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 12, 2013

Contents

Chapter 1

Introduction

1.1 Motivation and background

One of the problems arising in dataset analysis is the discovery of interesting patterns. These patterns can show how the dataset is formed, how it repeats itself or they can be characteristic to some particular subset of the data.

For example a protein motif in a genomic sequence could predict disease. Patterns in medical diagnoses could show relations between diseases. Repeating pattern in source code could show how code could be minimized.

Research in pattern discovery is mainly driven by biology, which means most of the discovery algorithms have been designed for genomic sequences in mind. The techniques are usually constrained to the genomic sequences, but these algorithms could be useful elsewhere. The algorithms probably could be useful in other fields as well.

With genomic sequences there is another problem, the amount of data[GC95]. The data collected are growing with increasing speed, which means we need to use more computational resources to analyse them. This means the pattern discovery algorithms should take advantage of multicore processors, highly parallel processors and clusters.

write a priori vs some prior knowledge

write broad examples from nat processing, code

The usual limitations of algorithms are in the patterns themselves.

write usual limitations, alphabet, pattern language...

1.2 Algorithm parallelization

Taking an existing algorithm and making it parallel can be sometimes easier than writing an parallel algorithm from scratch. Existing algorithms may have already proven themselves in practice and have already good concepts. Algorithm parallelization can be divided into three subproblems:

1. generalizing the algorithm,
2. decomposing to independent tasks and
3. reifying the generalized version with parallelization in mind.

Generalizing the algorithm means loosening the order constraints and using minimal abstract data types for data storage. Here mathematical definitions and formulation of the problem is helpful. The less constraints there are the more freedom we have to change the implementation side.

Decomposing the algorithm means dividing it into independed tasks that could be ran in parallel. This also means trying to minimize the interaction and dependencies that "algorithm pieces" have.

Reifying the algorithm means finding suitable data structures for parallelization and mapping the independent tasks to different processes. The suitable structures and mapping to processes is dependent on the target architecture. For example data-structures involving vector operations work better on highly parallel processors.

Generalizing and decomposition steps can also make the algorithm simpler. Generalization makes algorithm more applicable to other fields, since there are less dependencies on the original problem.

1.3 Contributions of this work

We have derived a new parallel algorithm called SPEXS2 for discovering interesting patterns from a set of sequences. We describe SPEXS2 in a generic way and show how to possibilities of extending it further.

The practical and "ideal" versions of an algorithm can often diverge due to performance and implementation details, therefore we also explain problems and possible solutions with implementing such algorithm. We also have provided a concise implementation of the algorithm that captures the generic description more closely. Then we show some possible applications for the algorithm and analyse parallelization benefits.

1.4 Structure of the thesis

In this thesis we explore an algorithm for parallel pattern discovery. We choose an existing algorithm SPEXS[Vil02] and show how it can be parallelized.

In Chapter 2 we introduce the terminology. In Chapter 3 we give an overview of already existing algorithms and discuss why SPEXS[Vil02] was chosen as a base for parallelization. We generalize the SPEXS algorithm in Chapter 4 and reify it in Chapter 5. We discuss an implementation of the parallelized algorithm in Chapter 6 and in Chapter 7 show its possible applications and performance characteristics. The conclusions are presented in Chapter 8.

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 Σ 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 Σ. If the length of the string is 0, it is called an empty sequence or ϵ.

Example 2.1.1. `ACGTGCCATC` is a sequence where $\Sigma = \{$ `A`, `C`, `G`, `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 { [`This is an example`], [`This is an other example`] } and the alphabet is $\Sigma = \{$ `this`, `is`, `an`, `example`, `other` $\}$.

2.2 Pattern

Our aim is to discover repetetive 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 α is a structure then we say that structure α *matches exactly* pattern p. We say that α *matches* p if any of structure $all(p)$ is a sub-structure of α.

In this thesis we only consider sequential pattern structures and use *pattern* to mean *sequential pattern structure*. One very common way of describing a patterns is with regular expressions[Kle51; Wik13]. In this thesis we use regular expressions to describe the patterns and introduce the syntax where needed.

Pattern size is the length of the pattern sequence.

Example 2.2.1. `.[AT]` is a pattern of size 2 and denotes a set { `AA`, `AT`, `CA`, `CT`, `GA`, `GT`, `TA`, `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 $q =< D, p, matches(p, D) >$ is called a *query*.

Example 2.3.1. Let out dataset be $D = [$ `ACGT`, `TXCGA` $]$ and our pattern be $p =$ `C.` . The corresponding query is $< D, p, \{[1,3],[2,4]\} >$, 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 $f : Query \mapsto Any$.

We also need to see how "interesting" one query is compared to the others. *Query interestingness* is a function $f : Query \mapsto Value$ 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 $f : Query \mapsto Boolean$ and shows whether the query matches the criteria.

Example 2.3.2. Pattern occurences 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

Approaches to Pattern Discovery

Work in progress

In this chapter we give a overview of different algorithms used for pattern discovery.

Reorganize somehow

3.1 Algorithms

Overview of different combinatorial algorithms.

3.1.1 SPEXS

SPEXS is an pattern discovery algorithm described in "Pattern Discovery from Biosequences"[Vil02]. 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 ← new node
4: Root.label ← ϵ
5: Root.pos ← (1,2,...,n)
6: enqueue(Q, Root, order)
7: while N ← dequeue(Q) do
8:     Create all possible extensions p ∈ 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 ← p
12:            calculate F(p, S)
13:            enqueue(Q,p,order)
14:            if p fulfills the output criteria then
15:                store p in output queue O
16: Report the list of top-ranking patterns from output queue 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

TEIRESIAS[RF98] is an algorithm for the discovery of rigid patterns in biological sequences.

write more

TEIRESIAS operates in two phases: scanning and convolution. Scanning phase identifies elementary patterns that are frequent. During convolution these elementary patterns are combined to make larger patterns.

