e^{(\text{LambertW}(4e^{-1})+1)} \approx 5.572$. We choose a base of four for similar reasons as before. A new reason for choosing a smaller than a optimal X is that more iterations give more possibilities to discard tuples, what can reduce the *I/O volume* costs. Consider for example a suffix S_i of an input string S with $\text{maxlcp}_\text{i} = 6$ and $0 \le i < |S|$, for which we need two *4-Discarding* iterations and also two *5-Discarding* iterations. But *5-Discarding* is more expensive per iteration.

THEOREM 8.5.1. *The I/O optimal tuple size for X-Discarding is six.*

PROOF.

X	3	4	5	6	7
$f(X) := \frac{4+X}{\ln X}$	6.37	5.77	5.59	5.58	5.65

□

CHAPTER 9

DCX

The *DC3* algorithm was first introduced by J. Kärkkäinen and P. Sanders in [**18**] as one of three independent solutions for linear time suffix array construction [**18, 21, 22**] introduced in 2003. *DCX*, the generalisation of *DC3*, was first introduces in [**5**]. After we present the idea and a short description of the *DC3* algorithm, we come to one of the main results of this work, a pipelined external version of *DC3*. We also discuss further improvements and generalisations.

We always assume in this chapter that the input string S is closed with a special characters \$ and its size is a power of three to make the suffixes comparable and to give S the matching size for the algorithm (to avoid special cases and for easier understanding). However, in our implementations the string size can be of any size.

9.1. The Idea Of The DC3 Algorithm

(1) For the string S construct a string R of size $\frac{2}{3} \cdot |S|$, from which we can derive $SA_{2/3}$. $SA_{2/3}$ contains names for all suffixes S_i with $i \not\equiv 0 \mod 3$ ordered as in the final suffix array SA. This can be done by sorting the characters in R if it contains only unique characters. Otherwise we go into recursion.
(2) $SA_{1/3}$ contains names for all suffixes S_i with $i \equiv 0 \mod 3$ ordered as in the final suffix array SA. Construct $SA_{1/3}$ with help of the $SA_{2/3}$ and S.
(3) Merge $SA_{2/3}$ and $SA_{1/3}$ into SA.

As shown later computing $SA_{1/3}$, $SA_{2/3}$ with a given R and step three have a $O(\text{sort}(N))$ *I/O volume*. Since $\sum_{i \in \mathbb{N}_0} \left(\frac{2}{3}\right)^i = 3$ (see recursion in step one), the overall *I/O volume* is in $O(\text{sort}(N))$. If we only use internal memory and if we allow an *integer alphabet* for the input string and for the computed names, the *DC3* algorithm needs only $O(N)$ running time using radix sort.

9.2. The DC3 Algorithm In Detail

Algorithm 8 is a short description of the original *DC3* algorithm, as we find it in [**18**].

Algorithm 8 DC3

(1) Sort the suffixes S_i of the input string S with $i \not\equiv 0 \mod 3$ among themselves. This is done by computing the names n_i $(0 \leq n_i < \frac{2}{3}|S|)$ of the substrings $S[i, i+2]$ with radix sort. If there are no equal triples, we build $SA^{12}[n_i] = i$ in one scan step. Else we go into recursion with the new string $s^{12} := [n_i : i \equiv 1 \mod 3] \circ [n_i : i \equiv 2 \mod 3]$ to obtain SA^{12}. The entries of SA^{12} representing the suffixes S_i with $i \not\equiv 0 \mod 3$ are ordered as in SA.

(2) The suffixes S_i with $i \equiv 0 \mod 3$ for constructing $SA_{1/3}$ are generated by sorting the pairs $\langle S[i]\ ,\ S_{i+1}\rangle$. because S_{i+1} is already known from step one. $SA_{1/3}$ can be computed in a single pass of radix sort.

(3) Now we have to merge the two obtained suffix arrays to obtain SA. To compare a suffix S_j with $j \equiv 0 \mod 3$ and a suffix S_i with $i \not\equiv 0 \mod 3$ we distinguish two cases:

If $i \equiv 1 \mod 3$, we write S_i as $\langle S[i]\ ,\ S_{i+1}\rangle$ and S_j as $\langle S[j]\ ,\ S_{j+1}\rangle$. We know the relative order of S_{i+1} and S_{j+1}, because we can lookup their position in SA^{12} . We precompute an array $\widetilde{SA}^{12}$ with $\widetilde{SA}^{12}[i] = j + 1$ if $SA^{12}[j] = i$ in linear time which is needed to lookup S_{i+2} and S_{j+2}.

For the case $i \equiv 2 \mod 3$ we can do the same with the tuples $\langle S[i]\ ,\ S[i+1]\ ,\ S_{i+2}\rangle$ and $\langle S[j]\ ,\ S[j+1]\ ,\ S_{j+2}\rangle$.

We now give a short overview of what has to be done, if we want to design a pipelined version of *DC3*. Foremost, we eliminate the recursion in the first step by two loops. The first loop writes the s^{12} subproblem data to disk, until we reach a subproblem with unique names. In the worst case this needs to store $6 \cdot N$ words to the disks. In the second loop, we compute SA^{12} solving the subproblems bottom-up. The next problem is to minimise the number of random disk accesses. To reduce these, we use an external sort algorithm, which is the external merge sort described in **[10, 9]**. Furthermore we use pipelining, whenever it is possible. For example, if we want to compute the inverse suffix array SA^{-1} from SA, we simply need to sort the tuples $\langle i, SA[i]\rangle_{0 \leq i < N}$ with respect to their second component. The result is a tuple sequence $\langle \widetilde{SA}^{-1}[i], i\rangle_{0 \leq i < N}$.

9.3. A Pipelined DC3 Algorithm

Unlike the non pipelined *DC3* algorithm, the pipelined version should not do many random accesses over the whole string. The number of additional random disk accesses besides the ones of the sorters is very small. More precisely, we access external arrays in bulk accesses (see *Chapter* 4) as the used external merge sort does. *Algorithm* 9 shows the pseudo-code for the *DC3* algorithm and *Figure* 9.3.1 shows how it was implemented with the pipelining structure. The first loop stores the subproblem data in file nodes F_i, which are the input strings of all *DC3* sub problems, represented in the pseudo-code by the input of the function DC3 in recursion i. We compute the inverse suffix array of SA^{12} by using two sorters. The output of the sorters is SA^{-1}_{mod1} and SA^{-1}_{mod2}, which

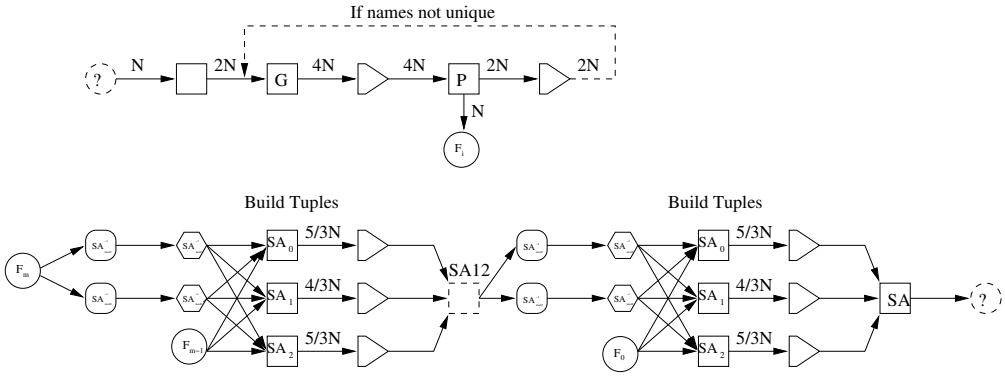

FIGURE 9.3.1. Pipelined DC3 flow chart.

are the first and second half of the inverse suffix array of SA^{12}. These are also the separated names for the suffixes T_i with $i \not\equiv 0 \mod 3$, in the right order to build partial suffix arrays $SA0$, $SA1$ and $SA2$ efficiently. Building these external memory arrays[1] can be done by scanning SA^{-1}_{mod1}, SA^{-1}_{mod2} and the sub problem string T simultaneously. The second pipe of *Figure* 9.3.1 can also be implemented as a loop by using the run formation pipes of the sorters, which compute SA^{-1}_{mod1} and SA^{-1}_{mod2}, to transfer data from the loop end to the loop begin.

The scan *I/O volume* is caused by reading and writing names from/to file nodes.

THEOREM 9.3.1. *The DC3 algorithm has an I/O volume of at most* $\text{sort}(30N) + \text{scan}(6N)$.

PROOF. We simply count the *I/O volume* of the sorter and external arrays over all recursions. Let m be the number of recursions in which a sorter or an external array is needed. Since $\sum_{0 \le i < m} \left(\frac{2}{3}\right)^i < \sum_{0 \le i < \infty} \left(\frac{2}{3}\right)^i = 3$ is true for all $m \in \mathbb{N}_0$, we can compute upper bounds for the *I/O volume* of all relevant pipes.

(1) Sort naming tuples: $\text{sort}\,(4N) \cdot \sum_{0 \le i < m} \left(\frac{2}{3}\right)^i < \text{sort}\,(12N)$.
(2) Write names: $\text{scan}\,(N) \cdot \sum_{0 \le i < m} \left(\frac{2}{3}\right)^i < \text{scan}\,(3N)$.
(3) Read names: $\text{scan}\,(N) \cdot \sum_{0 \le i < m-1} \left(\frac{2}{3}\right)^i < \text{scan}\,(3N)$.
(4) Sort merge tuples: $\text{sort}\left(\left(\frac{5}{3} + \frac{4}{3} + \frac{5}{3}\right) \cdot N\right) \cdot \sum_{0 \le i < m-1} \left(\frac{2}{3}\right)^i < \text{sort}\,(14N)$.
(5) Build inverse suffix array: $\text{sort}\,(2N) \cdot \sum_{1 \le i < m-1} \left(\frac{2}{3}\right)^i < \text{sort}\,(4N)$.

□

It turns out that the *DC3* algorithm is the fastest suffix array construction algorithm on our test system for at least three reasons. Firstly, it has for nearly all cases the smallest *I/O volume* and secondly, it needs few comparisons. Consider, for example, the three sorters in the merge step, of which two sorters need

[1]With pipelining, we do net even need external memory arrays in this case because we can feed the tuples of SA_0, SA_1 and SA_2 directly into the sorters.

