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

Author:	.....	"..."	May 2013
Supervisor:	.....	"..."	May 2013
Professor:	.....	"..."	May 2013

Tartu May 20, 2013

Contents

Chapter 1

Introduction

1.1 Motivation and background

An interesting research problem in dataset analysis is the discovery of patterns. Patterns can show how the dataset was formed and how it repeats itself; can also be characteristic to some particular subset of the data.

For example, a protein motif in a genomic sequence could predict a disease. Patterns in medical diagnoses could show relations between diseases. A repeating pattern in source code could show how the code could be minimized. Patterns in event logs could find causes for error events or detect intrusion attempts.

Research in pattern discovery is mainly driven by biology, which means that most of the discovery algorithms have been designed with genomic sequences in mind. The techniques are usually constrained to the genomic sequences, but these algorithms could be useful in other fields as well. Such benefit was already demonstrated by Wespi et al[WDD99], where they used a biological sequence pattern discovery algorithm for intrusion detection. Searching new ways of using pattern discovery should, thus, be actively researched.

Today, the field of genomic sequences is facing problems caused by the increasing amount of data[How+08; GC95]. We need to use more compu-

tational resources to analyze the continuously growing amount of collected data. This means that the pattern discovery algorithms should take advantage of multi-core processors, highly parallel processors and clusters.

1.1.1 Pattern Discovery

Patterns can occur in different types of data: images, text, sounds, sequences, graphs, signals and more. The pattern representation can vary depending on the data itself and the parts that the pattern captures. Patterns in graphs can be sub-graphs with some additional info. Patterns present in images can be a collection of different features.

Pattern discovery aims to find *a priori* unknown patterns that can be considered interesting in some specific aspects; for example, we could look for patterns that occur frequently or have an interesting structure. A pattern that occurs more frequently than expected by chance, may be considered interesting. Also, a graph pattern forming a ring graph may be considered interesting.

When choosing a pattern structure, we should take into account the data, expected results and its efficiency. For example, theoretically we could choose an abstract pattern structure that can represent all possible patterns, but practically this could be inefficient and irrelevant to the problem. Finding image features from a visualization of sound would probably not produce anything meaningful.

When choosing a pattern rating method we should take into account that many patterns can occur by chance. For example, a pattern that matches anything is very frequent although it isn't very meaningful. Many methods have been used for comparing patterns, such as pattern occurrences, ratio of pattern occurrences between datasets, binomial and hypergeometric probability estimate and Z-score.

1.1.2 Algorithm parallelization

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

1. generalizing the algorithm,
2. decomposing the algorithm into 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. Mathematical definitions and effective problem formulation are helpful in this matter. The less constraints there are, the more freedom we have to change the algorithm implementation details.

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

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

The generalizing and decomposition steps can also make the algorithm simpler. Generalization makes the algorithm more applicable to other fields, since there are less relations to the original pattern discovery problem.

1.2 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 some possibilities for 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 associated with implementing such an algorithm and possible solutions for them. 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 analyze the benefits of parallelization.

1.3 Structure of the thesis

In Chapter 2 we introduce the terminology used throughout the thesis. In Chapter 3 we give an overview of the already existing algorithms and discuss the reasons for choosing SPEXS[Vil02] as a basis for parallelization. We generalize and decompose 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 we show some possible applications and its parallelization benefits. 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 or any other fixed element.

Any sequence $S = a_1a_2...a_n, \forall a_i \in \Sigma$ is called a *sequence* over the alphabet Σ. We denote the *length* of sequence with $|S|$. If the length $|S|$ of the sequence S is 0, it is called an empty sequence or ϵ.

Example 2.1.1. `ACGTGCCATC` is a sequence over $\Sigma = \{$`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. The 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 repetitive and characteristic structures in data. We call such structures *patterns*. One generic way to define a *pattern* is as a set of all the sub-structures it represents. This means that 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 as sequences, graph patterns are represented as graphs; but in some situations sequence patterns could also be represented as graphs.

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 the structure α *matches exactly* the pattern p. We say that α *matches* p if any structure from $all(p)$ is a sub-structure of α.