This method is a divide and conquer method to only consider frequent patterns.

patterns with any symbol

write more

3.1.3 Verbumculus

Verbumculus[AGL03] is...

statistical analysis, pattern matching

no complex patterns

write more

3.1.4 MobyDick

MobyDick[BLS00]... statistical prediction of frequent

no complex patterns

write more

3.1.5 RSAT

RSAT[Tho+08] is ...

matrix based pattern

write more

3.1.6 Other

[RSK09; Jen+06]

3.2 Reviews

[P+00; DD07; SD06]

write about some reviews

3.3 Problems

Work in progress

Algorithms are fixed and hard to extend with new pattern types, structures and optimizations. Generalization usually comes at the cost of performance and complexity.

write more

Sequential algorithms do not take advantage of multicore processors.

write more

Data that exceeds computer memory can't work efficiently... can't be distributed efficiently.

write more

Chapter 4

SPEXS Generalization

In this chapter we show how to make SPEXS algorithm more abstract by allowing flexibilty 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:     prepare(in, dataset)
3:     while q ← in.pop() do
4:         extended ← extend(q, dataset)
5:         for qx ∈ extended do
6:             if extend?(qx) then
7:                 in.push(qx)
8:                 if output?(qx) then
9:                     out.push(qx)
```

When the algorithm starts by initializing the *in* pool. The *in* pool shall contain queries which we wish to further examine. In the simplest case this means we create an empty pattern query and put it into the *in* pool. We could also start the process with an already existing pattern.

As the next step 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

Since pools act independently from the rest of the algorithm they are free to reorder, store on disk or even discard the queries, if needed. If we wish to get 100 best results the output pool could immediately discard the bad ones.

We can also use different types of structures as pools. For example using a queue would make it start examining breadth first, using a stack would make it run depth first. We can use priority queue to choose the best queries to reach faster the good results as suggested in "Patterns Discovery from Biosequences"[Vil02].

4.3 Filtering

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

The filters can make the decision, whether the query should be extended, based on any available information. For example query pattern, number of occurences, metadata in sequences, metadata in dataset or use data from configuration.

Example 4.3.1. We can add metadata to the sequence about the datafile. We count the occurences of a query in each datafile and then if the ratio between counts is smaller than some number defined in the configuration.

Although there is only one filter "function" specified the filter could be a composite of multiple filters.

Example 4.3.2. Pattern length is greater than three and pattern occurs at least 10 times in the dataset can be seen as a single filter that is composed of two filters.

4.4 Extending

The extending process is at the core of the algorithm and there are several ways of doing it. The main criteria is that the extending should guarantee that all possible patterns get eventually enumerated.

Extender is analogous to an inductive step. Our base case is formulated by *prepare* step in the SPEXS2 algorithm and the induction steps are carried out by the extender.

