

UNIVERSITY OF TARTU  
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE  
INSTITUTE OF COMPUTER SCIENCE

Egon Elbre  
Parallel Pattern Discovery  
Master's Thesis

Supervisor: Prof. J. Vilo

Tartu May 5, 2013

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Motivation and background . . . . .	4
1.2	Structure of the thesis . . . . .	4
<b>2</b>	<b>Definitions</b>	<b>6</b>
2.1	Sequence and Dataset . . . . .	6
2.2	Pattern . . . . .	7
2.3	Query . . . . .	7
2.3.1	Query features . . . . .	8
2.4	Pool . . . . .	8
2.5	Pattern Discovery . . . . .	9
<b>3</b>	<b>Algorithms</b>	<b>10</b>
3.1	Algorithms . . . . .	10
3.1.1	SPEXS . . . . .	10
3.1.2	TEIRESIAS . . . . .	11
3.1.3	Verbumculus . . . . .	12
3.1.4	MobyDick . . . . .	12
3.1.5	RSAT . . . . .	12
3.2	Reviews . . . . .	12
<b>4</b>	<b>SPEXS Generalization</b>	<b>13</b>
4.1	Algorithm . . . . .	13

better  
title  
needed

4.2	Pools . . . . .	15
4.3	Filtering . . . . .	15
4.4	Extending . . . . .	15
4.4.1	Sequences . . . . .	16
4.4.2	Groups . . . . .	17
4.4.3	Star . . . . .	18
4.4.4	Optimizations . . . . .	19
4.5	Combining . . . . .	19
<b>5</b>	<b>Parallelization</b>	<b>20</b>
5.1	Process . . . . .	20
5.2	Extending . . . . .	21
5.3	Distributed process . . . . .	22
<b>6</b>	<b>Implementation</b>	<b>24</b>
6.1	Architecture . . . . .	24
6.2	Configuration . . . . .	25
6.3	Input and Output . . . . .	26
6.4	Alphabet and Database . . . . .	26
6.5	Pools . . . . .	26
6.6	Features and Filters . . . . .	27
6.7	Synchronized Tree Traversal . . . . .	28
6.8	Debugging . . . . .	31
<b>7</b>	<b>Applications and experimental results</b>	<b>33</b>
7.1	Examples . . . . .	33
7.1.1	DNA sequences . . . . .	33
7.1.2	Protein sequences . . . . .	33
7.1.3	Text mining . . . . .	33
7.1.4	Code mining . . . . .	34
7.2	Performance . . . . .	34

<b>8</b>	<b>Conclusions</b>	<b>35</b>
	<b>Bibliography</b>	<b>36</b>
<b>A</b>	<b>spexs2</b>	<b>39</b>
	A.1 source . . . . .	39
<b>B</b>	<b>Concise Implementation</b>	<b>41</b>

# Chapter 1

## Introduction

write at least 3-4 pages

### 1.1 Motivation and background

Collecting new data has been increasing more rapidly than algorithms and computer processing power. The average size of each dataset has also been increasing. This suggests that the only way to keep up with analysis is to parallelize algorithms.



cite

One of main drivers of such large datasets is analysis of genomic and proteomic sequences.

write todo

write a priori vs some prior knowledge

write broad examples from nat processing, code

write usual limitations, alphabet, pattern language...

### 1.2 Structure of the thesis

In this thesis we explore an algorithm for finding patterns and show how generalizations can make it scalable and flexible, and simpler both in theory

and implementation compared with non-abstract version.

write the basic premise of parallelization

In the the Chapter 2 we give definitions for terminology that is used to describe the alogrithm. In the Chapter 3 we investigate the original SPEXS algorithm [1] and other algorithms . We generalize the SPEXS algorithm to get a flexible algorithm in Chapter 4 and we partition the algorithm to make it suitable for parallelization in Chapter 5. In Chapter 6 we discuss the implementation of SPEXS. In Chapter 7 we show possible applications for the algorithm.

cite  
other  
algo-  
rithms

# Chapter 2

## Definitions

Pattern discovery is a research area aiming to discover unknown patterns in a given set of data structures that are frequent and interesting according to some measure. In this chapter we formally define necessary terms used in this thesis.

### 2.1 Sequence and Dataset

We use  $\Sigma$  to denote the set of tokens in the dataset, an *alphabet*. The *size* of the alphabet is  $|\Sigma|$ . *Tokens* can be numbers, letters, words or sentences - any symbol.

Any sequence  $S = a_1a_2...a_n, \forall a_i \in \Sigma$  is called a *sequence* over the token set  $\Sigma$ . If the length of the string is 0, it is called an empty sequence or  $\epsilon$ .

**Example 2.1.1.** `ACGTGCCATC` is a sequence where  $\Sigma = \{\text{A}, \text{C}, \text{G}, \text{T}\}$ .

A *dataset* is a collection of sequences.

**Example 2.1.2.** In a document sentences can be considered as a *dataset*, where a single sentence is a *sequences* and each word is a *token* in the alphabet. Text `This is some example. This is an other example.` has sequences  $\{ [\text{This is an example}], [\text{This is an other example}] \}$  and the alphabet is  $\Sigma = \{\text{this}, \text{is}, \text{an}, \text{example}, \text{other}\}$ .

## 2.2 Pattern

Our aim is to discover repetitive and common structures in data. We call such structures *patterns*. A generic way to define a *pattern* is as a set of all the sub-structures it represents. This means we can say whether some data sub-structure is represented by a pattern.

The *pattern structure* is usually dependent on the data-structures which it represents. For example sequence patterns are usually represented sequences, graph patterns are represented as graphs; but sequence patterns could also be represented as a graph.

We denote the set of structures that a pattern structure  $p$  defines as  $all(p)$ . If  $\alpha \in all(p)$ , where  $\alpha$  is a structure then we say that structure  $\alpha$  *matches exactly* pattern  $p$ . We say that  $\alpha$  *matches*  $p$  if any of structure  $all(p)$  is a sub-structure of  $\alpha$ .