Algorithm 9 Pipelined DC3

```
Function DC3(T)
  N := |T|
  G := ⟨i, ⟨T[i], T[i+1], T[i+2]⟩⟩_{(0≤i<N)∧(i≢0 mod 3)}
  I := ⟨e, G[e]_[2]⟩_{0≤e≤|G|}  // build new indices
  sort(I) by I_[2]
  P := name(I)
  if ∃i, j : i ≠ j ∧ P[i]_[2] = P[j]_[2] then
    sort(P) by ⟨P[i]_[1] mod 2, P[i]_[1] div 2⟩_{0≤i<N}
    SA^12 := DC3(P_[2])
    P := ⟨SA^12, i⟩_{0≤i<|SA^12|}
  sort(P) by P_[1]
  // P_[2] is now the inverse suffix array of SA^12
  // Unique names name_i for
  // i ≡ 1 mod 3 suffixes of T
  SA^-1_mod1 := ⟨P[i]_[2]⟩_{0≤i<|P|/2}
  // Unique names name_i for
  // i ≡ 2 mod 3 suffixes of T
  SA^-1_mod2 := ⟨P[i]_[2]⟩_{|P|/2≤i<|P|}
  // Scanning T, SA^-1_mod1, and SA^-1_mod2 we build
  SA0 := ⟨i, T[i], T[i+1], name_{i+1}, name_{i+2}⟩_{i=0+3e∧0≤e<|T|/3}
  SA1 := ⟨i, name_i, T[i], name_{i+1}⟩_{i=1+3e∧0≤e<|T|/3}
  SA2 := ⟨i, name_i, T[i], T[i+1], name_{i+2}⟩_{i=2+3e∧0≤e<|T|/3}
  sort(SA0) by SA0_[2,4]
  sort(SA1) by SA1_[2]
  sort(SA2) by SA2_[2]
  SA := merge(SA0, SA1, SA2)
  return SA

Function merge(S0, S1, S2)
  result := ⟨⟩
  f01 : (i, j) ↦ ⟨T[S0[i]_[1]], name_{S0[i]_[1]+1}⟩ ≤
                 ⟨T[S1[j]_[1]], name_{S1[j]_[1]+1}⟩
  f02 : (i, j) ↦ ⟨T[S0[i]_[1]], T[S0[i]_[1]+1], name_{S0[i]_[1]+2}⟩ ≤
                 ⟨T[S2[j]_[1]], T[S2[j]_[1]+1], name_{S2[j]_[1]+2}⟩
  f12 : (i, j) ↦ name_{S1[i]_[1]} ≤ name_{S2[j]_[1]}
  i0 := 0; i1 := 0; i2 := 0;
  while(i0 + i1 + i2 ≠ |T| + 3)
    if(f01(i0, i1) ∧ f02(i0, i2))
      result.push_back(S0[i0]_[1]); ++i0
    if(!f01(i0, i1) ∧ f12(i1, i2))
      result.push_back(S1[i1]_[1]); ++i1
    if(!f02(i0, i2)∧!f12(i1, i2))
      result.push_back(S2[i2]_[1]); ++i2
  return result
```

only one tuple component for sorting, but the third one two. As a third reason, we should not forget about the fact that *DC3* is assymptotical *I/O* optimal.

The disk space complexity of *DC3* is $\text{sort}_{space}\left(\left(\frac{5}{3}+\frac{4}{3}+\frac{5}{3}\right)N\right)+\text{scan}_{space}(6N)$, because in the merge step, we need the data of three sorters in parallel and we need the stored sub problem data to be simultaneously on the disks.

9.4. A Pipelined DC3 Algorithm With Discarding

For building the subproblem strings of size $\frac{2}{3}\cdot\#\{S\}$, we use a naming step as in our *doubling* derivatives. This prompts us to use the same discarding strategy as for *doubling with discarding*. The algorithm is described in *Algorithm* 10 and a small example follows in *Figure* 9.4.1.

DC3 with discarding is slightly more *I/O* expensive per iteration than the *DC3* algorithm. Since the external memory array H and the first component of the finished tuples in external memory array FT are sorted, we can store them efficiently. Nevertheless, even without data compression, we need in the first step never more than the *I/O volume* of the *DC3* algorithm plus an *I/O volume* of $\text{scan}(3N)$ for H and FT. We also need to encode the information whether a character is finished (only for the sub problems).

The correctness of the algorithm comes from the fact that we pass the name information of the finished tuples (with unique unique names) to the next sub problem. We discard them after we have built the naming triples.

Another question is, whether it is worth trying *discarding* because the subproblems of *DC3* get small very fast. On the other hand, in an iteration step, the longest common prefix of S decreases within the subproblem to one third of its original size. For easier understanding, we will again ignore boundary cases, which comes from different input string sizes.

Algorithm 10 Pipelined DC3+Discarding

```
Function DC3+Discarding(T)
  N := |T|
  f : i ↦ #{T[j] | j < i ∧ (T[j] finished)}
  // Tuples are discarded here
  G := ⟨i, ⟨T[i], T[i+1], T[i+2]⟩⟩ (0≤i<N)∧(i−f(i)≢0 mod 3) ∧(T[i] unfinished)
  I := ⟨e, G[e]_[2]⟩_0≤e≤|G|
  sort(I) by I_[2]
  P := name3(I)
  // Notice that FT is sorted by first component
  FT := ⟨P[i]_[2], e⟩_0≤e≤|G|∧(P[i] unique)
  if ∃i, j : i ≠ j ∧ P[i]_[2] = P[j]_[2] then
    sort(P) by ⟨P[i]_[1] mod 2, P[i]_[1] div 2⟩_0≤i<N
    // Needed for first merge step
    H := ⟨P[i]_[1]⟩_0≤i<|P|∧(P[i] unfinished)
    SA^12_unmerged := DC3 + discarding(P_[2])
    R := ⟨SA^12_unmerged, H[i]⟩_0≤i<|SA^12|
  sort(R) by R_[1]
  P := merge_isa(R, FT)
  // P_[2] is now the inverse suffix array of SA^12
  // Unique names name_i for
  // i ≡ 1 mod 3 suffixes of T
  SA^-1_mod1 := ⟨P[i]_[2]⟩_0≤i<|P|/2
  // Unique names name_i for
  // i ≡ 2 mod 3 suffixes of T
  SA^-1_mod2 := ⟨P[i]_[2]⟩_|P|/2≤i<|P|
  // Scanning T, SA^-1_mod1, and SA^-1_mod2 we build
  SA0 := ⟨i, T[i], T[i+1], name_i+1, name_i+2⟩_i=0+3e∧0≤e<|T|/3
  SA1 := ⟨i, name_i, T[i], name_i+1⟩_i=1+3e∧0≤e<|T|/3
  SA2 := ⟨i, name_i, T[i], T[i+1], name_i+2⟩_i=2+3e∧0≤e<|T|/3
  sort(SA0) by SA0_[2,4]
  sort(SA1) by SA1_[2]
  sort(SA2) by SA2_[2]
  SA := merge(SA0, SA1, SA2)
  return SA

Function merge_isa(P1, P2)
  result := ⟨⟩
  i := 0; j := 0
  while(i + j < |P1| + |P2| − 2) do
    if(P1[i]_[1] < P2[j]_[1])
      result.push_back(P1[i]); i++
    else
      result.push_back(P1[j]); j++
  return result

Function name3(I)
  store := 0
  result := ⟨I[0]⟩
  for i := 1 to |I| − 1 do
    if(I[i]_[2] = I[i − 1]_[2]) store++
    else store = 0
    result.push_back(⟨I[i]_[1], i − store⟩)
  return result
```

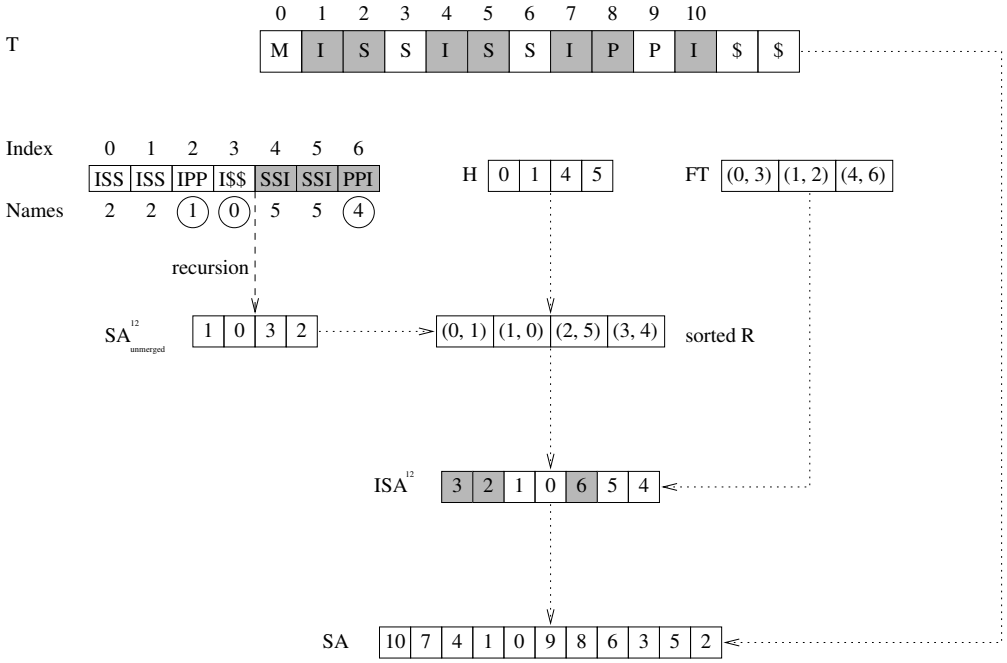

FIGURE 9.4.1. DC3+Discarding example for "MISSISSIPPI"

9.5. Thoughts About A Pipelined DCX Algorithm

For $S \subset \mathbb{Z}_X$, we define $d(S) := \{a - b \mid a, b \in S\}$. If $d(S) = \mathbb{Z}_X$, we call S a *difference cover* of $\mathbb{Z}_X$. We call S a *perfect difference cover* if $X = |S|^2 - |S| + 1$. More details about *difference covers* are described in *Appendix* A.2. *DCX* was first introduced in **[5]** as a lightweight internal suffix array construction algorithm. We use *difference cover* to choose a minimal set of index positions from an input string T, for which we compute names for the merge step of *DC3*. These names make it possible to compare two arbitrary substrings of T, represented by tuples of constant size in a final merge step. For the generalisation *DCX* of *DC3*, we choose a positive integer $X \in \mathbb{N}$ for $\mathbb{Z}_X$ and a fitting *difference cover* S. We can compute from S the set of all sample indices

$$I := \{a \mid (a = i \cdot X + s) \wedge (\bar{s} \in S) \wedge (0 \leq a < |T|)\}$$

of a string T. A concrete example is *DC3* with $X = 3$, $S = \{\bar{1}, \bar{2}\}$, $I = \{1, 2, 4, 5, 7, 8, 10\}$ and $T =$ "*Mississippi*".

We derive the costs of *DCX* from algorithm description of *DC3* (see *Appendix* C.1) and we restrict our considerations to *rings* $\mathbb{Z}_X$, for which we can find a *perfect difference cover* $S \subset \mathbb{Z}_X$. *Perfect difference covers* minimise the *I/O volume* of the naming step of *DCX* and reduce the input string size of its sub problems because the size of S is minimal to cover $\mathbb{Z}_X$.