Example 4.4.1. We start with an empty query and we know all the locations of it. If our extender generates all the queries where the patterns are longer by 1 then we are guaranteed to enumarate all the patterns.

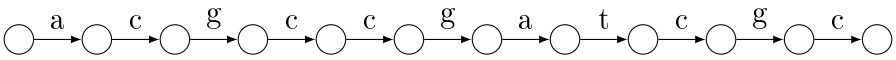
Example 4.4.2. We can start with the empty query and all queries with patterns of length 1. Now if our extender generates queries where the patterns are longer by 2 we can also examine all of the queries.

The extender determines which patterns and pattern classes will be generated. We can modify and compose different extenders to get new patterns. Often more complex patterns can adversely affect performance.

The general idea for extending is to find all the following patterns from all the previous query positions and then group the similar patterns into queries.

This process can be visualized with graphs. We make the sequence into a graph where the links between nodes are the sequence tokens. Each pattern then can walk the edges and match find the ending position for each pattern.

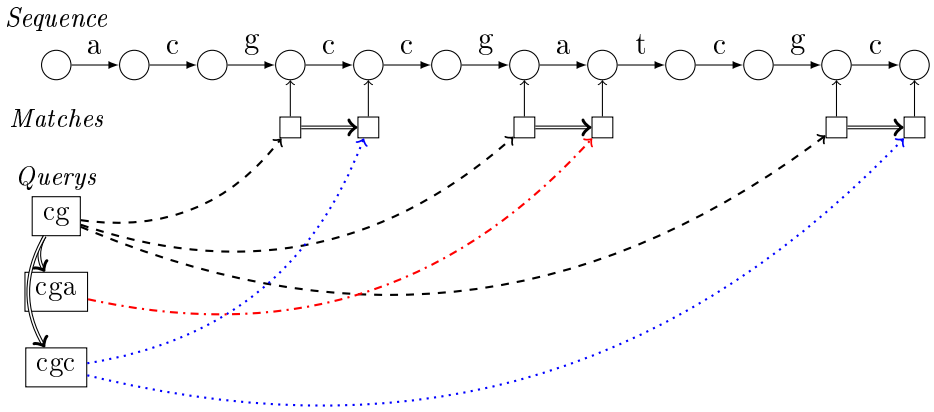
For example the sequence `ACGCCGATCGC` would look like:



For simplicity we use nucleotides as our alphabet $\Sigma = \{$ `A`, `C`, `G`, `T` $\}$ in the following examples.

4.4.1 Sequences

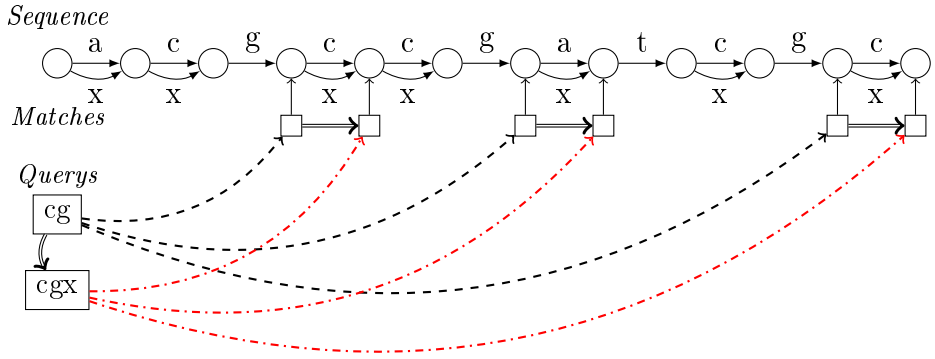
The simplest case how the *next* function behaves is when we are only looking for simple sequences. Let's consider a sequence `ACGCCGATCGC` and a pattern `CG`.



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 Group tokens

One common addition in a pattern language is matching 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.



There is a universal group `.` or *wildcard* that matches any token in the alphabet.

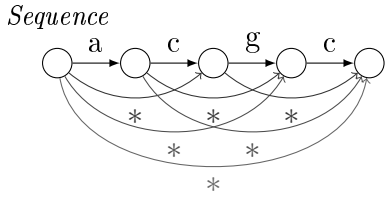
Example 4.4.3. Pattern `T.` would match patterns `TA`, `TC`, `TG`, `TT`.

4.4.3 Star

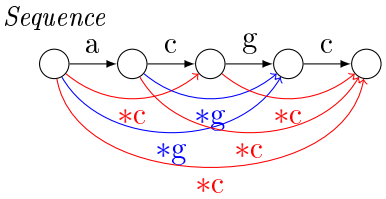
Another possible extension is capturing a run of wildcards.

Example 4.4.4. A pattern `A.*T` would match `ACT`, `ATTC`, `ATTTC` and so on.

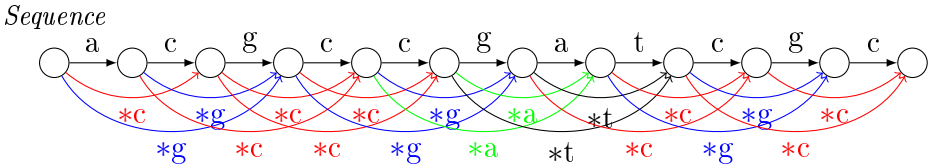
On the graph instead of `.*` we use only `*` symbol.



Constructing more complicated patterns increases the amount of queries required to enumerate. There are of course optimizations to avoid intermediary steps and repeated walking on the dataset. For example we can skip the extension with only `.*` and instead extend with `.*Y`, where `Y` is a token from the alphabet. This means we avoid this one large query and instead have multiple smaller queries.



We can limit the length of the run to speed up the process. Limiting the run length to be either 2 or 3 would look like:



4.4.4 Optimized group tokens

Instead of immediately extending the group tokens we can take the output of an other extender and combine its results. If we have a group token γ that contains $tokens(\gamma)$ then the *matches* for such group is

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

.

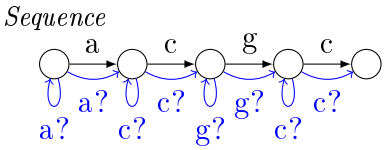
Example 4.4.5. A pattern `A[CTG]` is located in document D at positions $matches($`AC`$, D) + matches($`AT`$, D) + matches($`AG`$, D)$.

4.4.5 Other possible extensions

By adding a multi-step from a node to itself and to the next node we can capture optional tokens.

Example 4.4.6. A optional token `Y?` means that the token `Y` can occur either zero or one time. For example `AT?` matches `A` and `AT`.

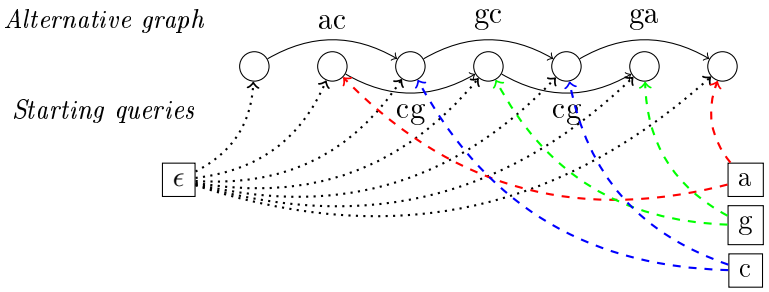
The graph for such token would look like:



Similarly we can use the same technique for optimization as for the group tokens ($matches(pY, D) = matches(p) \cup matches(pY)$, where Y is a token).

4.4.6 Alternative extensions

We mentioned that we can also extend by different number of tokens at a time as long as we guarantee that all patterns will be searched. For optimality we also do not want to iterate over the same pattern multiple times. As a simple example the sequence `ACGCGA` could be iterated with setup:



Since at the starting point we have all the patterns of length 0 and 1, by adding patterns with length 2 we can be sure to enumerate all of them. Of course the previous methods for group and star patterns require modification.

4.5 Summary

The extender was shown to work via graphs, practically it is much more reasonable to minimize it as simple sequence as already mentioned in "Pattern Discovery from Biosequences"[Vil02]. The additional extension links shown in the graphs can either be precalculated or calculated at runtime.

From the previous results we can also derive the minimal requirements for the dataset. First we need to get the initial empty query - which means we should be somehow be able to get all the positions where a pattern could start. The other operator is finding the next position and token from a given position. Finding of next positions from a given positions on sequence can be interpreted as a forward iterator.

Since the best way to visualize was on graphs suggests that the *spexs2* algorithm could be made to work on trees and then on graphs. Finding sequential patterns from a tree should be straightforward since the generic algorithm is oblivious to the amount of following tokens any position can have. Graphs can be more difficult since we need to remove duplicates caused by other extensions.

To use this generic version of the SPEXS algorithm we need to 1. choose our pool structures, 2. choose our filters, 3. choose our extender and preparation and 4. dataset implementation.

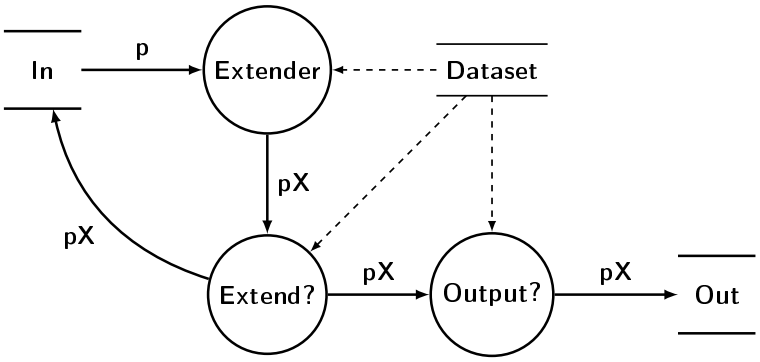
Chapter 5

Parallelization

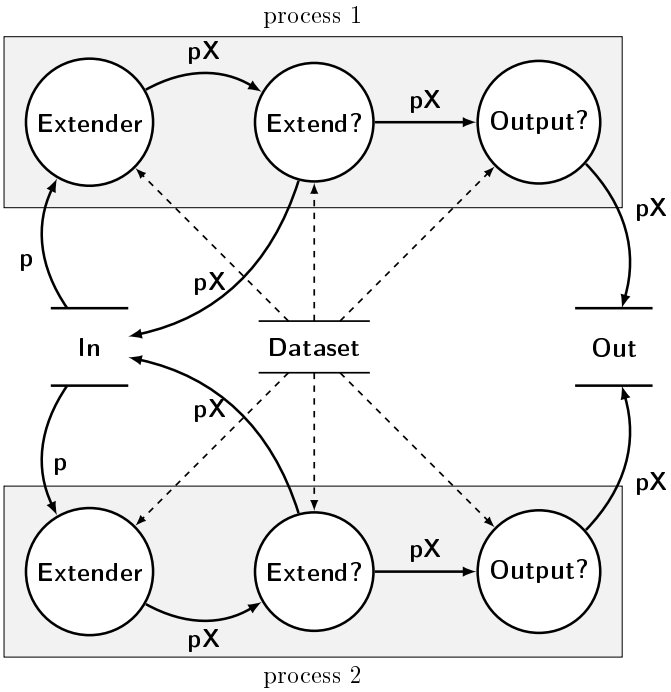
Here we discuss different ways how we can reify the algorithm to support parallelism. There are often several ways of making program parallel. Using parallelization means that there is additionally a need for some communication and synchronization to make the processes arrive at the final result. So it is useful to find as many possible parallelizations, but it is not wise to use all of them.

5.1 Process

The main process of the algorithm as described by dataflow diagram.[Kah74; LP95] 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 such concurrency will introduce a source of indeterminism.

5.2 Extending

The extender can be parallelized via map and reduce concepts[DG08; Jr09]. The extender was based on two concepts finding the *next* positions from previous pattern position and then group those positions together to find the next queries.

The finding the next position from a position can be easily implemented

via mapping by using the *next* function of the dataset. The grouping requires some attention - the grouping is itself a reduction into a map by key with joining.

The pseudo-code representation for such function compositions would be very difficult and would require a lot of new syntax. Therefore we present this idea in Clojure[Hic08] which should be readable to people who know lisp. We use the reducers library to show how the extension can be implemented.

Algorithm 3 Parallel extender