In this thesis we only consider sequential pattern structures and use *pattern* to mean *sequential pattern structure*. We represent such patterns with regular expressions.

write about regexps

cite  
regex

*Pattern size* is the length of the pattern sequence.

**Example 2.2.1.** `. [AT]` is a pattern of size 2 and denotes a set  $\{ \text{AA}, \text{AT}, \text{CA}, \text{CT}, \text{GA}, \text{GT}, \text{TA}, \text{TT} \}$ ; it matches `CCTC` and exactly matches `AT`.

We denote the set of all pattern  $p$  *matches* in a dataset  $D$  as  $matches(p, D) = \{match(p, \alpha) | \alpha \in D, matches(p, \alpha)\}$ .

## 2.3 Query

We need to somehow understand where given pattern  $p$  is located in a dataset  $D$ . This compound structure  $q = \langle D, p, matches(p, D) \rangle$  is called a *query*.

**Example 2.3.1.** Let our dataset be  $D = [\text{ACGT}, \text{TXCGA}]$  and our pattern be  $p = \text{C.}$ . The corresponding query is  $\langle D, p, \{[1, 3], [2, 4]\} \rangle$ , which means



that the pattern  $p$  ends in sequence 1 at position 3 and in sequence 2 at position 4.

### 2.3.1 Query features

When we talk about how "interesting" a pattern is, we are actually evaluating the query, since the pattern requires a context where it can be "interesting".

Queries can have different properties: length, number of matches in the dataset, pattern textual representation etc. Such properties can be represented by a function that take a query as an input and return the property. Formally a *query feature* is a function  $f : \text{Query} \mapsto \text{Any}$ .

We also need to see how "interesting" one query is compared to the others. *Query interestingness* is a function  $f : \text{Query} \mapsto \text{Value}$  where the *Value*s are well-ordered. This gives a measure to compare two different queries. Often we can represent such interestingness measures as a real number.

We should also be able to somehow specify criterias for query. *Query filter* is a function  $f : \text{Query} \mapsto \text{Boolean}$  and shows whether the query matches the criteria.

**Example 2.3.2.** Pattern occurrences in a document is a interestingness measure. Whether query pattern is at least 3 tokens is a query filter.

## 2.4 Pool

*Pool* is an abstract datatype for a collection of queries. The pool allows queries to be stored. The only operations that pool must provide is "push", for adding a query, and "pop", for getting a query.

**Example 2.4.1.** Stacks and queues both satisfy the pool requirement. We could also define a pool that stores the queries on the disk; also it could pack or reorder the queries for performance reasons.

## 2.5 Pattern Discovery

In this thesis *pattern discovery* is a process of finding the most interesting subset, according to a query interestingness, of sequential patterns, that conform to some criteria, in a sequence dataset.

**Example 2.5.1.** Let our search problem be "Finding most common nucleotide patterns that are at least 3 nucleotides long from a shotgun sequencing output.", then *most common* defines our interestingness measure. *At least 3 nucleotides* is the pattern subset criteria. *Sequencing output* is our dataset and *nucleotides* define the token alphabet.

# Chapter 3

## Algorithms

better  
title  
needed

There are many itemset discovery algorithms, but only few are general and can discover more complex patterns.

cite  
???

### 3.1 Algorithms

#### 3.1.1 SPEXS

SPEXS is an pattern discovery algorithm described by Vilo et al. [1] This algorithm finds patterns from a sequence. We take this as our basis for developing a new parallel algorithm. In this chapter we describe original algorithm so that we can later show the changes made to this algorithm.

We describe the general representation of the SPEXS algorithm. The original algorithm was as follows:

move algorithm to generalization

---

**Algorithm 1** The SPEXS algorithm

---

**Input:** String  $S$ , pattern class  $\mathcal{P}$ , output criteria, search order, and fitness measure  $\mathcal{F}$

**Output:** Patterns  $\pi \in \mathcal{P}$  fulfilling all criteria, and output in the order of fitness  $\mathcal{F}$

```
1: Convert input sequences into a single sequence
2: Initiate data structures
3: Root  $\leftarrow$  new node
4: Root.label  $\leftarrow \epsilon$ 
5: Root.pos  $\leftarrow (1, 2, \dots, n)$ 
6: enqueue( $\mathcal{Q}$ , Root, order)
7: while  $N \leftarrow$  dequeue( $\mathcal{Q}$ ) do
8:   Create all possible extensions  $p \in \mathcal{P}$  of  $N$  using  $N$ .pos and  $S$ 
9:   for extension  $p$  of  $N$  do
10:    if pattern  $p$  and position list  $p$ .pos fulfill the criteria then
11:       $N$ .child  $\leftarrow p$ 
12:      calculate  $\mathcal{F}(p, S)$ 
13:      enqueue( $\mathcal{Q}$ ,  $p$ , order)
14:    if  $p$  fulfills the output criteria then
15:      store  $p$  into output queue  $\mathcal{O}$ 
16: Report the list of top-ranking patterns from output queue  $\mathcal{O}$ 
```

---

The main idea of the algorithm is that first we generate a pattern and a query that matches all possible positions in the sequence. We then put this query into a queue for extending. Extending a query means finding all queries whose patterns length is longer by 1. If any of the queries is fit, by some criteria, it will be put into the main queue, for further extension, and output queue for possible output.

### 3.1.2 TEIRESIAS

write overview

[2]

### 3.1.3 Verbumculus

write overview

[3]

### 3.1.4 MobyDick

write overview

[4]

### 3.1.5 RSAT

write overview

[5]

## 3.2 Reviews

write about some reviews

# Chapter 4

## SPEXS Generalization

Work in progress

In this chapter we show how to make SPEXS algorithm more abstract by allowing flexibility through function composition and finding minimal requirements for the data-structures.