The second *I/O* expensive part is the merge step, which needs for *DC3* three expensive sorters in parallel. For calculating its cost for *DCX*, we have to know the merge tuple sizes. Unfortunately, no formula is available to compute these. Nevertheless, we can compute them with an algorithm for all relevant

TABLE 1. Sample distance table for the difference cover {0,2,6} over $\mathbb{Z}_7$

$a \setminus b$	0	1	2	3	4	5	6	$\mathrm{dmax}_r(a)$
0	0	6	0	6	2	2	0	6
1	6	1	5	6	5	1	1	6
2	0	5	0	4	5	4	0	5
3	6	6	4	3	3	4	3	6
4	2	5	5	3	2	2	3	5
5	2	1	4	4	2	1	1	4
6	0	1	0	3	3	1	0	3

sizes of X. We denote the optimal average merge tuples size with δ_X. What we know about the merge tuples is that we need one component for the index and $R = |S|$ components[2] for names of substrings T_i with $i \in I$. Thus δ_X is larger than $R + 1$.

For the recursion cost factor, which was 3 for *DC3* we get for *DCX*:

$$\sum_{i \in \mathbb{N}_0} \left(\frac{R}{R^2 - R + 1} \right)^i = \frac{R^2 - R + 1}{R^2 - 2R + 1} = 1 + \frac{R}{R^2 - 2R + 1}$$

The overall *I/O volume* for our simple model of *DCX* is (shown in *Appendix* C.1):

$$\mathrm{sort}\left(\left(\left((R^2 - R + 2) + \delta_X + 2 \right) \cdot \frac{R^2 - R + 1}{R^2 - 2R + 1} - 2 \right) \cdot N \right) + \mathrm{scan}\left(\left(2 \cdot \frac{R^2 - R + 1}{R^2 - 2R + 1} \right) \cdot N \right)$$

To get the optimal *I/O volume,* we have to split the merge tuples in at most X sorters, as we did it for *DC3*. We can estimate the number of needed input characters with an easy algorithm. Let

$$\mathrm{dmax}_r(i) := \max\{k \mid (\overline{i+k} \in S) \wedge (k < X)\}$$

be the *maximal distance to the right* from starting position i to the next sample, then the merge tuple size for the merge tuples with indices $i \equiv e \mod X$ is $\mathrm{dmax}_r(e) + 1 + R$ because we need all R samples to be able to compare a given merge tuple with an arbitrary other merge tuple. Hence the overall average merge tuple size is:

$$\delta_X = 1 + R + \frac{1}{X} \sum_{0 \le i < X} \mathrm{dmax}_r(i)$$

An example for the use of the dmax_r function is shown in *Table* 1 . For $a, b \in \mathbb{Z}_X$ the table entries of *Table* 1 are given by:

$$\mathrm{dist}(a, b) := \min\{k \mid \overline{a+k},\ \overline{b+k} \in S\}$$

[2]For a perfect difference cover S we need all $|S|$ residue classes to cover $\mathbb{Z}_X$.

With help of the δ_x function, we now can compute the exact *I/O volume* cost with respect to our model of *DCX*, as we have done it for the perfect difference covers with $X \in \{3,\ 7,\ 13,\ 21,\ 31,\ 39,\ 57\}$ (see *Appendix* C.3). Since our sorter needs to process twice the *I/O volume* of the scan *I/O volume,* we have the equal *I/O volume* for *DC3* and *DC7*. A reason for choosing *DC3* is for example the smaller disk space complexity.

TABLE 2. *I/O volume* of DCX

$X = R^2 - R + 1$	3	7	13	21	31	39	57
sort $[N]$	30	31.25	43.23	59.93	81.64	102.96	136.7
scan $[N]$	6	3.50	2.89	2.63	2.48	2.39	2.33
Total	66	66	89.35	122.49	165.76	208.31	275.73

Nevertheless, *DC7* might be an alternative to *DC3*. Especially if we use *DC7* in the first step for input data with a small alphabet. If we consider, for example, the genome data with a five character alphabet. We can put in the first naming step all needed characters for a naming tuple in one 32 bit word, instead of using seven words for seven characters. This would save us just for the naming step an *I/O volume* of $\mathrm{sort}(6N)$ for this special case.

CHAPTER 10

Suffix Array Checking

To ensure the correctness of large suffix arrays, we need an external memory suffix array checker. In [**5**] is mentioned a fast checker, which we have adapted to the pipeline structure with *Algorithm* 11.

THEOREM 10.0.1. [**5**] *An array* $SA[0, N-1]$ *of integers is a suffix array of a string* $S[0, N-1]$ *if and only if the following conditions are satisfied:*

(1) In SA is stored a permutation of the numbers $0, \ldots, N-1$.
(2) For all $0 < i < N$, $S[SA[i-1]] \leq S[SA[i]]$.
(3) For all $0 < i < N$, such that $S[SA[i-1]] = S[SA[i]]$ and $SA[i-1] \neq N-1$, there exists $j, k \in \{0, \ldots, N-1\}$ such that $SA[j] = SA[i-1]+1$, $SA[k] = SA[i]+1$ and $j < k$.

Condition one can be checked in linear time while computing an inverse suffix array, which we need in condition three and two, or alternatively can be checked during the construction of P. For condition two, we need to sort the tuple sequence $\langle S[i], SA^{-1}[i]\rangle_{0 \leq i < N}$ with respect to its second component. In condition three, we can compute j and k with $j = SA^{-1}[SA[i-1]+1]$ and $k = SA^{-1}[SA[i]+1]$, because $SA^{-1}[SA[i]] = i$. We can check the last two conditions in one step, if we sort the triples sequence $G := \langle SA^{-1}[i], SA^{-1}[i+1], S[i+1]\rangle_{0 \leq i < N-1}$ by its second component. Note, that there are some special cases left which we have to check separately with few effort (as for example $S[SA[0]] \leq S[SA[1]]$). With the last component of the sorted sequence G, we can check the most cases of condition two because we have computed the ordered sequence $S[SA[1]],\ S[SA[2]], \ldots,\ S[SA[N-2]]$. The first component of the sorted sequence G computes $SA^{-1}[SA[i]+1]$, ordered by i. Thus, we can check also condition three by scanning G.

The additional *I/O volume* costs are only $\text{sort}(5N)) + \text{scan}(N)$, because SA is already computed and we have to read T again.

Algorithm 11 Pipelined Checker For SA

```
Function sa_check(S, SA)
  ck := true
  I := ⟨i, SA[i]⟩_{0≤i<N}
  sort(I) by I_[2]
  P := ⟨I[i]_[1], I[i+1]_[1], S[i+1]⟩_{0≤i<N−1}
  sort(P) by P_[2]
  for i := 0 to N − 2 do
    if P[i]_[3] = P[i+1]_[3] then
      if P[i]_[1] > P[i+1]_[1] then ck := false
    else
      if P[i]_[3] > P[i+1]_[3] then ck := false
  return ck
```

CHAPTER 11

Experiments

We have implemented six suffix array construction algorithms for our experiments, which are *2*-Tupling (doubling), *4-Tupling*, *2-Discarding* (discarding), *4-Discarding*, *doubling without pipelining* and the *DC3*[1] algorithm. The main reasons for our choices are already given in the earlier chapters. Our goal was always to minimise the needed *I/O volume,* but also respecting the internal work. This strategy is effective, because we are CPU bounded with four or more disks. The implementation of *doubling without pipelining* is thought to show, how expensive non pipelined algorithms can be with respect to the *I/O volume*. Tests with a different number of disks will follow for the *DC3* algorithm. Note, that we do not use any of the alphabet increasing techniques, proposed in *Chapter* 12. Furthermore, we only use one of the two CPUs of our test system. The unused CPU has less than one percent work for the operating system during the suffix array construction.

11.1. Our Test System Configuration

For compiling the algorithms, we use *g++ 3.2.3* (-fomit-frame-pointer -O2) on the *GNU/Linux* distribution *debian* with a 2.4 *kernel*. The hardware consists of

- *Dual 2GHz Xeon (of which our sort algorithm uses only one CPU)*
- *1 GB DDR RAM*
- *Intel E7500 Chipset*
- *8 × 80 GB disks with up to 375 MByte/s total bandwidth*

For more detailed informations about the test computer see **[10]**. For the most of the tests we use only four disks, because some of the algorithms only take small advantages of more disks. Since *DC3* is the algorithm with the smallest *CPU* load (cheap comparison functions), we will test it with up to 8 hard disks. For sorting, we use the pipelined external merge sort algorithm from the STXXL external memory library of Roman Dementiev **[9]**.

11.2. Characterising Our Input Data Instances

Our input data is real world data as English text, html pages, genome data and open source code. We run our algorithms with power of two prefixes of these data instances, starting with strings of size 2^{16} bytes. We also have bad case input instances for all powers of two.

[1] *DC3* is also known as skew algorithm.

Gutenberg:

The *Gutenberg*[2] text file was build from most of the English texts of the "*Project Gutenberg*". We have filtered out some larger files, which were mostly genome data. As the table entry for the *average lcp* shows, we have long repetitions in the file.

Genome:

The *Genome*[3] data is a snapshot of the human chromosomes from may 2004. The very long repeats come from not yet analysed parts of the chromosomes, which were replaced with a special character as placeholder for one of the four other characters in the alphabet.

HTML:

The *.gov* html data is a collection of html pages. This collection consists of pure html pages from the .gov domain, without pictures or binary files. As for all other test data, the pages are simply concatenated together. The complete file was only tested with the *DC3* algorithm, because of the $2^{31}-1$ input character limit of *X-Discarding* and the slowness of the other algorithms.

Open Source:

We also have concatenated the source code of several *Open Source* projects together (coreutils, gcc, gimp, kde, xfree, emacs, gdb, linux kernel and open office). Note, that we took only one version of each project and that we took only the pure source code from these projects, which were mostly written in *C++*.

Bad Case:

The *Bad Case* data consists for each input size of two times the same random string concatenated. We will show later how to preprocess a string online to reduce the number of iterations, without additional *I/O volume*. There is a nice strategy, which can take advantage from repeating characters and for which a string of equal characters (the normal worst case) needs only one iteration. For generating the random string, we took an implementation of the Mersenne twister prime number generator **[25]**, which has a period of $2^{19937}-1$.