```
(require '[clojure.core.reducers :as r])

; fold-join based grouping function
(defn group-map-by [g f coll]
  (r/fold
   (r/monoid (partial merge-with into) (constantly {}))
   (fn [ret x]
     (let [k (g x)]
       (assoc ret k (conj (get ret k []) (f x)))))
   coll))

(defn extend [dataset query]
   (let [ steps   (r/mapcat #(walk dataset %) (:positions query))
          grouped (group-map-by :token :position steps)]
        (r/map #(child-query q %) grouped))))
```

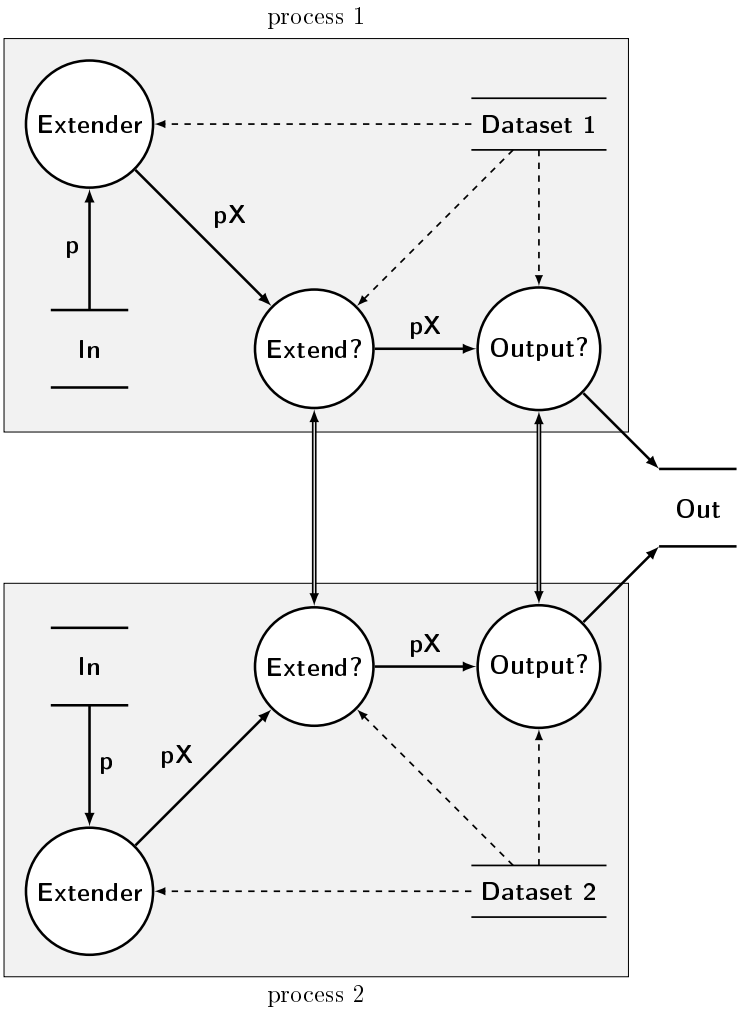
Such approach may not give much improvement on desktop CPUs, since we already can process multiple queries at the same time. This parallelization could be benefitial for highly parallel processors such as GPGPUs or FPGAs.

5.3 Distributed processes

Since the dataset and the process memory consumption can get quite large it would also be benefitial to be able to partition the dataset between multiple machines. This can also help on a single machine since we can interlace running the different processes and store the non-running process in non-volatile memory.

The extension results for a given query stays inside the sequence which means we can partition by assigning 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 calculate partial results and let the filters communicate the results. This could be also done in a seprate process instead of directly communicating.



Example 5.3.1. 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 each other and add these results together locally. Depending on that result we know how to proceed.

Such distribution could be used to separate the process into more managable chunks, but adds significant communication overhead.

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 A.

6.1 Language

The language choice in a project is very subjective. Usually the language chosen is either the language that the author feels most comfortable with or widely used language that has many libraries. This project used Go for practical implementation and here we discuss the *opinions* why such choice was made. The information here was collected from several sources[Mac13; Ful13; Hun11; GPT13; Hic08; Bez+12] and should be seen as opinions rather than facts.

This project should be a reference implementation and should be as readable as possible; this means that languages such as Fortran, C++, Java are probably not the best choices due to their complexity. The project deals with a lot of generic code; this means Clojure, Haskell and OCaml would be

useful, but it would require a lot of effort if someone wishes to contribute and doesn't know the language. Languages such as Matlab, Octave or R would be ideal due to their ease of use, but their speed or memory performance is worse than C.

There are newer languages Go, D and Julia that suit the problem better. D is an high performance language that was designed as an replacement for C++, but still has a steep learning curve. Julia is a high-performance dynamic language designed for technical computing, but at the time of starting the project it had significant language runtime bugs. Go is a systems language designed to be simple, but it is worse in performance than D and Julia.

Go seemed to be the best choice due its simplicity, stability and concurrency support. The performance characteristics also seemed good enough. Language simplicity has several benefits: ease of learning, readability and portability. Ease of learning means that if some new feature is needed a contributor can learn the language fast. Readability means that it is a better reference implementation. Portability means that most of the language semantics and structures can be directly translated to other languages if there is a need for better performance.

In summary, Go is a nice compromise between Python and C. It adds additional rigidity and concurrency primitives and is a simple language.

6.2 Architecture

The main criteria for designing program have been described in D. Parnas paper "On the Criteria To Be Used in Decomposing Systems into Modules"[Par72]. It suggests decomposing into isolated units and parts that are likely to change together.

We chose the following module 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

The program starts by interpreting flags, then it marshals configuration file onto an internal data structure, this configuration structure as an input to the setup module. Setup module initializes (as defined by configuration) a reader, a printer and also prepares structures for algorithm. Then the reader reads input to the database. Then the algorithm is activated and finally it is printed out with Printer.

This structure is universal for algorithm implementations and allows easily to add more configuration options, different input formats and different output formats. By changing the configuration, reader and printer we could make it a web service instead of running it on a command line.

6.3 Configuration

One problem with flexible algorithms is that there are a lot of ways to be run. This can lead to having tens or hundereds of command-line flags for the application.

To avoid this problem we decided to use a *json* file for the program configuration. This format is widely known and well structured. For example a configuration file for pattern discovery in protein sequences:

```
1  {
2      "Dataset": {
3          "fore" : { "File" : "$inp$" },
4          "back" : { "File" : "$ref$" }
5      },
6      "Reader" : {
7          "Method" : "Delimited"
8      },
9      "Extension": {
10         "Method": "Group",
```