### 4.1 Algorithm

The algorithm in a more conventional view is:

---

**Algorithm 2** The spexs2 algorithm

---

**Input:** *dataset*, *in* and *out* are pools, *extend* is an extender function, *extend?*, *output?* are filters

**Output:** Patterns satisfying filters and *extender* are in *out* pool

```
1: function SPEXS2(dataset, in, out, extend, extend?, output?)
2:    $\epsilon \leftarrow \text{NewEmptyQuery}(\text{dataset})$ 
3:   in.put( $\epsilon$ )
4:   while  $q \leftarrow \text{in.pop}()$  do
5:     extended  $\leftarrow \text{extend}(q, \text{dataset})$ 
6:     for  $qx \in \text{extended}$  do
7:       if  $\text{extend?}(qx)$  then
8:         in.push( $qx$ )
9:       if  $\text{output?}(qx)$  then
10:        out.push( $qx$ )
11:    postprocess( $q$ )
```

---

When the algorithm starts we create an empty pattern query  $\epsilon$  and put into the *in* pool. The *in* pool contains queries whose patterns should be further examined.

We pick a query from the *in* pool for extending. The extending means generating all queries whose pattern size is larger by one. There can be several such queries.

If any of the queries should be further examined as defined by the *extendable* query filter, it will be put into the *in* pool.

If the query is suitable for output as defined by the *outputtable* filter, it will be put into the *out* pool.

If we extend each pattern at each step by one we guarantee that we examine all the patterns that conform to our criteria as defined by *extendable* filter.

## 4.2 Pools

It has no guarantees on how the queries are stored internally and in which order they are taken out. This gives an option to store on disk the queries if needed.

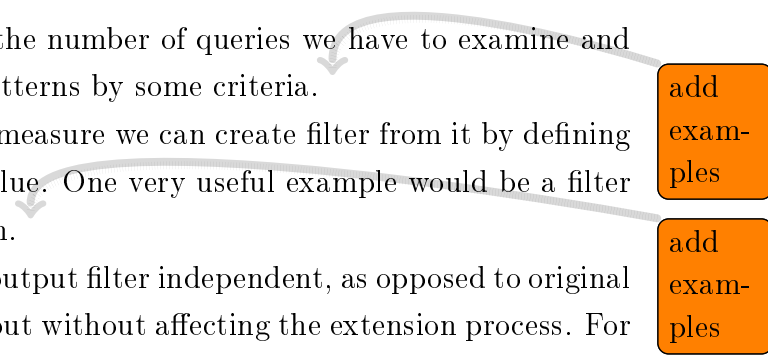
In practice this means we can use any collection such as list, set, queue as a pool. This gives us different performance and memory characteristics.

## 4.3 Filtering

Filtering allows us to reduce the number of queries we have to examine and allows to select a subset of patterns by some criteria.

If we have interestingness measure we can create filter from it by defining it's minimum or maximum value. One very useful example would be a filter for limiting the pattern length.

By making extension and output filter independent, as opposed to original SPEXS, we can still limit output without affecting the extension process. For example if we wish to see only patterns of length 3 we cannot do it with one filter, since we still would need to extend patterns of length 0, 1 and 2.



add  
exam-  
ples

add  
exam-  
ples

## 4.4 Extending

The extending process is at the core of the algorithm and there are several ways of doing it.

write motivation

write comments

The idea of extending method is:



---

**Algorithm 3** SPEXS2 extender

---

**Input:**  $q$  query,  $walk$  function

**Output:** result contains queries that have been extended by one

```
1: steps  $\leftarrow$  new collection([Token, Pos])
2: for pos  $\in$  positions( $q.matches$ ) do
3:   [token, loc]  $\leftarrow$  walk( $q.document$ , pos)
4:   steps.add([token, pos])
5: matches  $\leftarrow$  new map(Token  $\mapsto$  [Pos])
6: for [token, loc]  $\in$  steps do
7:   if matches[token] doesn't exist then
8:     matches[token]  $\leftarrow$  empty set
9:   matches[token]  $\leftarrow$  matches[token] + loc
10: queries  $\leftarrow$  new collection([Query])
11: for (token, positions)  $\in$  matches do
12:    $qx \leftarrow$  new query
13:    $qx.document \leftarrow q.document$ 
14:    $qx.pattern \leftarrow q.pattern + token$ 
15:    $qx.positions \leftarrow positions$ 
16:   queries.add( $qx$ )
```

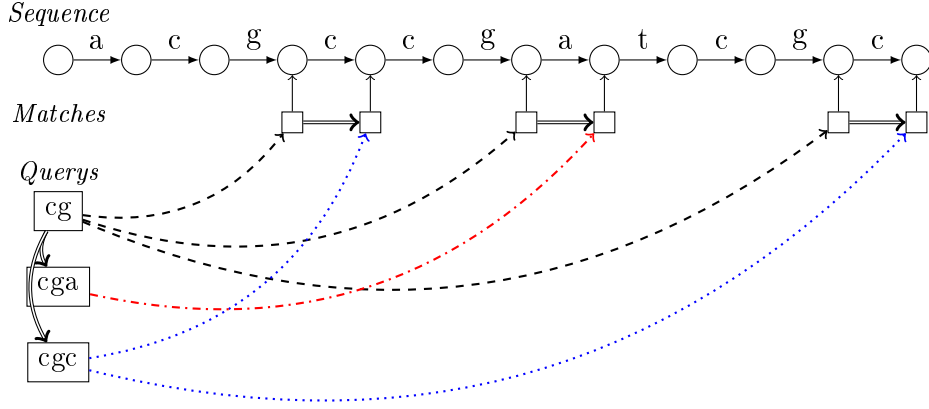
---

The behavior of *extender* depends on how *next* is implemented, we shall look at different ways how to implement it. The different implementations can capture more complex patterns, but often at the cost of performance.