Table 1 shows us different values, which can tell us how hard the data for our algorithms is. From maxlcp, we can compute the required number of iterations for *X-Tupling* and *X-Discarding* algorithms. Also the difference between $\overline{\text{lcp}}$ and $\overleftrightarrow{\text{lcp}}$ can tell us about the repetition of long substrings in the input string. With loglcp_b, we can estimate the *I/O volume* of the *X-Discarding* algorithms. We have computed loglcp_b for $b = 2$, but one can change the base, since we do not round the sum. *Figure* 11.2.1 shows the loglcp_2 data for all tested data instances. We have implemented the linear time LCP construction algorithm from **[20]** to compute the LCP values for the two tables. We have started the algorithm on computer with $184GB$ of internal memory, of which we have used at most $52GB$. The reason why the *Gutenberg* data has the largest loglcp_2 values is, that several files appearing more than once. We also explain the

[2] http://gutenberg.net/list.shtml

[3] http://genome.ucsc.edu/downloads.html

TABLE 1. Test data informations

Text	$\lvert\Sigma\rvert$	*Chars*	$\overline{\text{lcp}}$	$\overleftrightarrow{\text{lcp}}$	maxlcp	loglcp_2
Gutenberg	~128	3277099765	45617.2	86405.6	4819356	10.34
Genome	5	3070128194	454111	455269	21999999	6.53
Html (.gov)	~128	4214295245	1108.1	1880.4	102356	6.99
Open Source	~128	547505710	431.1	592.1	173317	5.80
Bad Case	~128	$2^{32}-1$	$\approx 2^{29}$	$\approx 2^{29}$	$\approx 2^{31}$	≈ 29.56

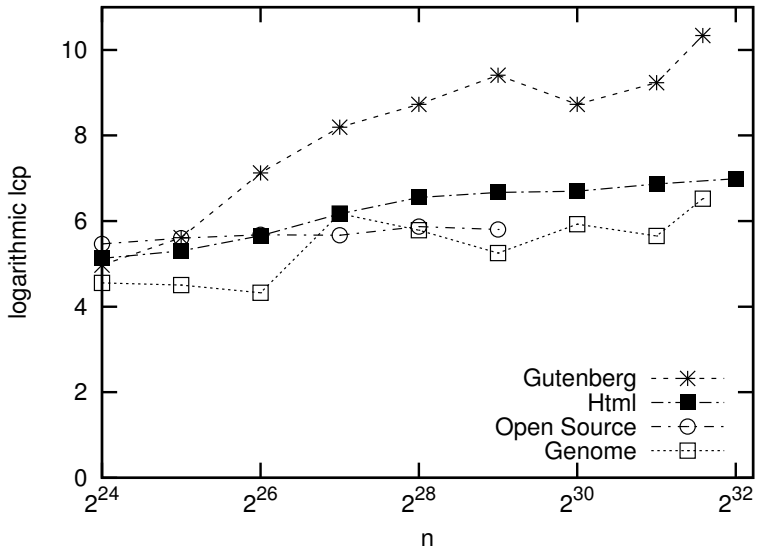

FIGURE 11.2.1. Loglcp values for the test data

"zigzag" behaviour of the genome data (see Figure 1), which is caused by long sequences of not yet analysed chromosome partitions. The analysed chromosome partitions behave like random data, in the sense of data compression. Hence the discarding algorithms can discard most of the suffixes very fast.

11.3. Results For The Non-Pipelined Doubling Algorithm

The reason, why we have implemented a non-pipelined doubling algorithm is that we wanted to study the impact of pipelining for external memory algorithms. For the non-pipelined doubling, we use all tricks from pipelined doubling. Not using pipelining makes one loop step such expensive, with respect to the *I/O volume,* that an iteration raises the costs massively. An example is the *Gutenberg* data in *Figure* 11.3.1, which needs from fourteen to nineteen iterations. The non-pipelined doubling is easier to implement because we can use a simpler loop structure and we have no overhead on code for the pipelining interface. We can also leave the loop after a scan step because scan steps are not forced to pipe their data in the next sorter. For the external merge sort this means a waste of a run formation phase (but not merging the sorted runs

of size M). The *I/O volume* of non-pipelined doubling is in a loop iteration about $\text{sort}(5N) + \text{scan}(20N)$, which is an *I/O volume* of about $30N$, because an non-pipelined sorter must read and write the data twice and the scan steps only have to read and store arrays. Since we *are CPU* bound with four disks and the doubling with and without discarding have nearly the same number of comparisons, the per element time is close together in the time graphs of *Figure* 11.3.1. The reason, that in the measurements the running time and the *I/O volume* for $n < 2^{27}$ are rather small is explained by the fact that the sort algorithm of STXXL sorts data internally if data fits in the internal memory. The non-pipelined doubling for example is affected by this in the first two points in the graph. We use in this case 256 MByte of main memory for each sorter and the two sorters in the loop have for 16 MByte input data a sorting volume of 128 MByte and 192 MByte. Thus the pipelined sorter needs only to read and store the data one time. For 32 MByte input data only one of the two sorter in the loop can sort internally. Everything larger must be sorted externally.

11.4. Results For The X-Tupling Algorithm

Pipelined doubling is the first algorithm which uses the pipelining model. If we compare one loop iteration with an iteration of non-pipelined doubling, we count only an *I/O volume* of $\text{sort}(5N)$, which is with an I/O volume of $10N$ if a pipelined sorter is used. A pipelined sorter only needs to read and write the data once. If we consider the input data with more than $\geq 2^{26}$ characters in the *I/O volume* graphs, we can see this advantage of pipelined doubling in most of the graphs in *Figure* 11.3.1. The STXXL pipelined sorters also sort internally if we give the sorters enough internal memory for a given input instance. As mentioned before, the last iteration wastes one store/read step. If the last scan step notices the last iteration, the data is already piped in the next iteration. This means a waste of a storing/reading pass and also a waste of comparisons, because the written data is partially sorted in blocks of size M. Nevertheless pipelined doubling always has a smaller *I/O volume* than the non-pipelined one, if the sorters sort externally.

If we compare *2-Tupling* (doubling) and *4-Tupling* (quadrupling) with respect to our optimisation function $f(x) = \frac{3+x}{\ln x}$, *4-Tupling* has 30 percent *I/Os* less.

11.5. Results For The X-Discarding Algorithm

With its slightly higher *I/O volume* per iteration with respect to *X-Tupling*, *X-Discarding* has its problems with the *Bad Case* data, that has very long lcp values and a small number of tuples to discard per iteration. If the *I/O volume* of *2-tupling* is $\frac{5}{6}$ of the *I/O volume* of *2-Discarding* per iteration and if there are no tuples to discard, this advantage shrinks between *4-Discarding* and *4-tupling* (*4-Tupling* needs $\frac{7}{8}$ of the *I/O volume* of *4-Discarding*). Since *X-Discarding* has more complex scan pipes and needs to compare triples in the second sorter, it consumes more time per element. The CPU load is our bottleneck in most of the cases. The good news are, that discarding works well for real world data. The good performance of *X-Discarding* on Genome data can be explained by the fact

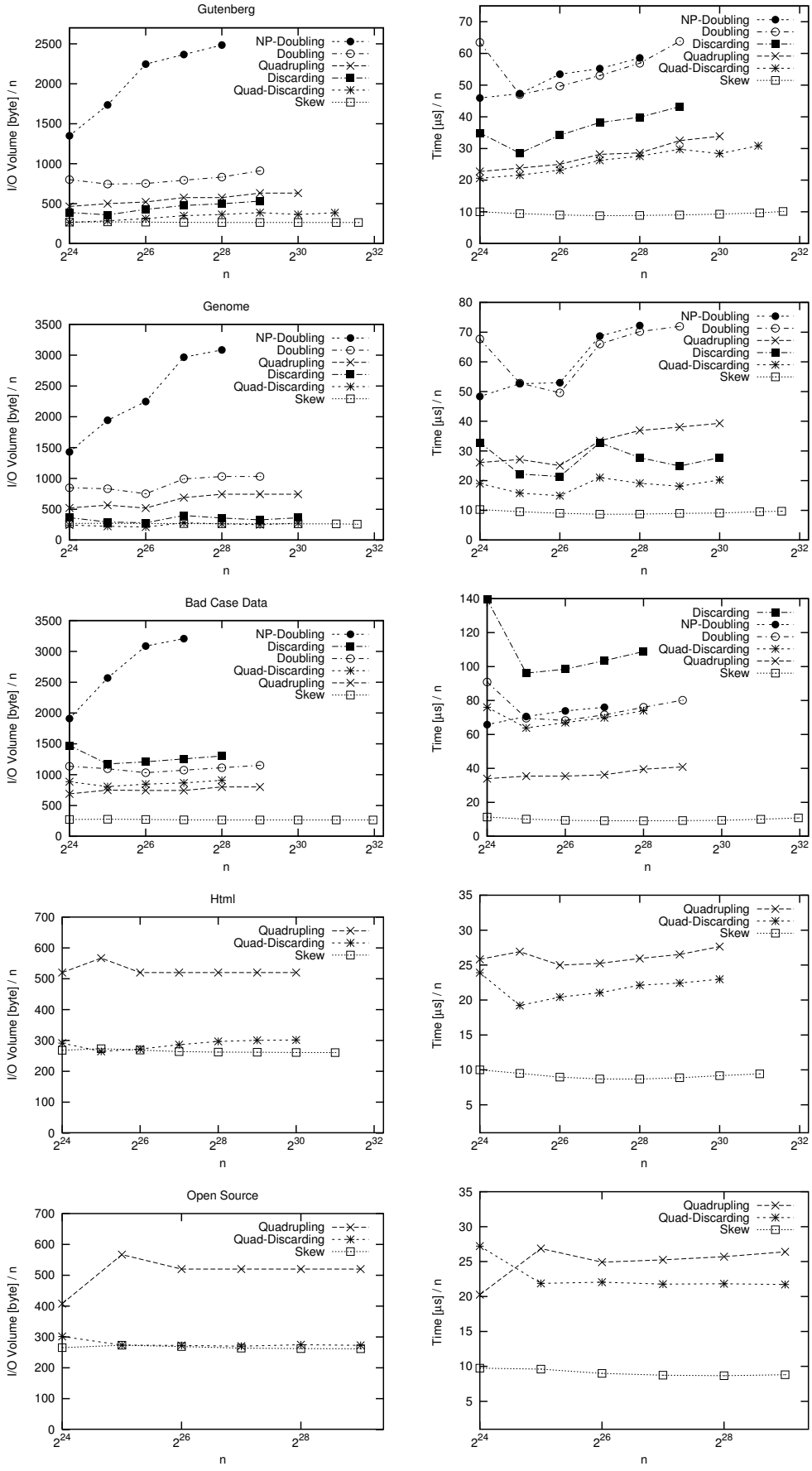

FIGURE 11.3.1. Test Data Results

that for Genome data the $\overleftrightarrow{\text{lcp}}$ value is much less than the maxlcp value. Without the large gaps of undetermined parts in the *Genome* data, the rest fragments behave almost as random data (hardly to compress with known algorithms). For *X-Discarding,* this means early discarding of most the suffixes. Even for the *Gutenberg* data, where many files appearing more than once, the algorithm is better than the *X-Tupling* variants.