In this thesis we only consider sequential pattern structures and use *pattern* in the meaning *sequential pattern structure*. One very common way of describing patterns is by using regular expressions[Kle51; Wik13a]. In this thesis we use regular expressions to describe the patterns and introduce the regular expression 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 matching positions in a dataset D as $positions(p, D)$. The exact representation of a single position depends on the pattern structure. For sequences it can be simply a tuple of the sequence number and the starting and end positions. For graphs this can be represented as a mapping of the pattern to the graph.

2.3 Query

We need to somehow understand where a given pattern p is located in the dataset D. This compound structure $q = \langle D, p, positions(p, D) \rangle$ is called a *query*. The words pattern and query are interexchangable. When talking about pattern parameters, then it makes sense to do so only in a specific context, so we are actually talking about the query.

Example 2.3.1. Let our dataset be $D =$ [`ACGT` , `TXCGA`] and our pattern be $p =$ `C.` . The corresponding query is $\langle D, p, \{[1, 3], [2, 4]\} \rangle$, which means that in the sequence 1, pattern p ends at position 3, and in the sequence 2, the pattern ends 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 takes a query as an input and returns 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. We can often represent such interestingness measures as a real number.

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

Example 2.3.2. Pattern occurrences in a document can be considered an interestingness measure. Whether a query pattern is at least 3 tokens long is a query filter.

2.4 Pool

Pool is an abstract data type for a storing the queries. 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 pools that store the queries on the disk; it could internally pack or reorder the queries for performance reasons.

2.5 Pattern Discovery

In this thesis, *pattern discovery* is defined as a process for finding, according to a query interestingness measure, the most interesting subset of sequential patterns that meet certain criteria in a sequence dataset.

Example 2.5.1. Let our research problem be "Finding the most common nucleotide patterns, which are at least 3 nucleotides long, from a shotgun sequencing output." *"Most common"* defines our interestingness measure. *"At least 3 nucleotides"* is the criteria for selecting a subset of patterns. *"Sequencing output"* is used as our dataset and *"nucleotides"* define how the sequence looks like.

Chapter 3

Approaches to Pattern Discovery

Pattern discovery algorithms can be organized in different ways[DD07; SD06; P+00; HN05]. In this chapter we give an overview of the different ideas; for thorough descriptions we suggest "Motif Discovery on Promotor Sequences" by M. Häußler and J. Nicolas[HN05] and "A survey of motif discovery methods in an integrated framework" by Sandve and Drabløs[SD06].

3.1 Algorithmic techniques

The algorithmic techniques can be largely classified into 1. pattern growth, 2. alignment-based and 3. probabilistic pattern discovery.

There are two basic ways to grow the patterns: iteration and combining. The iteration method uses a pattern and then starts iteratively expanding the pattern with new tokens. This approach can be very well optimized due to its simplicity, but it often requires more memory. The iterative approach can also have problems with larger patterns. The combining method first generates a set of simple patterns and then starts combining them to generate new patterns.

Alignment-based approaches work in two phases: 1. building a set of elementary patterns and 2. produce consensus pattern. The elementary patterns are usually very simple subsequences. The elementary patterns

are aligned and merged to a consensus pattern that best describes all the patterns. The patterns could also be clustered before producing consensus patterns. This approach is usually done by two separate tools; one to generate frequent patterns and the other to align the patterns.

Probabilistic algorithms use a statistical model to iteratively improve the pattern parameters to identify the real patterns until a stop criteria is met. Common statistical techniques are Expectation Maximization (EM) and Gibbs sampling.

3.1.1 Algorithms

In this section we describe algorithms that we consider interesting or important in their algorithmic structure or approach. The list here is by no means exhaustive.

SPEXS [Vil02] is an iterative pattern discovery algorithm. It iteratively grows a pattern trie while maintaining pattern occurrences of each query. Only patterns frequent enough are expanded. It can capture different regular expression tokens: groups (a token that matches multiple tokens in the alphabet) and wildcard positions (a token that matches any subsequence). It can also order the result based on interestingness criteria.