#### 4.4.1 Sequences

The simplest case how the next function behaves is when we are only looking for simple sequences – the alphabet for patterns and sequences is the same.

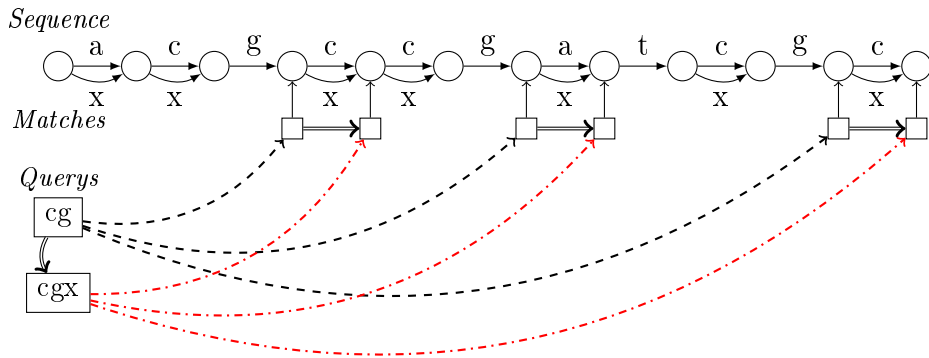
Let's consider a sequence *ACGCCGATCGC* and a pattern *CG*.



Initially we have matches only for query  $CG$ . Then by taking the *next* token from the sequence we can build up querys  $CGA$  and  $CGC$ .

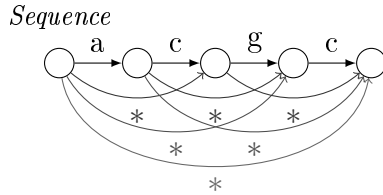
#### 4.4.2 Groups

One common addition in a pattern language is capturing a group of tokens. For example we can use  $X = [AC]$  to denote both tokens  $A, C$ . By adding where either one transitions we can capture such groups in the extension process.



### 4.4.3 Star

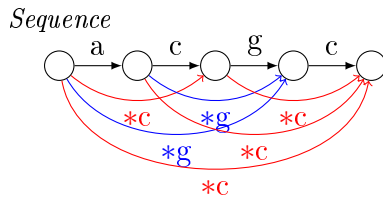
Another possible extension is the *dot-star* or more simply capturing a run of elements. Here we just show the expanded sequence with \* symbol.



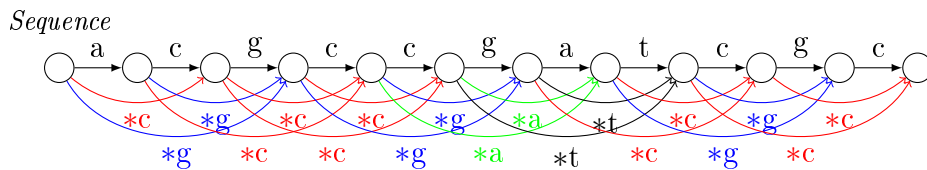
Here we immediately notice how the complexity increases by introducing pattern token.

write properly

We can skip this intermediary step isn't necessary we can instead extend with  $*Y$ , where  $Y$  is some other token. This means we avoid this single large query and have multiple smaller queries.



We can also limit the length of the run.



Here we have limited the run length to be either 2 or 3.

#### 4.4.4 Optimizations

We can further optimize *group next* by inferring the positions from simple *sequence next*. We can see that the positions of a group extension is the same as the union of positions by the group tokens.

If we have a group token  $\gamma$  that contains  $tokens(\gamma)$  then the *matches* for such group is

$$matches(p\gamma, D) = \bigcup_{t \in tokens(\gamma)} matches(pt, D)$$

### 4.5 Combining

write what follows, how used

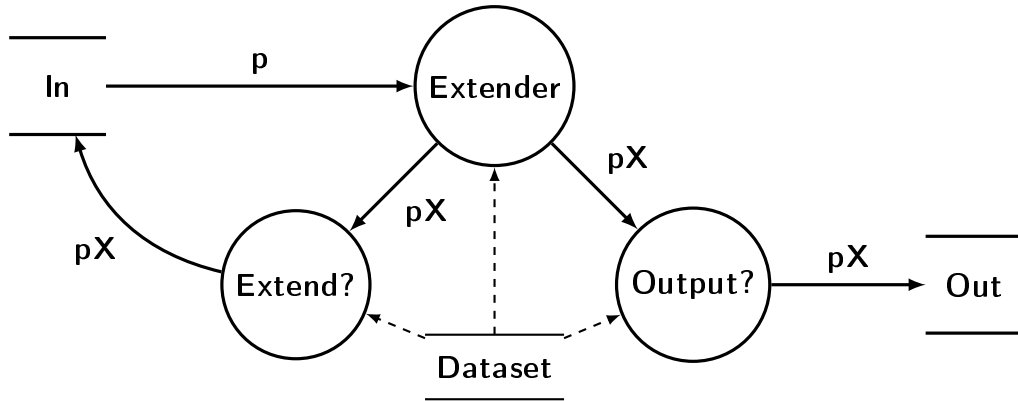
# Chapter 5

## Parallelization

Here we describe how we can partition the algorithm to support parallelism. A proof of concept for fully parallelized code can be seen in appendix B.