11.6. Results For The DC3 Algorithm

On the first glance the *DC3* algorithm seems to be the lowest curve in each graph. If we look at the Genome data for the data with 2^{26} elements, discarding is slightly better. If we consider *DC3* as a discarding algorithm, it discards in each iteration about one third of its elements. Since most of the data behaves as random data, *X-Discarding* can discard elements very fast if there are no larger repetitions. Unlike the doubling derivates, the work of the *DC3* algorithm is predictable very precisely for all kinds of input data. The reason of this is that most work is done in the first few iteration (in the pseudo-code recursions). If we compare our *Bad Case* data with the *Html* collection, which has the smallest maxlcp, we measure 265 bytes *I/O volume* per element versus 261 bytes *I/O volume* per element (compared 512 MByte input data). The Html collection has seven iterations more than the *Bad Case* data and also the running time shows a similar behaviour. Another interesting point is that *DC3* needs less internal work than other algorithms, especially in the *Open Source I/O volume* graph. *4-Discarding* needs there nearly the same *I/O volume,* but *DC3* is more than two times faster in every case. Even if we would halve the *I/O volume* and the time per element of the *4-tupling,* the *DC3* would be faster. The reason for this is, that *DC3* does very cheap comparisons in its sorters and during the merge step. The most expensive comparison are done during the naming in the first loop, where at most three elements of a four tuple are compared. Otherwise, all comparisons between five or four tuples are with respect to one or two elements. Because of this, we measure the speedup of the DC3 if we use parallel disks.

TABLE 2. Speedup of the *DC3* algorithm with one or more disks

#Disks	1	2	4	6	8
$\mu s/byte$	13.96	9.88	8.81	8.65	8.52

The table shows us once more, that DC3 is almost CPU bound. It also shows us, that even using a single disk we can compute a large suffix array on a cheap computer in reasonable time. A possibility to reduce the running time further would be to reduce the total number of comparisons, which might be possible with *DCX* for $X > 3$.

CHAPTER 12

String Preprocessing

We noticed especially for the *discarding* algorithms that most the work is done in the first few iterations, which means that discarding works not very well in this case. The first reason for this is, that real world data has small alphabets (genetic alphabet size is 4 and English text with special characters has around 128 characters). Another reason is, that the naming strategy start very slowly and computes only names for small suffixes. If we consider, for example, the sentence start $T :=$"In the past", we get around 10.5 million hits in a web search engine. Thus, if we want to build a suffix array from a large text it is likely that we need at least four *doubling* iterations (i. e. the computed names represent 16 characters) to compute a unique name for a suffix starting starting with T.

Since computer arithmetic work fast for word size (in our case 32 bit), we always use words no matter how many bits the names need. It would be optimal, if we had a 32 bit alphabet in the first iteration. How to increase the alphabet size of a input string and thus to increase the information encoded in an character is the topic of this chapter.

12.1. Alphabet Increasing By Concatenating Characters

An easy way to increase the alphabet size is to replace each character by itself, followed by the next characters (or some bits of these). For example, we have the string $S = a \circ b \circ b \circ a \circ \$$ and transform it into $T = ab \circ bb \circ ba \circ a\$ \circ \$\$$. We have increased the alphabet size from three to five and all characters in T are already unique, whereas in S only the end symbol is unique. It is easy to see, that we can do this transformation while reading the input, without increasing the *I/O volume*. The advantage of this attempt is that we can skip, for example for doubling, the first two iterations, if we have an eight bit alphabet.

THEOREM 12.1.1. *The strings S and T have the same suffix array , if T is a transformation of the following kind: $T[i] = S[i] \circ S[i+1]$ for all i.*

PROOF. Let $S_i = A \circ b_1 \circ C_1$ and $S_j = A \circ b_2 \circ C_2$ $(i \neq j)$ be two suffixes of S with $S_i < S_j \Leftrightarrow b_1 < b_2$. Then substring A is the common prefix both suffixes, b_1 and b_2 denotes the first distinct characters and C_1 and C_2 the rest of the suffixes. If we now compare the suffixes $T_i = A' \circ b_1' \circ C_1'$ and $T_j = A' \circ b_2' \circ C_2'$ (with analogous notation) of the transformed string T, we know that $\#\{A\} = \#\{A'\} + 1$ and we have to compare only the strings $b_1' = a' \circ b_1$ and $b_2' = a' \circ b_2$, if a is the last character of A. Since a' is common in b_1' and b_2' , the relation between b_1' and b_2' is the same as between b_1 and b_2. □

It is easy to see, that we even can take bits of the following characters to get a larger alphabet (simply interpret our string as a one bit alphabet).

12.2. More Efficient Alphabet Increasing By A Base Change

A disadvantage of simply attaching characters together, stored as binary numbers, is that we waste alphabet range. More precisely, we waste with an alphabet Σ of size $n := |\Sigma|$ $m = 2^k - n$ $(k \in \mathbb{N} \wedge n \leq 2^k)$ spaces to encode further alphabet characters, if we store a character in a k bit word. Assume now, that we have a string S of size r over the alphabet Σ. We can encode S with an integer $\sum_{i=0}^{r-1} c_i n^{r-1-i}$, where $c_i \in \mathbb{N}_0 \wedge 0 \leq c_i < n$ encodes a character in Σ. If we want to encode the string $T :=$ "*edcba*", for example , we need a five character alphabet $\Sigma := [\ a,\ b, c, d, e] = [0,\ 1,\ 2,\ 3,\ 4]$, but the next power of two is eight. Thus, we need five times three bits to encode T with the method described in *Chapter* 12.1. The encoding of T with base five is $4 \cdot 5^4 + 3 \cdot 5^3 + 2 \cdot 5^2 + 1 \cdot 5^1 + 0 \cdot 5^0 = 2500 + 375 + 50 + 5 = 2930 = \ldots + 2^{11}$ and needs 11 bits instead of 15 bits to encode T.

12.3. Alphabet Increasing By Using Global Information

From [**22**] we can derive an interesting way to increase the alphabet size in a global manner and again without further *I/Os*. In this method we can separate all suffixes T_i, which start with the same first character, in two classes, which are the "larger ones" L and the "smaller ones" S. A suffix T_i is in the class L, if it is larger than T_{i+1}. Otherwise, it is in the class S. *Theorem* 12.3.2 gives the idea of how to use this separation. If character c (or the suffix beginning at this position) is of type L, we replace the character by $c \circ 0 = c'$, else by $c \circ 1 = c'$.

THEOREM 12.3.1. *For a type L (or type S) suffix $T_i = a^p \circ T[i+p] \circ T_{i+p+1}$ with $T[i] \neq T[i+p]$ all suffixes T_e with $i \leq e < i+p$ are also of type L (or type S).*

PROOF. Let T_i be of type L. If $p = 1$, we have $T[i] > T[i+1] = T[i+p]$ because of the definition of type L suffixes. For $p > 1$ all suffixes T_e with $i \leq e < i+p$ are of type L because $T_e > T_{e+1} \Leftrightarrow a^e > a^{e-1} \circ T[i+p]$. Let T_i be of type S. If $p = 1$, we have $T[i] < T[i+1] = T[i+p]$ because of the definition of type S suffixes. For $p > 1$ all suffixes T_e with $i \leq e < i+p$ are of type S because $T_e < T_{e+1} \Leftrightarrow a^e < a^{e-1} \circ T[i+p]$. □

THEOREM 12.3.2. *(Lemma 2 from* [**22**]*): A type S suffix is lexicographically larger than a type L suffix if their first characters are equal.*

PROOF. See [**22**] or our version. Consider the the type S suffixes $T_s = a^p \circ T[p+s] \circ T_{p+s+1}$ and type L suffixes $T_l = a^q \circ T[q+l] \circ T_{q+l+1}$ $(p, q > 0)$ with $a \neq T[p+s]$ and $a \neq T[q+l]$. From the proof of *Theorem* 12.3.1, we know that $T[q+l] < a < T[p+s]$. For the three comparism cases $p > q$, $p = q$ and $p < q$, we get $T_s > T_l \Leftrightarrow a^{q+1} > a^q \circ T[q+l]$, $T_s > T_l \Leftrightarrow T[p+s] > T[q+l]$ and $T_s > T_l \Leftrightarrow a^p \circ T[p+s] > a^{p+1}$. □

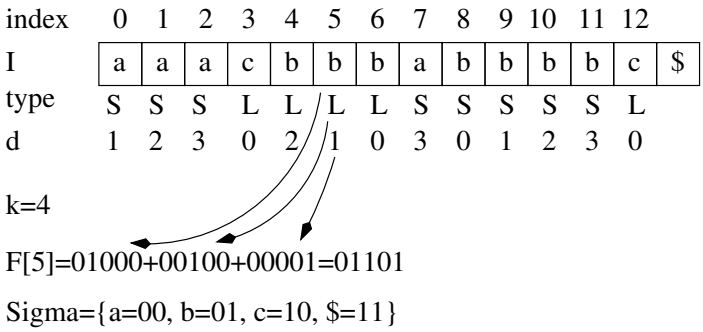

FIGURE 12.4.1. Alphabet increasing example for the string "aaacbbbabbbbc". The variable d denotes for T_i of type L the value $T[i] - \alpha \cdot k$ and for T_j of type S the value $T[i] - \beta \cdot k$.

Now the choice of the new c' is clear. If the substrings with same first character of type S and type L are nearly equally distributed, we have a good chance to double the information with just one more bit.

12.4. Alphabet Increasing By Reducing Repeating Characters

Input data with long series of repeating characters is bad case data for doubling like algorithms. It would be nice, if we simply could assign unique numbers (within the series) to the elements in this series to distinguish them from each other. *Theorem* 12.3.2 and *Theorem* 12.3.1 gives us information related to the order of two suffixes with equal leading character. The idea of this alphabet increasing strategy is to put this information in the leading character of each suffix.

Let I be a character string over an integer alphabet Σ, $i,\ j \in \mathbb{N}$, $a \in \Sigma$, $I_i = a^p \circ I[i+p] \circ I_{i+p+1}$ and $I_j = a^q \circ I[j+q] \circ I_{j+q+1}$ are suffixes of I. We use *Theorem* 12.3.2 to separate them in two classes. The substrings, which are of type L and the substrings of type S.

If I_i is of type L, then the substrings $I_i = a^e \circ I[i+e] \circ I_{i+e+1}$ with $e \in \{1 \dots p\}$ are also of type L, what follows from *Theorem* 12.3.1 (Analogously are all $I_j = a^e \circ I[j+e] \circ I_{j+e+1}$ with $e \in \{1 \dots q\}$ of type S).

If I_i and I_j are of different type, we know that either $I_i < I_j$ or $I_i > I_j$ holds. We can encode this order information in a new least significant bit by replacing the characters a of I_i by $\alpha := a \cdot 2 + 1$, if I_i is of type L, else by $\beta := a \cdot 2$. Doing this for all suffixes of I, we get a new string T with suffixes $T_i = \alpha^p \circ T[i+p] \circ T_{i+p+1}$ for I_i of type L, $T_j = \beta^q \circ T[j+q] \circ T_{j+q+1}$ for I_j of type S and if I_i and I_j have the same leading character.