```
        "Groups" : {
            "." : { "elements" : "ACDEFGHIKLMNPRQSTVWY"}
        },
        "Extendable": {
            "PatGroups()" : {"max" : 3},
            "PatLength()" : {"max" : 6},
            "Matches(fore)" : {"min" : 20},
            "NoStartingGroup()" : {}
        },
        ...
    },
    "Output": {
        "SortBy": ["-Hyper(fore, back)"],
        "Count": 100
    },
    "Printer" : {
        "Method" : "Formatted",
        "Format": "Pat?()\tMatches(fore)\tMatches(back)\tHyper(fore,back)\n"
    }
}
```

In hindsight *json* for configuration may not be the best option due to rigidity. Users can often forget to add or remove a comma or forget to add quotes. This suggests that formats such as *rson* or *yaml* would be better choices.

Other problem with configuration files is that they are harder to modify than command-line flags. By mixing command-line flags and configuration files we can get a solution that works better in practice than either of them independently.

One easy way to implement is to add custom syntax into the configuration file:

```
    "Datasets" : {
        "fore" : { "File" : "$argument:default$"
        ...
```

Now some command-flags can be interpreted as replacements into the configuration file. Using `spexs2 -conf conf.json argument=other` would transform the configuration file into:

```
    "Datasets" : {
        "fore" : { "File" : "other"
        ...
```

If no such parameter is given then the default value "main" can be used.

Configuration files are also problematic for running the first time. As a user you need to find a configuration file that suits your needs, then modify it and finally run it. To remedy this problem the application can embed sample configuration files so called "profiles" that can be used directly from the command line. This means you can directly use `spexs2 -p=protein input=some.data min-p=1.0`.

6.4 Alphabet and Database

spexs2 was designed also to work with texts and words. To support such large alphabets we first needed to map each token to an identifier (an integer) and convert the sequences to a sequence of these identifiers. Unfortunately this is problematic for large datasets since this can take a lot of time as compared to just copying the file to memory.

There are analyses that may require many datasets. For example instead of comparing two datasets at a time you may want to compare pair-wise multiple datasets at the same time. Supporting multiple datasets in the code is rather trivial, but exposing this to the user is more interesting.

To support multiple datasets we first added a name for each dataset in the configuration:

```
1     "Datasets" : {
2         "A" : { "File" : "$A$" },
3         "B" : { "File" : "$B$" },
4         "C" : { "File" : "$C$" },
5         ...
```

We also use the command-line argument syntax to make it easier to use. Now when we are defining filters we need to specify how exactly they are calculated. For example we can not just say ratio of occurrences in dataset since that would be ambigous. We used syntax similar to function calls in the configuration file:

```
1     ...
2     "Extension": {
3         "Outputtable": {
```

```
4               "OccurencesRatio(A, B)" : {"min" : 2},
5               ...
```

This allows to clearly see that the occurences ratio between datasets A and B must be at least 2. Obviously we can also define features that take more arguments.

6.5 Pools

The input pool can direct the flow of extending process. This means that the order of can have performance and memory impact. The performance doesn't get affected as much because the algorithm is exhaustive and hence every query gets examined, unless it is terminated early or the filters tuned during runtime.

There are several ways run the extension process: breadth first, depth first, most frequent first, least frequent first and others. Breadth first and depth first can easily be achieved by a queue and a stack respectively. We can use priority queues to implement other ways of extending.

The major concern was running on large datasets which meant that the memory consumption important. Although the most frequent first can arrive to the interesting results faster it can also use a lot of memory due to unextended less frequent nodes.

The least frequent approach would minimize the memory use since the least frequent would be most likely to be discarded by filters. This requires the use of a priority queue. The depth first approach would also use little memory but can use a stack.

Since we need concurrent access to the pool it requires very fast adding and removing operations. This means that the depth-first ordering is more suitable since adding and removing from stack is faster than from priority queues.

For the output queue a priority queue is the obvious choice since we usually only want a limited number of the most interesting results.

6.6 Query and Location set

The structure of query required some special consideration. The proposed solution in "Pattern Discovery from Biosequences" was to use a trie for remembering each pattern and then use optimized set for storing locations inside the database. Since the algorithm must now work concurrently using a simple trie was not an option because adding a child to the parent from multiple processes causes a race condition.

The first approach was to flip the trie, which means instead of parent pointing to the child the child points to the parent. The original tree can be extracted by reversing the links after the algorithm has finished. This started causing problems since we are working in a garbage collected enviornment and each pointer adds overhead to the garbage collection. Rough estimation also suggested that memory benefit from using trie is minimal.

The second approach was just to copy the any needed content to the child and not to link them. This worked out very well.

The other problem was how to store the position list. Initially it seemed that a very tightly packed set structure is required to keep the memory usage of the program minimal. This of course would have impacted the performance. This actually turned out not to be the case since all the queries could be packed with any packing algorithm when they are stored in the pools and that had similar memory benefit with a simpler code.

One interesting optimization we found was a related to storing sparse positions. A simple way to implement large bitset is using a hash map to a small bitarray. We use the first bits as the key and the rest store as a position in the bitarray. We know that the occurrences of a pattern are likely to be sparse, hence it is also quite likely there are only bitarrays where only a single bit is set. The soltion is to move last bits to the front of the integer. This makes the last bits to be less uniform and hence the packing will be better. Rough memory testing showed that this made the structure use about 2x less memory.

6.7 Query features, interestingness and filters

When we first described the query features we showed that filters and interestingness are a special case query features. In *spexs2* the features are used to print information about the results.

Since most of the features implemented could also be used as a interestingness measure we used a simplification for the "Feature" function definition:

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

Which means that the function returns two types, a real value and a string. In the implementation there are only few features that return arbitrary types so it was easier to convert it into a string. The only place where such features were needed is for printing. For example one of such features is the representation of the pattern.

And an implementation of a Feature constructor:

```
// the count of matching sequences
func Matches(datasets []int) Feature {
    return func(q *Query) (float64, string) {
        matches := q.Matches()
        return countf(matches, group), ""
    }
}
```

The argument to the function defines over which datasets the resulting function counts matches. To bind this constructor to the configuration we register it and with reflection use the function name as the name used in the configuration file.

The filters can be very similar to features in their implementation. For example a filter for pattern length is similar to the pattern length feature. Although implementing all combinations is possible we can use function composition to avoid such repetition.

By specifying a minimum or a maximum value for a feature we can turn it into a filter. One way to do it is using a lexical closure. For example:

```
func MakeFeatureFilter(fn Feature, min float64, max float64) Filter {
    return func(q *Query) bool {
        v, _ := feature(q)
```

```
4             return (min <= v) && (v <= max)
5         }
6     }
```

Of course there are some filters that cannot be defined by features hence there is still possibility to make separate filters. For example disallowing a wildcard in the beginning of the pattern.

In languages which do not support such composition we could use object composition and/or function pointers.

6.8 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 4 Tree traversal

Output: All nodes in tree get processed with fn

```
 1: function VISIT(tree, start, fn, examine?)
 2:     unvisited ← { start })
 3:     start workers
 4:         while unvisited not empty do
 5:             node ← unvisited.take()
 6:             fn(node)
 7:             for child ∈ 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 5 Synchronized graph traversal