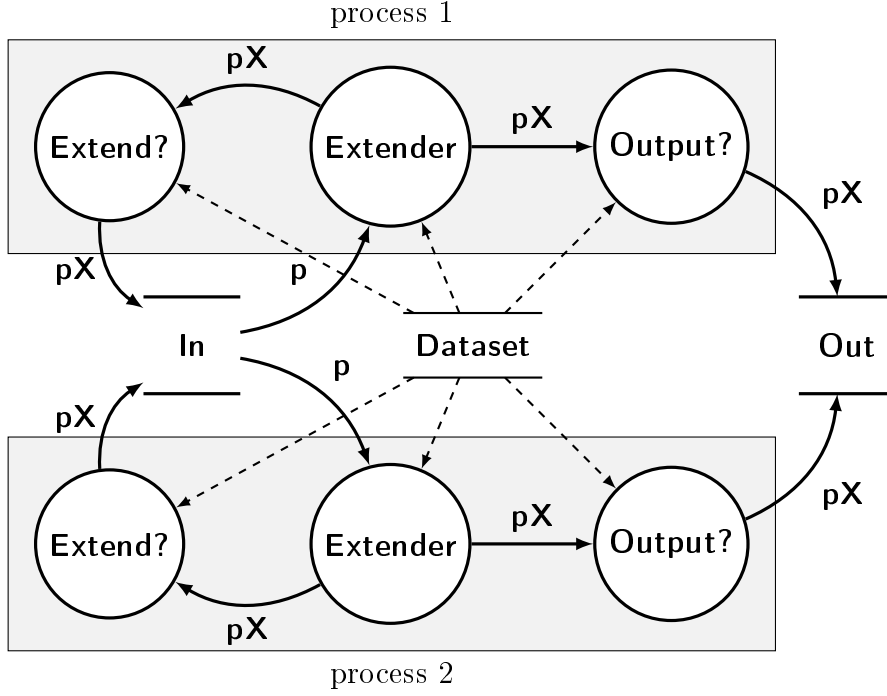
### 5.1 Process

The main process of the algorithm as described by dataflow diagram.[6, 7]  
Circles denote processes and unfinished rectangles denote data stores.



We can see that the different query extensions do not share a dependency, except the dataset. Since dataset itself is read-only for a given process, it

means we can use multiple extender processes. The same applies for extendability and output filter.



We can add more processes in a similar fashion without affecting the end result. Although this will introduce a source of indeterminism.

## 5.2 Extending

*Extender* can be parallelized by using MapReduce[8] concepts. We use here Clojure[9] reducers library to show how this can be implemented.

---

**Algorithm 4** Parallel extender

---

```
1 (require '[clojure.core/reducers :as r])
2
3 ; fold-join based grouping function
4 (defn group-map-by [g f coll]
5   (r/fold
6     (r/monoid (partial merge-with into) (constantly {}))
7     (fn [ret x]
8       (let [k (g x)]
9         (assoc ret k (conj (get ret k []) (f x))))))
10    coll))
11
12 (defn extend [dataset query]
13   (let [steps (r/mapcat #(walk dataset %) (:positions query))
14         grouped (group-map-by :token :position steps)]
15     (r/map #(child-query q %) grouped))))
```

---

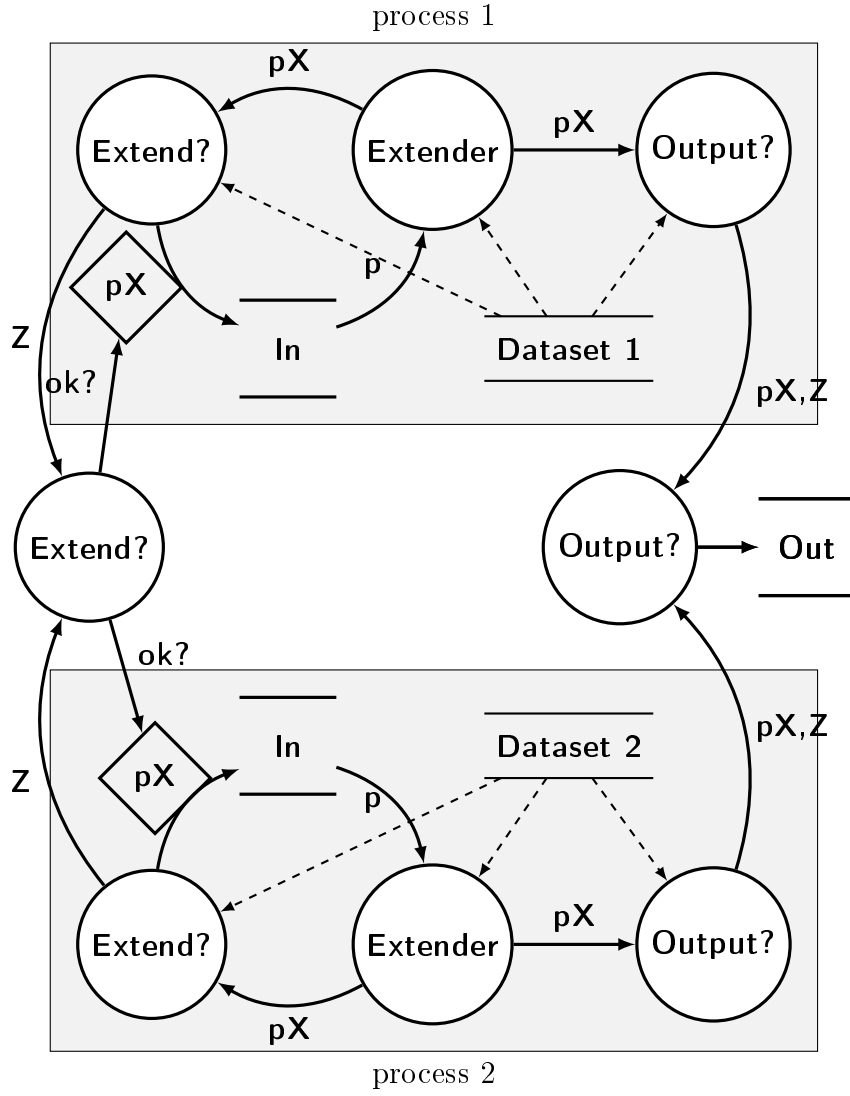
On CPU-s this may not give much improvement, but this parallelization could be beneficial for highly parallel processors such as GPGPU-s or FPGA-s.