Further, we choose an integer $k = 2^c$ with $c \in \mathbb{N}$ and replace $T[i]$ of all T_i of type L by $\left\{ \begin{smallmatrix} \alpha \cdot k + (p-1), \text{ if } p<k \\ \alpha \cdot k + (k-1), \text{ if } p \geq k \end{smallmatrix} \right\}$ and $T[j]$ of all T_j of type S by $\left\{ \begin{smallmatrix} \beta \cdot k + (k-q-1), \text{ if } q<k \\ \beta \cdot k, \text{ if } p \geq k \end{smallmatrix} \right\}$ to get a new string F. Note, that F has the same suffix array as S because of *Theorem* 12.3.2 and *Theorem* 12.3.1. *Figure* 12.4.1 shows an example of this technique.

CHAPTER 13

Summary And Conclusion

We have adapted recent suffix array construction algorithms to external memory and optimised them. Therefore, we first have theoretically determined how to minimise the *I/O volume* of the algorithms for arbitrary input instances and then we have chosen a tradeoff between the optimal *I/O volume* and the possibility to use fast CPU arithmetic. All of our algorithms only need a small amount internal memory. The memory is mostly consumed by the used sort algorithms. For example, we have computed a large suffix array from $\approx 4GByte$ input data over a weekend using only $< 1MByte$ of internal memory.

In *Chapter* 7, we have improved the *doubling* algorithm [**3**] in three ways. The first improvement is a special comparison function for sorting, which eliminates all scan *I/Os* in the *doubling* algorithm. Secondly, we have designed a pipelined version of doubling, that reduces the *I/O volume* to about 30% of the *I/O volume* of non-pipelined doubling algorithm. As the third improvement, we have generalised doubling to *X-Tupling* and we have computed the optimal value for the parameter X. The result is a reduction of the *I/O volume* to about 70%[1] of the *I/O volume* of the pipelined doubling algorithm.

The *discarding* algorithm [**7, 1**], described in *Chapter* 8, allows to reduce the *I/O volume* in each iteration of the algorithm. Since the structure of *discarding* is similar to *doubling,* we apply all optimisations known for *X-Tupling* also for the generalisation of *discarding* called *X-Discarding.* We get similar improvements for the *I/O volume.* As the *X-Tupling* algorithm, *X-Discarding* is asymptotically suboptimal but there are input instances for which the *I/O volume* of *4-Discarding* is smaller than the *I/O volume* of the *I/O* optimal *DC3* algorithm. The reason is that one *X-Discarding* iteration is cheaper than the first recursion step of the *DC3* algorithm. This happens for smaller data instances with small loglcp_2 (*Open Source* and the *Genome* data).

In *Chapter 9,* we present a pipelined version of the *DC3* algorithm [**18**], which is an asymptotically *I/O* optimal suffix array construction algorithm. According to our tests it is the fastest one and also has the smallest *I/O volume* for most of the input instances we have tested. Since the most expensive part of *DC3* are the first few recursion steps (about 70% work is done in the first three steps), it has a similar *I/O volume* for all input data with equal size and alphabet. Hence, we can predict the time for constructing a suffix array with a good precision. We also introduce a description of the *DC3* algorithm combined with discarding. The algorithm can be of interest for input data for which *X-Discarding* works more efficiently than *DC3.*

[1] If the input data is large enough.

We have designed a pipelined external memory suffix array checker based on a theorem in [**5**].

Most of our input data has small alphabet size with respect to the word size of nowadays CPUs[2]. Thus we have introduce in *Chapter* 12 online methods for increasing the alphabet size of strings, without additional *I/Os*. Using one of the alphabet increasing techniques, we can encode more than 13 characters of a *Genome* string in one 32 bit CPU word. Nevertheless, we have not done this optimisation for our tests.

[2]The *Genome* data, for example, has an alphabet size of five, but our CPU word size is 32 bit.

CHAPTER 14

Future Work

Firstly, we should combine the alphabet increasing techniques with discarding algorithms, what also means to implement a *DC3* version with discarding. Especially the *Genome* data would benefit from this improvement because most of the data would be discarded in the first iteration.

Note, that our algorithms currently can only handle input strings of size $< 2^{32}$ with ≤ 32 bit alphabets. Most of our input instances need the whole 32 bit range because of their size in characters. Our *Html* instance was only a $\approx 4GByte$ part of a larger collection and it is also thinkable to build suffix arrays from a concatenation of different genome instances. Thus it might be nessessary to extend the algorithms to support larger than 32 bit alphabets in future. Modern *CPUs* have 64 bit machine word size, what might be a good choice for the supported alphabet size.

Since our algorithms are *CPU* bound with four disks on our test system, we should make our implementations more scalable. This means to make it possible to choose the X parameter for *X-Tupling*, *X-Discarding* and *DCX*. On our test system, a larger and thus non optimal X would probably take benefit from more than four disks. Another point is, that some of our sorters only need to sort tuples with respect to their indices, which are a permutation of $0, \ldots, N-1$. This is a special case in sorting and should be optimised. Since the lion's share of the suffix array construction work is done by the sorters, we also should consider to parallelise them.

Our input data instances consist of $\approx 2^{29}$ to $\approx 2^{32}$ characters. Thus, we compute names, which use nearly the whole 32 bit range of our word size. Hence, we have not wasted many bits during the name computation[1], what would be the case for 64 bit words with the same data. Especially for machines with slow *I/O*, it makes sense to compress[2] the data, what needs again CPU time. In the *run formation* node, our sorter stores sorted sub sequences to disk. In our case, we sort tuples with respect of one or more tuple components. In many cases, we have to store a sorted sequence of tuple components $s_0 < s_1 < \ldots < s_{N-1}$. A way to save *I/O volume* is to save the compressed[3] sequence $s_0, (s_1 - s_0), \ldots, (s_{N-1} - s_{N-2})$. In our case this can be done with at least one tuple component of the tuples of each sorter.

[1] Besides the missing alphabet increasing techniques in our tests.

[2] For example putting two 32 bit values in one 64 bit word.

[3] Smaller values get a cheaper bit encoding.

APPENDIX A

Foundations

A.1. Symbol Table

θ	*empty string*
Σ	*string alphabet*
\$	smallest alphabet element in Σ
$S := S[a,\ b]$	*string* with $b - a + 1$ characters, that begins/ends at position a/b
$S[c, d]$	*substring* of $S[a,\ b]$ if $a \leq c < d \leq b$
S_i	substring of $S[a,\ b]$ with $S_i := S[i,\ b]$
$S[i]$	*i-th element* of an indexed structure S
SA	*suffix array* of S
SA^{-1}	*inverse suffix array* ($SA^{-1}[j] = i$ if $SA[i] = j$)
$\#\{A\}$	size (*cardinality*) of a structure A (for ex. a set)
$\|A\|$	alternative for the *cardinality* or the value of a structure A
$A \circ B$	concatenation of structure A with structure B
$x \equiv a\ (b)$	same as $x \equiv a \mod b$
$(A_i)_{i \in B}$	a shortcut of $\|B\|$ different structures A_i where B is an arbitrary index set
$\langle A_i \rangle_{i \in B}$	the same for sequences.
$\langle a_{1,i}, \ldots, a_{x,i} \rangle_{0 \leq i < N}$	is a x-tuple sequence of size N
$S_{[i_1,\ldots,i_t]}$	if $S = \langle a_{e,1}, \ldots, a_{e,x} \rangle_{0 \leq e < N}$ then $S_{[i_1,\ldots,i_t]} := \langle a_{e,i_1}, \ldots, a_{e,i_t} \rangle_{0 \leq e < N}$ for $t \leq x$ ($i_r \in \{1, \ldots, x\}$ for $1 \leq r \leq t$)
$\mathrm{lcp}(i,\ j)$	is the value of the *longest common prefix* of two substrings $S_{SA[i]}$ and $S_{SA[j]}$ ($i < j$). For $j > N - 1$ and $i < 0$ we define $lcp(i, j) := 0$.
LCP	*longest common prefix array* build from the suffix array SA and its source string S. If $\|S\| = N$ we have $LCP[i] := lcp(i, i + 1)$ for $0 \leq i < N$. Otherwise $LCP[i] = 0$
maxlcp	is the *maximum lcp* $max_{0 \leq i < N} lcp(i, i + 1)$ of a string S
maxlcp_i	$max_{0 \leq j < N,\ i \neq j} lcp(i, j)$

$\overline{\text{lcp}}$	is the *average longest common prefix* defined as $\frac{1}{N-1}\sum_{0\leq i<N-1} lcp(i,i+1)$
$\overleftrightarrow{\text{lcp}}$	is the *left right average longest common prefix* defined as $\frac{1}{N}\sum_{0\leq i<N} \text{maxlcp}_i$
loglcp_b	is the *logarithmic average lcp* with base b defined as $\frac{1}{N}\sum_{0\leq i<N} \log_b(1+\text{maxlcp}_i)$
$\mathbb{Z}_n$	The residue class *ring* $\mathbb{Z}_n := \{\overline{k} \mid a = k+s\cdot n = \overline{k} \wedge k < n \wedge k,\ s,\ a,\ n \in \mathbb{N}_0\}$ for all $n > 1$.
$\mathbb{Z}_p$	A well known *field* is $\mathbb{Z}_n$ for $n = p$ is a prime number.

A.2. Difference And Sum Cover Basics

For $S \subset \mathbb{Z}_n$ we define $d(S) := \{a - b \mid a, b \in S\}$ and $s(S) := \{a + b \mid a, b \in S\}$. We call $d(S)$ a *difference cover* iff $d(S) = \mathbb{Z}_n$ and analogously for $s(S) = \mathbb{Z}_n$ a *sum cover* for $\mathbb{Z}_n$. A *difference cover* $d(S)$ with $n = |S|^2 - |S| + 1$ is called *perfect difference cover* since this means that all differences in $d(S)$ have a unique results, if the result is not equal to zero. Analogously for a *perfect sum cover* we have $n = \frac{1}{2}|S| \cdot (|S| - 1) + |S|$. Note, that for perfect *sum/difference cover* $|S|$ is minimal (simply count all possible unique sums and differences possible with $|S|$ elements).

THEOREM A.2.1. *Let S be a difference cover of $\mathbb{Z}_n$ then $\forall a, b \in \mathbb{Z}_n$ there exist $t \in \mathbb{Z}_n$ with $(a + t), (b + t) \in S$.*

PROOF. Let $a = b + c$ with $c \in \mathbb{Z}_n$ then there exists $i, j \in S$ with $c = i - j \iff i = j + c$ because of the definition of *difference covers*. Hence there must be a $t \in \mathbb{Z}_n$ with $a + t = i$ and $b + t = j$ □

The next theorem shows how to construct a combined difference and sum cover.