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

```
 1: function VISIT(graph, start, fn, examine?)
 2:     added ← new semaphore(0)
 3:     terminate ← false
 4:     mutex ← new mutex()
 5:     workers ← 0
 6:     unvisited ← { 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 ← unvisited.take()
17:             workers ← workers + 1
18:             mutex.unlock()

19:             fn(node)

20:             for child ∈ 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 ← workers − 1
28:             if workers = 0 and unvisited = {} then
29:                 terminate ← 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.

6.9 Debugging

Seeing how the algorithm runs 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 such tracing is implemented by adding debug statments.

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
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 "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 measurements

The version used were *spexs2.1.1@f1d6f7 (spexs2)* compiled with *go1.1rc3* and *spexs.0.2.a01*.

Unfortunately I was unable to run *spexs* with large datasets 5e6 protein sequences.

The original *spexs* works faster in 1 thread case which is to be expected due to runtime and language differences.

Chapter 8

Conclusions

In this thesis we analysed how to develop a parallel pattern discovery algorithm. We showed how we can take an already exisiting algorithm and parallelize it by generalizing, decomposing and then reifying. Finding the general idea of the algorithm can simplify the algorithm and provide more intuitive way of interpreting it. Decomposing allows us to talk about separate parts of algorithm and modify them without affecting the general idea of the algorithm. If we have an abstract algorithm we can substitute those parts with parallel structures and algorithms.

As a practical part we implemented a parallelized algorithm based on *spexs*[Vil02]. We discussed several problems of implementing an algorithm and discussed interesting approaches to these problems. The program has been designed to be easily extendable for different inputs, filters and interestingness criteria. We discussed different possible uses for the implementation and analysed the performance gained from parallelization.

Further work on *spexs2* should investigate the best configurations for different applications. We discussed that *spexs2* could be made to find tree and graph patterns, it could be an interesting research topic.

Approaches suggested in this thesis could be used to generalize and parallelize other algorithms. Finding generic algorithms can be an easy way of discovering new optimizations, new algorithms and new potential applica-

tions for algorithms. If these generalizations can be implemented practically we make the implementation easily extendable and also usable for wider range of problems.

Bibliography

[AGL03] A. Apostolico, F. Gong, and S. Lonardi. "Verbumculus and the Discovery of Unusual Words". In: *Journal of Computer Science and Technology* 19.1 (2003), pp. 22–41. URL: http://www.cs.ucr.edu/~stelo/papers/verb03.ps.gz.

[Bez+12] Jeff Bezanson et al. "Julia: A Fast Dynamic Language for Technical Computing". In: *CoRR* abs/1209.5145 (2012).

[BLS00] Harmen J. Bussemaker, Hao Li, and Eric D. Siggia. "Building a dictionary for genomes: Identification of presumptive regulatory sites by statistical analysis". In: *Proceedings of the National Academy of Sciences* 97.18 (2000), pp. 10096–10100. DOI: 10.1073/pnas.180265397. eprint: http://www.pnas.org/content/97/18/10096.full.pdf+html. URL: http://www.pnas.org/content/97/18/10096.abstract.

[DD07] Modan Das and Ho-Kwok Dai. "A survey of DNA motif finding algorithms". In: *BMC bioinformatics* 8.Suppl 7 (2007), S21.

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. ISSN: 0001-0782. DOI: 10.1145/1327452.1327492. URL: http://doi.acm.org/10.1145/1327452.1327492.

[Ful13] Brent Fulgham. *The Computer Language Benchmarks Game*. [Online; accessed 12-May-2013]. 2013. URL: http://benchmarksgame.alioth.debian.org/.

[GC95] Mark S Guyer and Francis S Collins. "How is the Human Genome Project doing, and what have we learned so far?" In: *Proceedings of the National Academy of Sciences* 92.24 (1995), pp. 10841–10848.

[GPT13] Robert Griesemer, Rob Pike, and Ken Thompson. *The Go Programming Language*. [Online; accessed 12-May-2013]. 2013. URL: http://golang.org/.

[Hic08] R. Hickey. "The Clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM New York, NY, USA. 2008.

[Hic12a] R. Hickey. *Anatomy of a Reducer*. [Online; accessed 12-May-2013]. May 2012. URL: http://clojure.com/blog/2012/05/15/anatomy-of-reducer.html.

[Hic12b] R. Hickey. *Reducers - A Library and Model for Collection Processing*. [Online; accessed 12-May-2013]. May 2012. URL: http://clojure.com/blog/2012/05/08/reducers-a-library-and-model-for-collection-processing.html.

[HN05] Maximilian Häußler and Jacques Nicolas. *Motif Discovery on Promotor Sequences*. Anglais. Rapport de recherche RR-5714. INRIA, 2005, p. 136. URL: http://hal.inria.fr/inria-00070303.

[Hun11] Robert Hundt. "Loop Recognition in C++/Java/Go/Scala". In: *Proceedings of Scala Days 2011*. 2011. URL: https://days2011.scala-lang.org/sites/days2011/files/ws3-1-Hundt.pdf.

[Jen+06] Kyle L. Jensen et al. "A generic motif discovery algorithm for sequential data". In: *Bioinformatics* 22.1 (2006), pp. 21–28. DOI: 10.1093/bioinformatics/bti745. eprint: http://bioinformatics.oxfordjournals.org/content/22/1/21.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/22/1/21.abstract.

[Jr09] Guy L. Steele Jr. "Organizing functional code for parallel execution or, foldl and foldr considered slightly harmful". In: *ICFP*. 2009, pp. 1–2.

[Kah74] G. Kahn. "The semantics of a simple language for parallel programming". In: *Information processing*. Ed. by J. L. Rosenfeld. Stockholm, Sweden: North Holland, Amsterdam, Aug. 1974, pp. 471–475.

[Kle51] Stephen Cole Kleene. "Representation of events in nerve nets and finite automata". In: (1951).

[LP95] E.A. Lee and T.M. Parks. "Dataflow process networks". In: *Proceedings of the IEEE* 83.5 (May 1995), pp. 773–801. ISSN: 0018-9219. DOI: 10.1109/5.381846.

[Mac13] David R. MacIver. *Hammer Principle*. [Online; accessed 12-May-2013]. 2013. URL: http://hammerprinciple.com/.

[P+00] Pavel A Pevzner, Sing-Hoi Sze, et al. "Combinatorial approaches to finding subtle signals in DNA sequences". In: *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*. Vol. 8. 2000, pp. 269–278.

[Par72] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Commun. ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: http://doi.acm.org/10.1145/361598.361623.

[RF98] I Rigoutsos and A Floratos. "Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm." In: *Bioinformatics* 14.1 (1998), pp. 55–67. DOI: 10.1093/bioinformatics/14.1.55. eprint: http://bioinformatics.oxfordjournals.org/content/14/1/55.full.pdf+html. URL: http://bioinformatics.oxfordjournals.org/content/14/1/55.abstract.

[RSK09] Pedro Ribeiro, Fernando Silva, and Marcus Kaiser. "Strategies for network motifs discovery". In: *e-Science, 2009. e-Science'09. Fifth IEEE International Conference on*. IEEE. 2009, pp. 80–87.

[SD06] Geir Kjetil Sandve and Finn Drablos. "A survey of motif discovery methods in an integrated framework". In: *Biol Direct* 1.11 (2006).

[Tho+08] Morgane Thomas-Chollier et al. "RSAT: regulatory sequence analysis tools". In: *Nucleic Acids Research* 36.suppl 2 (2008), W119–W127. DOI: 10.1093/nar/gkn304. eprint: http://nar.oxfordjournals.org/content/36/suppl_2/W119.full.pdf+html. URL: http://nar.oxfordjournals.org/content/36/suppl_2/W119.abstract.

[Vil02] Jaak Vilo. "Pattern Discovery from Biosequences". PhD thesis. University of Helsinki, 2002.

[Wik13] Wikipedia. *Regular expression — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-May-2013]. 2013. URL: http://en.wikipedia.org/wiki/Regular_expression.

Appendix A

spexs2

The program can be found at `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

Reference Implementation

This is a concise implementation of the parallel spexs2 algorithm. It is presented in Clojure[Hic08].

```
(require '[clojure.core.reducers :as r])

; a parallel grouping function
(defn group-map-by [g f coll]
  (r/fold
   (r/monoid (partial merge-with into) (constantly {}))
   (fn [ret x]
     (let [k (g x)]
       (assoc ret k (conj (get ret k []) (f x)))))
   coll))

; these are the minimal requirements for a dataset
(defprotocol Dataset
  (all  [this]     "return all possible positions on the dataset")
  (walk [this pos] "return coll of Step from pos"))

(defrecord Query [pattern positions])
(defrecord Step [token position])

; create an empty query for a dataset
(defn- empty-query [dataset]
  (Query. [] (all dataset)))

; create a child query for parent given a token and positions
(defn- child-query [parent [token positions]]
  (Query. (conj (:pattern parent) token) positions))

; declare our extension functions:
(defn walk-extend [dataset positions]
  (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                querys (spexs-step ds q extend)
56                new-in  (concat qs  (filter extend? querys))
57                new-out (concat out (filter output? new-in))]
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 12, 2013