### 5.3 Distributed process

Since the dataset and the process memory consumption can get quite it would also be beneficial to be able to partition the dataset between multiple machines.

The extension results for a given query stays inside the sequence which means we can partition by dividing sequences to separate datasets.

The whole dataset is required only for filtering, since even one of the simplest operations ("counting matches in dataset") requires full knowledge of all matches over the dataset. We can use partial results ( $Z$ ) to reduce the communication overhead.



For example to see whether some query is over some count limit we first count matches in the partial datasets, then send the partial results to a process that adds these results together. Depending on that result we can decide whether it needs further processing.



# Chapter 6

## Implementation

Here we discuss a practical implementation, *spexs2*, for pattern discovery in sequences.

In this chapter we will discuss a practical implementation, *spexs2*, for pattern discovery in sequences. We only discuss parts that we consider non-trivial or interesting and could be useful in implementing other algorithms.

Information about the full source code is in the Appendix B and C.

### 6.1 Architecture

The main criteria for designing program have been described in D. Parnas paper "On the Decomposition of Programs"[10]. It suggests decomposing into isolated units and parts that are likely to change together.

We chose the following decomposition for the application:

**Configuration** structure for holding the configuration data

**Setup** based on the configuration initializes data-structures and functions for the algorithm

**Reader** reads in the data from files

**Database** a collection of datasets

**Algorithm** the SPEXS2 algorithm

**Printer** prints the result queries

**Debugging** utilities for the program

The algorithm was decomposed based on the generalization:

**Pattern** represents a pattern

**Query** stores the pattern with database and pattern

**Set** stores matching positions

**Pool** stores queries

**Extender** drives algorithms extending step

**Feature** computes a value from a query

**Filter** filters a query

write why is this important



architecture flow

## 6.2 Configuration

One problem with flexible algorithms is that there are a lot of ways to be run. This often leads to having tens or hundreds of program flags.

To avoid this problem we first decided to use a *json* file for the program configuration. Using *json* requires exact placement of commas and quotes that caused a lot of problems, hence we started using *yaml* that is more suitable for human input.

write Short example of a configuration file.

When running the program from command line it can be uncomfortable to make little changes to the configuration files. In these cases a command line flag would be much easier. To solve this problem we added markup into the json files so they could be replaced with a flag.

For example, inside the configuration file:

```
"Datasets" : {  
  "fore" : { "File" : "$input:main$"  
  ...
```

Using *speers2 -conf conf.json input=other* would replace the "File" value with "other". If there is no flag from command line is given, it will be replaced with "main".

write profiles

## 6.3 Input and Output

write about importance of separating the input reader and output printer from the algorithmic code

## 6.4 Alphabet and Database

write about problems with large alphabets and datasets

## 6.5 Pools

There can be different performance characteristics when using a particular implementation. Most importantly to support parallelism they should be ideally lock-free, but we can use locked version as well.

remove the lock-free stuff

If we use a fifo queue as the in pool the algorithm does a breadth first search of patterns. This can be problematic since we would need a lot of memory to hold all the patterns in memory.

A lifo queue for in pool is a more reasonable choice for memory problems since we need to hold less patterns in memory.

A priority queue suits for the output pool since we can easily then use some feature to sort the queue. A lock-free priority queue would be preferred, but it can be hard to get right.

[link to lock-free queues](#)

Trivial solution would be to use locks for enqueueing and dequeuing, but under high contention it will become a bottleneck.

We know that most of the time we do not need all the results, but only the best. We also know that the result limit is usually several magnitudes smaller than the number of output candidates; this means most of the queries put into the result queue will be immediately discarded.

Uf the query is worse than the worst in the priority queue, it can be discarded immediately without doing push/pop. If we allow for that check to fail once in a while - we can make it mostly lock-free.

## 6.6 Features and Filters

write simpler and examples

One problem is that the amount of possible filters it's useful to construct them from some other features. Or if we wish to use multiple filters we can combine them.

We can use these features to find out something about the query. They each feature is defined as:

```
type Feature func(q *Query) (float64, string)
```

Most of features are in  $\mathfrak{R}$ , but for some there is some extended information that we may wish to know - hence the need for additional string value. One of the simplest is Pattern representation.

Also many of the features are defined in terms of multiple datasets. We can use a closure to easily define a more generic feature.

Example:

```
func Matches(dataset []int) Feature {
    return func(q *Query) (float64, string) {
        matches := countf(q.Pos, dataset)
        return matches, ""
    }
}
```

Here the Matches function creates a feature function defined for dataset.

We can use the name in the configuration file as "Matches(fore)".

If a feature returns a floating point value we can easily turn that into a filter by specifying a minimum or/and maximum value.

For example to give a lower and higher limits to some feature:

```
func featureFilter(feature Feature, min float64, max float64) Filter {
    return func(q *Query) bool {
        v, _ := feature(q)
        return (min <= v) && (v <= max)
    }
}
```

Of course there are some filters that cannot be defined by features hence there is still possibility to make separate filters. Such as disallowing star symbol in the beginning of the pattern.

## 6.7 Synchronized Tree Traversal

*spexs2* can be seen as a pattern tree traversal algorithm with some extra logic. Implementing search over a tree requires synchronization such that there are only a certain number of workers and that they wouldn't die of starvation.

Without synchronization the parallel version looks like:

---

**Algorithm 5** Tree traversal

---

**Output:** All nodes in tree get processed with fn

```
1: function VISIT(tree, start, fn, examine?)
2:   unvisited  $\leftarrow$  { start }
3:   start workers
4:     while unvisited not empty do
5:       node  $\leftarrow$  unvisited.take()
6:       fn(node)
7:       for child  $\in$  children(node) do
8:         if examine?(child) then
9:           unvisited.put(child)
10:
11:   wait for workers
```

---

This would not work correctly with multiple workers since there are race conditions and the workers can die early due to starvation.