THEOREM A.2.2. *For all $n \in \mathbb{N}$, $n > 1$ there exists an $S \subset \mathbb{Z}_n$ with $|S| \leq 2\sqrt{n} - 1$ which is a difference and sum cover.*

PROOF. Let $R := \{0, \ldots, r\}$ $(r < n)$ and $M := \{\overline{a} \mid a = r + x(r + 1) \wedge a < n \wedge 0 < x\}$ then $S := R \cup M$ is a *difference cover* and also a *sum cover* (simply fill the gaps of M by adding (for *sum covers*) or subtracting (for *difference covers*) elements of R). $|M| = \lfloor \frac{n-|R|}{r+1} \rfloor = \lfloor \frac{n}{r+1} - 1 \rfloor$ If we count the elements in S we get:

$$r + 1 + \lfloor \frac{n}{r+1} - 1 \rfloor \leq r + \frac{n}{r+1}$$

From $f(r) := r + \frac{n}{r+1}$ and its derivative $f'(r) := 1 - n(r + 1)^{-2}$, we can calculate now the optimal r to minimise $|S|$.

$$\begin{aligned} & 1 - n(r+1)^{-2} = 0 \\ \iff \quad & (r+1)^2 = n \\ \iff \quad & r = \sqrt{n} - 1 \end{aligned}$$

The theorem is proven by $f(\sqrt{n} - 1) = \sqrt{n} - 1 + \frac{n}{\sqrt{n}} = 2\sqrt{n} - 1$ □

A better estimation for pure difference covers was shown by C. J. Colbourn and A.C.H. Ling in **[6]**.

LEMMA A.2.3. *For any N there exists a difference cover of size $\leq \sqrt{1.5N} + 6$.*

APPENDIX B

Implementation Issues

A significant part of the work for this diploma thesis is about 13000 lines of source code, written in the C++ programming language. Since before this point, we only have scratched the implementation issues, and because some readers might be interested in using the pipelining in their own code, we try to summarise basic code structures and some repeating code patterns, appearing in the suffix array construction algorithms.

B.1. Generator Pipeline Nodes

Assume, that we want to feed a simple series in several pipelines, which have reserved an input pipe for this purpose. The way to do this is to use generator pipeline nodes, which were mostly used in our code for generating indexes of tuples. The output of the pipeline node matter_source is a seven bit random number.

```
class matter_source{
public:
  typedef unsigned int value_type;

private:
  value_type result;

public:
  matter_source(){
    result = rand();
  }

  const value_type& operator*() const{
    return result;
  }

  matter_source& operator++(){
    result = rand() & 0x0000007f;
    return *this;
  }

  bool empty() const{
    return false;
```

```
  }
};
```

B.2. Naming Pipeline Nodes

Naming is needed in all of our suffix array construction algorithms with two different naming methods, which are naming for doubling and for discarding.

- The incoming four tuples $\langle index,\ name1,\ name2,\ name3\rangle$ are sorted with respect to the last three components
- The outgoing tuples $\langle index,\ new\_name\rangle$ are sorted by their second component and new_name is a name for $\langle name1,\ name2,\ name3\rangle$

Since the incoming tuples are sorted with respect to the name components, we just have to keep track of the last two input tuples, we call them *tmp* and *tmp2*. If they have equal name components then the new computed name is *counter*, else the new name for *tmp* is *counter* but for *tmp2* it is *counter+1*. Note, that we hide the type definitions in the name space skew_config, because the tuple definitions, we need from there, are only obvious generalisations of the STL pair[1].

```
// skew_quad type is a four tuple
// skew_tuple_type is a stl pair

using namespace skew_config;

 /* Check, if last two components of two quads are equal */

bool quad_eq(const skew_quad_type& a, const skew_quad_type& b){
  return (a.second==b.second)&(a.third==b.third)&(a.fourth==b.fourth);
}

/* Naming for DC3 (also called as skew algorithm) */

template<class Input>
class conv_naming{
public:
  typedef skew_tuple_type value_type;

private:
  Input& A;
  unsigned int counter;
  skew_quad_type tmp,tmp2;
  skew_tuple_type result;

public:
  conv_naming(Input& A_):A(A_), counter(0){
    assert(!A.empty());
```

[1] http://www.sgi.com/tech/stl/pair.html

```
    tmp=*A;
    result.first=tmp.first;
    result.second=counter;
  }
  const value_type& operator*() const{
    return result;
  }

  conv_naming& operator++(){
    assert(!A.empty());
    ++A;
    if(!A.empty()) tmp2=*A;
    if(!quad_eq(tmp,tmp2)) ++counter;
    result.first=tmp2.first;
    result.second=counter;
    tmp=tmp2;
    return *this;
  }

  bool empty() const{
    return A.empty();
  }
};
```

The following code fragment is not part of a finished algorithm, but it can replace, for example, the introduced naming pipe *conv_naming*. The difference to that naming pipe is that the name component of *unique* names in an outgoing tuple $\langle index,\ new\_name \rangle$ represents the final position of the suffix T_{index} in the suffix array, if T is our input string.

```
/* This naming needs more bits for the names,
 * but is compatible to the DC3 algorithm with
 * discarding and also with 3-discarding */

template<class Input>
class discard_naming{
public:
  typedef skew_tuple_type value_type;

private:
  Input& A;
  unsigned int counter;
  unsigned int store_counter;
  bool& finished;
  skew_quad_type tmp,tmp2;
  skew_tuple_type result;

public:
```

```
  discard_naming(Input& A_, bool& finished_):
  A(A_), counter(0),store_counter(0), finished(finished_){
    assert(!A.empty());
    finished = true;
    tmp=*A;
    result.first=tmp.first;
    result.second=counter;
  }

  const value_type& operator*() const{
    return result;
  }

  discard_naming& operator++(){
    assert(!A.empty());
    ++A;
    if(!A.empty()) tmp2=*A;
    ++counter;
    if(!quad_eq(tmp,tmp2)){
      store_counter=counter;
    }
    else{
      if(!A.empty()&(tmp2.second!=0)){
          finished=false;
      }
    }
    result.first=tmp2.first;
    result.second=store_counter;
    tmp=tmp2;
    return *this;
  }

  bool empty() const{
    return A.empty();
  }
};
```

B.3. Comparison Functions

A typical comparison function in our code looks like the binary function less_tuple_1st, which arranges the elements in the ascending order. The purpose of this function is to compare tuples with respect to the first component. The STXXL sorter needs the minimum and the maximum input values to be defined. The input values must be strictly greater than min_value() and strictly less than max_value().

```
// skew_tuple_type is a stl pair
```

```
struct less_tuple_1st{
  typedef skew_tuple_type value_type;

  bool operator()(const value_type& a, const value_type& b)const{
    return a.first<b.first;
  }

  value_type min_value() const  {
    return value_type(MIN_ELEM,MIN_ELEM);
  }

  value_type max_value() const  {
    return value_type(MAX_ELEM,MAX_ELEM);
  }
};
```

We use the following comparison function to arrange tuples for the update step of *X-Tupling* and *X-Discarding*. For X-Tupling and 2-Discarding the magic_cmp function, which is an implementation of the in *Section* 7.2 described comparism function, simplifies the update step. As we also have shown in *Section* 7.2, we can replace the function by two bit rotations for each index of a tuple before and after a sorter.

```
// Comparing with respect to ⟨i mod 2^X, ⌊i/2^X⌋⟩
// with X = shift_h
struct magic_cmp{
  const unsigned mask,mask1;
  typedef tuple_type value_type;

  magic_cmp(unsigned int shift_h):mask((1<<shift_h)-1),mask1(~mask){}

  bool operator()(const value_type& a, const value_type& b)const{
    unsigned int a_mod = (a.first & mask);
    unsigned int b_mod = (b.first & mask);

    if(a_mod==b_mod){
      return (a.first & mask1)<(b.first & mask1);
    }
    else{
      return a_mod<b_mod;
    }
  }

  value_type min_value() const{
    return value_type(MIN_ELEM,MIN_ELEM);
  }

  value_type max_value() const{
```

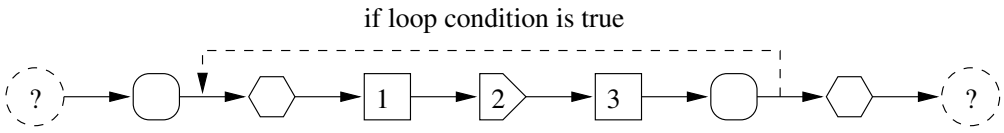

FIGURE B.4.1. Flow chart of the pipelined loop example. The numbers refer to pipeline nodes in the C++ example code.

```
    return value_type(MAX_ELEM,MAX_ELEM);
  }
};
```

B.4. Pipelined Loop Example

The following example code shows how to implement a loop efficiently with the pipelining interface and the STXXL **[9, 10]** external merge sort. We hide the type definitions which are not necessary to understand the loop structure. The loop consists of two scan pipes and two sorters, as we find loops in most of our suffix array construction algorithm implementations. One of the sorters is split into two phases. Sorter run formation (runs_creator(_init)_type) and sorter merge node (runs_merger_type). We use it to split the loops in several pipes. Each pipelined sorter needs a comparison function (sort_function), an input pipe and a number of bytes of main memory (ram_usage) for internal sorting. The block size is defined in the hidden type definitions . *Figure* B.4.1 shows the flow chart of the example.

```
...

// Pointer to sorter run formation nodes
runs_creator_init_type* runs_creator0=0;
runs_creator_type* runs_creator1=0;

// pointer to sorter merge node
runs_merger_type* merge_runs;

// Let input_pipe_instance be the last pipe
// outside the loop
// Connect input_pipe_instance with sorter run formation node
runs_creator0 = new runs_creator_init_type(
  input_pipe_instance,cmp_function,ram_usage);

// Connect sorter run formation with sorter merge node
merge_runs=new runs_merger_type(
  (*runs_creator0).result(),cmp_function,ram_usage);

// Condition for breaking the loop
bool ready=false;
```

```
while(!ready){
   // An arbitrary scan pipe which could be for example
   //  a naming pipe
   // Scan node 1 in Figure B.4.1
  scan1_type scan1(*merge_runs);

  // Sorter node
  // Sorter node 2 in Figure B.4.1
  sorted_stream_type sort_scan1_tuples(
    scan1,cmp_function2(),ram_usage);

  // Clean not used structures
  if(runs_creator1==0){
    delete runs_creator0;
    runs_creator0=0;
  }
  else{
    delete runs_creator1;
    runs_creator1=0;
  }
  delete merge_runs;
  merge_runs=0;

  // A scan pipe which also sets the ready flag
  // Scan node 3 in Figure B.4.1
  scan2_type scan2(sort_scan1_tuples,&ready);

  // connect scan2 with sorter run formation (pipe end)
  runs_creator1 = new runs_creator_type(
    scan2,cmp_function,ram_usage);
  // connect runs creator with runs merger node (pipe begin)
  merge_runs= new runs_merger_type(
    (*runs_creator1).result(),cmp_function,ram_usage);
}

// Load the first loop result in the variable tmp
runs_merger_type::value_type tmp=*(*merge_runs);
...
```