The solution is to control worker startup and only terminate workers if all have finished and there are no more items in unvisited set.

---

**Algorithm 6** Synchronized graph traversal

---

**Output:** All nodes in *graph* get processed with *fn*

```
1: function VISIT(graph, start, fn, examine?)
2:   added  $\leftarrow$  new semaphore(0)
3:   terminate  $\leftarrow$  false
4:   mutex  $\leftarrow$  new mutex()
5:   workers  $\leftarrow$  0
6:   unvisited  $\leftarrow$  { start }
7:   added.signal()
8:   start workers
9:     while true do
10:       added.wait()
11:       mutex.lock()
12:       if terminate then
13:         added.signal()
14:         mutex.unlock()
15:         break
16:       node  $\leftarrow$  unvisited.take()
17:       workers  $\leftarrow$  workers + 1
18:       mutex.unlock()

19:       fn(node)

20:       for child  $\in$  children(node) do
21:         mutex.lock()
22:         if examine?(child) then
23:           unvisited.put(child)
24:           added.signal()
25:           mutex.unlock()

26:       mutex.lock()
27:       workers  $\leftarrow$  workers - 1
28:       if workers = 0 and unvisited = {} then
29:         terminate  $\leftarrow$  true
30:         added.signal()
31:       mutex.unlock()
32:
33:   wait for workers
```

---

We use *mutex* to protect variables and data structures. Semaphore *added* tracks how many items are in the *unvisited* set, if the process finally terminates it is turned into a turnstile on line 31 and 13. Variable *workers* tracks how many workers are busy.

## 6.8 Debugging

Seeing how the algorithm works is very useful to get an understanding how the algorithm works. This often can help to either improve the input configuration or debug the program itself. Often this is resolved by adding debug statements.

For example:

```

1 func Spexs(s *Setup) {
2     for q, ok := s.In.Pop(); ok {
3         trace("started extending %v", q)
4         extended := s.Extend(q)
5         trace("extension result %v", extended)
6         for qx := range extended {
7             if s.Extendable(qx) {
8                 trace("> extendable %v" qx)
9                 s.In.Push(qx)
10            }
11            if s.Outputtable(qx) {
12                trace("> outputtable %v" qx)
13                s.Out.Push(qx)
14            }
15        }
16    }
17 }
```

Such statements make it harder to read the actual code, also it's hard to modify the statements for debugging or provide different ways of debugging.

We can use lexical closures to make it simpler:

```

1 type Extender func(q Query) []Query
2
3 func AddDebuggingStatements(s *Setup) {
4     fn := s.Extend
5     s.Extend := func(q Query) []Query {
6         trace("started extending %v", q)
7         extended := fn(q)
8         trace("extension result %v", extended)
9     }
```



```

9
10         for qx := range extended {
11             trace(" > %v", qx)
12             trace(" > extendable %v", s.Extendable(qx))
13             trace(" > outputtable %v", s.Outputtable(qx))
14         }
15         return extended
16     }
17 }
18
19 func Spexs(s *Setup) {
20     for q, ok := s.In.Pop(); ok {
21         extended := s.Extend(q)
22         for qx := range extended {
23             if s.Extendable(qx) {
24                 s.In.Push(qx)
25             }
26             if s.Outputtable(qx) {
27                 s.Out.Push(qx)
28             }
29         }
30     }
31 }
32
33 func run(){
34     S := CreateSpexsSetup()
35     AddDebuggingStatements(S)
36     Spexs(S)
37 }

```

We have removed the debugging statements from the algorithm. We could define other such "debug statement injectors" that provide different levels of details. This method of course has a slight performance impact due to the additional indirection. This can be extended to provide user interaction and other features.

# Chapter 7

## Applications and experimental results

Work in progress

### 7.1 Examples

Here we show examples for the program:

#### 7.1.1 DNA sequences

make an example

#### 7.1.2 Protein sequences

make an example

#### 7.1.3 Text mining

make an example

#### 7.1.4 Code mining

make an example

### 7.2 Performance

make an example

# Chapter 8

## Conclusions

Work in progress

write results 1

write results 2

write context, compare

write strength + limitations

write speed

write so what? why is it important

write what is next?

write strong conclusions

write take home message

In this thesis we showed how by making an algorithm more abstract and general we can also make it parallel. We showed that this algorithm can find patterns from sequences.

Although we showed that we can apply this algorithm on NFAs we did not analyze the performance characteristics. This suggests that this algorithm may be able to work on trees and graph, but would require slight modifications.

We also demonstrated how to make the algorithm more concrete and work

well on some particular datasets. This also took into consideration further development and flexibility of the algorithm.

The SPEXS2 implementation currently is already used in biosequenceing and text mining.

# Bibliography

- [1] J. Vilo, *Pattern Discovery from Biosequences*. PhD thesis, University of Helsinki, 2002.
- [2] I. Rigoutsos and A. Floratos, “Combinatorial pattern discovery in biological sequences: The teiresias algorithm.,” *Bioinformatics*, vol. 14, no. 1, pp. 55–67, 1998.
- [3] A. Apostolico, F. Gong, and S. Lonardi, “Verbumculus and the discovery of unusual words,” *Journal of Computer Science and Technology*, vol. 19, no. 1, pp. 22–41, 2003.
- [4] H. J. Bussemaker, H. Li, and E. D. Siggia, “Building a dictionary for genomes: Identification of presumptive regulatory sites by statistical analysis,” *Proceedings of the National Academy of Sciences*, vol. 97, no. 18, pp. 10096–10100, 2000.
- [5] M. Thomas-Chollier, O. Sand, J.-V. Turatsinze, R. Janky, M. Defrance, E. Vervisch, S. Brohée, and J. van Helden, “Rsat: regulatory sequence analysis tools,” *Nucleic Acids Research*, vol. 36, no. suppl 2, pp. W119–W127, 2008.
- [6] G. Kahn, “The semantics of a simple language for parallel programming,” in *Information processing* (J. L. Rosenfeld, ed.), (Stockholm, Sweden), pp. 471–475, North Holland, Amsterdam, Aug 1974.