APPENDIX C

More About DCX

C.1. The DCX I/O Volume

Let δ_X be the average merge tuple size (defined in *Section* 9.5), $R = |S|$, N the input string size and m the number of DCX recursions. If we count the *I/O volume* of each stage of the obvious generalisation DCX of *DC3* with $X = S^2 - S + 1$, we get:

(1) Sort naming tuples:
$\sum_{0 \le i < m} \text{sort}\left(\left(R^2 - R + 2\right) N \left(\frac{R}{R^2-R+1}\right)^i\right) <$
$\text{sort}\left(\frac{\left(R^2-R+2\right)\left(R^2-R+1\right)}{R^2-2R+1} N\right)$

(2) Write names:
$\sum_{0 \le i < m} \text{scan}\left(N \left(\frac{R}{R^2-R+1}\right)^i\right) < \text{scan}\left(\frac{R^2-R+1}{R^2-2R+1} N\right)$

(3) Read names:
$\sum_{0 \le i < m-1} \text{scan}\left(N \left(\frac{R}{R^2-R+1}\right)^i\right) < \text{scan}\left(\frac{R^2-R+1}{R^2-2R+1} N\right)$

(4) Sort merge tuples:
$\sum_{0 \le i < m-1} \text{sort}\left(\delta_X \cdot N \cdot \left(\frac{R}{R^2-R+1}\right)^i\right) < \text{sort}\left(\frac{\delta_X\left(R^2-R+1\right)}{R^2-2R+1} N\right)$

(5) Build inverse suffix array:
$\sum_{1 \le i < m-1} \text{sort}\left(2N \left(\frac{R}{R^2-R+1}\right)^i\right) < \text{sort}\left(\frac{R^2-R+1}{R^2-2R+1} 2N - 2N\right).$

The overall *I/O volume* for our simple model of DCX is:

$$\text{sort}\left(\left(\left(\left(R^2 - R + 2\right) + \delta_X + 2\right) \cdot \frac{R^2 - R + 1}{R^2 - 2R + 1} - 2\right) \cdot N\right) + \text{scan}\left(\left(2 \cdot \frac{R^2 - R + 1}{R^2 - 2R + 1}\right) \cdot N\right)$$

C.2. Further Thoughts About Reducing IO Volume

To encode N different equal distributed characters, we we need $\approx \log_2 N$ bits per character. Since the sub problem string size of DCX decreases by factor $\frac{R}{X}$, we can try to save more space storing only the required number of bits per character in each sub problem, what is $\approx \log_2 N \cdot (\frac{R}{X})^i$ bits in recursion depth i. In the following calculations is $x \in [0, 1)$.

(C.2.1)
$$\begin{aligned}
& \sum_{i\in\mathbb{N}_0} i\cdot x^i \\
= \;& x\cdot\sum_{i\in\mathbb{N}_0} i\cdot x^{i-1} \\
= \;& x\cdot\sum_{i\in\mathbb{N}_0} \frac{d}{dx}\cdot x^i \\
= \;& x\cdot\frac{d}{dx}\cdot\sum_{i\in\mathbb{N}_0} x^i \\
= \;& x\cdot\frac{d}{dx}\frac{1}{1-x} \\
= \;& \frac{x}{(1-x)^2}
\end{aligned}$$

The next calculation computes the number of bits one input string word causes over all iterations, if the cost of this word is $\log_2((\frac{R}{X})^i N)$ in recursion depth i.

$$\begin{aligned}
& \sum_{i\in\mathbb{N}_0} \log_2\left(N\cdot\left(\frac{R}{R^2-R+1}\right)^i\right)\cdot\left(\frac{R}{R^2-R+1}\right)^i \\
= \;& \log_2 N\cdot\sum_{i\in\mathbb{N}_0}\left(\frac{R}{R^2-R+1}\right)^i + \sum_{i\in\mathbb{N}_0}\log_2\left(\left(\frac{R}{R^2-R+1}\right)^i\right)\cdot\left(\frac{R}{R^2-R+1}\right)^i \\
= \;& \log_2 N\cdot\sum_{i\in\mathbb{N}_0}\left(\frac{R}{R^2-R+1}\right)^i + \log_2\frac{R}{R^2-R+1}\underbrace{\sum_{i\in\mathbb{N}_0} i\left(\frac{R}{R^2-R+1}\right)^i}_{(C.2.1)} \\
= \;& \log_2 N\cdot\frac{R^2-R+1}{R^2-2R+1} + \log_2\frac{R}{R^2-R+1}\cdot\frac{R^3-R^2+R}{R^2-2R+1}
\end{aligned}$$

The resulting formula is separated in two addends of which only the first one depends on the word cost in recursion depth zero. The first addend encodes the optimal machine word size, if we do not consider a possible cheaper word encoding for the sub problems. The second addend encodes the the sum of the saved bits over all iterations. For *DC3* and *DC7*, we compute $3\cdot\log_2 N$-3.51 and $\frac{7}{4}\cdot\log_2 N - 1.6$. For $N = 2^{32}$, we can save for *DC3* about 3.7% of the overall *I/O volume* and for *DC7* about 2.9%. Since bit alignment and bit encoding is expensive, we do not use this improvement in our code.

Another possible improvement for large difference covers is to use *X-Tupling* steps for the naming and/or the merge step. For example the merge tuples need in *DC57* from 36 up to 56 characters of the input string. Since this improvement depends on the *difference cover*, we cannot give an *I/O optimal* naming procedure and stop at this point.

TABLE 1. Difference Covers

R	Perfect Difference Covers of $\mathbb{Z}_{R^2-R+1}$
2	$\{0, 1\}$
3	$\{0, 1, 3\}$
4	$\{0, 1, 3, 9\}$
5	$\{0, 1, 6, 8, 18\}$
6	$\{0, 1, 3, 8, 12, 18\}$
7	$\{0, 1, 16, 20, 22, 27, 30\}$
8	$\{0, 1, 9, 11, 14, 35, 39, 51\}$
9	$\{0, 1, 3, 7, 15, 31, 36, 54, 63\}$
10	$\{0, 1, 7, 16, 27, 56, 60, 68, 70, 73\}$
11	$\{0, 1, 5, 8, 18, 20, 29, 31, 45, 61, 67\}$
12	$\{0, 1, 32, 42, 44, 48, 51, 59, 72, 77, 97, 111\}$

C.3. Perfect Difference Covers

Table 1 shows perfect difference covers from [**15**]. Note, that we can get more difference covers from a given difference cover S by just adding another residue class to all entries. Since these difference covers are isomorph, we will get the same merge tuple sizes.

Bibliography

[1] *Full-Text Indexes in External Memory*, volume 2625 of *Lecture Notes in Computer Science*, chapter 7, pages 149–170. Springer-Verlag Berlin Heidelberg, 2003.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, September 1988.

[3] L. Arge, P. Ferragina, R. Grossi, and J. S. Vitter. On sorting strings in external memory. In *29th ACM Symposium on Theory of Computing*, pages 540–548, El Paso, May 1997. ACM Press.

[4] L. Arge, O. Procopiuc, and J. S. Vitter. Implementing i/o-efficient data structures using TPIE. In *10th European Symposium on Algorithms (ESA)*, volume 2461 of *LNCS*, pages 88–100. Springer, 2002.

[5] S. Burkhardt and J. Kärkkäinen. Fast lightweight suffix array construction and checking. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*, volume 2676 of *Lecture Notes in Computer Science*, pages 55–69. Springer, 2003.

[6] C. J. Colbourn and A. C. H. Ling. Quorums from difference covers. *Information Processing Letters*, 75(1–2):9–12, July 2000.

[7] A. Crauser and P. Ferragina. Theoretical and experimental study on the construction of suffix arrays in external memory. *Algorithmica*, 32(1):1–35, 2002.

[8] A. Crauser and K. Mehlhorn. LEDA-SM a platform for secondary memory computations. Technical report, MPII, 1998. draft.

[9] R. Dementiev. Stxxl library. documentation and download at `http://www.mpi-sb.mpg.de/~rdementi/stxxl.html`.

[10] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*. ACM, 2003.

[11] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. 38th Annual Symposium on Foundations of Computer Science*, pages 137–143. IEEE, 1997.

[12] M. Farach, P. Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *Proc. 39th Annual Symposium on Foundations of Computer Science*, pages 174–183. IEEE, 1998.

[13] M. Farach-Colton, P. Ferragina, and S. Muthukrishnan. On the sorting-complexity of suffix tree construction. *Journal of the ACM*, 47(6):987–1011, 2000.

[14] G. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: PAT trees and PAT arrays. In W. B. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Data Structures & Algorithms*. Prentice-Hall, 1992.

[15] H. Haanpää. Minimum sum and difference covers of Abelian groups. Research Report A88, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, February 2004.

[16] W.-K. Hon, T.-W. Lam, K. Sadakane, and W.-K. Sung. Constructing compressed suffix arrays with large alphabets. In *Proc. 14th International Symposium on Algorithms and Computation*, volume 2906 of *Lecture Notes in Computer Science*, pages 240–249. Springer, 2003.

[17] W.-K. Hon, K. Sadakane, and W.-K. Sung. Breaking a time-and-space barrier in constructing full-text indices. In *Proc. 44th Annual Symposium on Foundations of Computer Science*, pages 251–260. IEEE, 2003.

[18] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[19] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *Proc. 4th Annual Symposium on Theory of Computing*, pages 125–136. ACM, 1972.

[20] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proc. 12th Annual Symposium on Combinatorial Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[21] D. K. Kim, J. S. Sim, H. Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003.

[22] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. 14th Annual Symposium on Combinatorial Pattern Matching*. Springer, June 2003.

[23] T.-W. Lam, K. Sadakane, W.-K. Sung, and Siu-Ming Yiu. A space and time efficient algorithm for constructing compressed suffix arrays. In *Proc. 8th Annual International Conference on Computing and Combinatorics*, volume 2387 of *Lecture Notes in Computer Science*, pages 401–410. Springer, 2002.

[24] U. Manber and G. Myers. Suffix arrays: a new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, October 1993.

[25] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACMTMCS: ACM Transactions on Modeling and Computer Simulation*, 8:3–30, 1998. `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[26] E. M. McCreight. A space-economic suffix tree construction algorithm. *Journal of the ACM*, 23(2):262–272, 1976.

[27] S. S. Rao. Time-space trade-offs for compressed suffix arrays. *Information Processing Letters*, 82(6):307–311, June 2002.

[28] J. M. Patel S. Tata, R. A. Hankins. Practical suffix tree construction. In *Proc. 30th Very Large Data Bases*. Very Large Data Bases, 2004.

[29] K. Sadakane and T.Shibuya. Indexing huge genome sequences for solving various problems. *Genome Informatics*, 12:175–183, 2001.

[30] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, I: Two level memories. *Algorithmica*, 12(2/3):110–147, 1994.

[31] P. Weiner. Linear pattern matching algorithm. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.