- [7] E. Lee and T. Parks, “Dataflow process networks,” *Proceedings of the IEEE*, vol. 83, pp. 773–801, may 1995.
- [8] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.
- [9] R. Hickey, “The Clojure programming language,” in *Proceedings of the 2008 symposium on Dynamic languages*, ACM New York, NY, USA, 2008.
- [10] D. L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Commun. ACM*, vol. 15, pp. 1053–1058, Dec. 1972.
- [11] M. Häußler and J. Nicolas, “Motif Discovery on Promotor Sequences,” Rapport de recherche RR-5714, INRIA, 2005.

# Appendix A

## spexs2

The program can be found at [github.com/egonelbre/spexs](https://github.com/egonelbre/spexs).

Several configurations can be found in the folder *examples*. It is best to start with an already existing configuration and modify it to your needs.

If the running *spexs2 -details* will print extended help about all the available features, filters, extenders.

### A.1 source

The source code in *src* has the following structure:

```
src/
├── spexs ..... algorithm definition
│   ├── extenders/ ..... query extenders
│   ├── features/ ..... query feature calculators
│   ├── filters/ ..... filter implementations
│   ├── pool/ ..... different queue implementations
│   ├── database.go ..... sequence dataset definition
│   ├── query.go ..... query definition
│   └── spexs.go ..... algorithm implementation
├── spexs2 ..... command-line utility
│   └── conf.go ..... configuration reader
```



- dataset.go ..... dataset reader
- features.go ..... parses and creates feature functions
- help.go ..... prints help for the program
- printer.go ..... prints the final output
- runtime.go ..... profiling and live-view setup
- setup.go ..... prepares everything for algorithm
- spexs2.go ..... main-entry point

There are also additional packages:

src/

- debugger/ ..... debugger for concurrent processes
- stats/ ..... statistical functions
  - binom/ ..... binomial p-value calculation
  - hyper/ ..... hypergeometric p-value calculation
- utils/ ..... additional utility functions
- bit/ ..... functions for bitmanipulation
- set/ ..... set implementations
  - hash/ ..... hash table with entry per value
  - bin/ ..... hash table with bitvectors
  - trie/ ..... 2-level hashtable with bitvectors

For compilation there are two scripts *make.bat* and *make.sh* that build the program into bin directory.

# Appendix B

## Concise Implementation

This is a concise reference implementation of the spexs2 algorithm. It is presented in Clojure[9].

```
1 (require '[clojure.core/reducers :as r])
2
3 ; a parallel grouping function
4 (defn group-map-by [g f coll]
5   (r/fold
6     (r/monoid (partial merge-with into) (constantly {})))
7     (fn [ret x]
8       (let [k (g x)]
9         (assoc ret k (conj (get ret k []) (f x)))))
10    coll))
11
12 ; these are the minimal requirements for a dataset
13 (defprotocol Dataset
14   (all [this] "return all possible positions on the dataset")
15   (walk [this pos] "return coll of Step from pos"))
16
17 (defrecord Query [pattern positions])
18 (defrecord Step [token position])
19
20 ; create an empty query for a dataset
21 (defn empty-query [dataset]
22   (Query. [] (all dataset)))
23
24 ; create a child query for parent given a token and positions
25 (defn child-query [parent [token positions]]
26   (Query. (conj (:pattern parent) token) positions))
27
28 ; declare our extension functions:
29 (defn walk-extend [dataset positions]
30   (let [steps (mapcat #(walk dataset %) positions)]
```

```

31     (group-map-by :token :position steps)))
32
33 ; function to combine multiple extension functions
34 (defn combine-extenders [extenders]
35   (fn [dataset positions]
36     (apply merge-with concat (map #(% dataset positions) extenders))))
37
38 ; finally the algorithm itself:
39 (defn spexs-step [ds q extend]
40   (map #(child-query q %) (extend ds (:positions q))))
41
42 (defn spexs [{
43   ds :dataset ; dataset
44   in :in      ; input coll
45   out :out     ; output coll
46   extend :extend ; position extender function
47   extend? :extend? ; query filter for further extension
48   output? :output? ; query filter for output
49 }]
50   (let [e (empty-query ds)]
51     (loop [in (conj in e)
52            out out]
53       (if-not (empty? in)
54         (let [[q & qs] in
55               queries (spexs-step ds q extend)
56               new-in (concat qs (filter extend? queries))
57               new-out (concat out (filter output? queries))]
58           (recur new-in new-out))
59         out))))
60
61 ; here is an example how to implement a dataset
62 (defn posify [row-index row-item]
63   (map (fn [pos] [row-index pos]) (range (count row-item))))
64
65 (defrecord SequenceDataset [items]
66   (token [this [row pos]]
67     (nth (nth (:items this) row) pos))
68
69   Dataset ; satisfy dataset interface
70   (all [this]
71     (mapcat posify (range) (:items this)))
72   (walk [this [row i]]
73     (let [row-item (nth (:items this) row)]
74       (if (> (count row-item) i)
75         [(Step. (token this [row i]) [row (inc i)])]
76         []))))
77
78 ; and how to use
79 (def simple-dataset (SequenceDataset. ["ACGT" "CGATA" "AGCTTCGA" "GCGTAA"]))
80
81 (spexs { :dataset simple-dataset :input [] :output []
82         :extend walk-extend

```

```
83      :extend? #(> (count (:positions %)) 3)
84      :output? #(> (count (:pattern %)) 2))}
```

**Non-exclusive licence to reproduce thesis and make thesis public**

I, Egon Elbre (date of birth: July 27, 1988),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
  - (a) reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
  - (b) make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, "Parallel Pattern Discovery", supervised by Jaak Vilo.
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, May 5, 2013