TEIRESIAS [RF98] is a combining algorithm. It starts with a list of elementary patterns that occur at least K times. Then it starts combining these elementary patterns into larger patterns. It uses the observation that a pattern P can be combined from pattern A, B if the suffix of A is the same as the prefix of B. For example, sequences $\alpha\Delta$ and $\Delta\beta$ can be combined into sequence $\alpha\Delta\beta$, where α, Δ, β are sequences.

MobyDick [BLS00] is a combining algorithm. It starts with a dictionary of sequences and then looks for concatenated sequences p, which has a low P-

value, based on the dictionary words, and then adds such p to the dictionary.

SANSPOS [BM11] is a iterative approach that uses a positioning matrix to expand the pattern multiple tokens at a time. The use of positioning matrix can significantly increase the performance in the presence of short tandem repeats.

3.2 Pattern rating

There are many ways of comparing patterns to find the "most interesting" pattern. Of course, pairwise comparison is often wasteful and it is better to have an interestingness measure that we can calculate only by using the query. For practical purposes it is useful to represent that measure with a floating point value.

One useful property that an interestingness measure can have is monotonicity when patterns are ordered by length. This means that when we move from a small pattern to a larger pattern, the interestingness always either grows or decreases. The more common name for this idea is the *Apriori principle*, which states that if a itemset is frequent, then all of its subsets must also be frequent, or if an itemset is infrequent, then all its supersets must also be infrequent. For example, if we make a pattern more specific, the number of matches can only decrease. This can be very helpful for pruning the search space. Mostly monotonic functions could provide probabilistic pruning, but we didn't find any information regarding it.

Example 3.2.1. Lets assume we are looking patterns that should have at least 10 matches. If we encounter a pattern `ACT` that has 8 matches then we do not have to examine patterns `xACT` and `ACTx`, where `x` is some token from the alphabet.

The most trivial measure for queries is the number of pattern matches in the dataset. When calculating the number of matches we must be aware

that we can have multiple matches per sequence. For example, a pattern `AA` matches in sequence `AAAAAA` 5 times. It is better to count the number of sequences that contain the matches to account for such pathological cases.

Many patterns can occur by chance; therefore, it is important to remove such false positives from output is important. As previously mentioned, a frequent pattern is not necessarily interesting. We can use a reference dataset to compare the frequencies of patterns. There are different possibilities for a reference dataset: background sequence, shuffled input sequences, background Markov-Model, binomial/multinomial models. The background sequence is usually a similar dataset to the dataset we are analyzing; for example, if we select a subset from data for analyzing we can use the rest as a background sequence. If there are no background sequences we can simply shuffle the input sequences to get a "randomized" sample. To preserve more of the data characteristics we can build more complex models for randomization, such as hidden Markov Model[Thi+01] or binomial and multinomial models.

We can specify interestingness measures using a reference dataset as a measurement for false positives. For example, one can use the ratio between the input sequence occurrence and background sequence occurrences. One problem with ratios is that, if the frequencies are small, then the ratios may be very high. By using binomial[HAC98] or hypergeometric model we can estimate how probable the number of occurrences is in the input and the background dataset. There are also measures Z-score[S+00], which estimates how many standard deviations an observation differs from the mean, and χ^2-Value[HS99], which estimates whether a frequency distribution differs from theoretical distribution.

If we happen to have several queries with the same "score", then we can break the "tie" by comparing them with additional measures. One such useful measure is the complexity of a pattern. For example, if we have patterns `ATG` and `C` and both have the same number of occurrences then the longer pattern is probably more interesting.

3.3 SPEXS

SPEXS is a pattern discovery algorithm described in "Pattern Discovery from Biosequences"[Vil02]. It was designed to find frequently occuring patterns from sets of sequences. It constructs patterns by incrementally expanding the prefixes of the frequent patterns. It can generate several pattern classes: subsequences, subsequences containing group characters (a token where alternative characters from a list can be used), and patterns containing wildcard positions. The thesis describes several versions of the algorithm for finding specific pattern structures and also provides a general algorithm for pattern discovery. We only show some algorithms from the thesis.

The main idea of the general algorithm 3.1 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 to the output queue for possible output.

The simplest algorithm 3.2 is for finding subsequences from multiple sequences. The other algorithm 3.3 is for finding patterns group symbols from multiple sequences. These algorithms are performant, but they mix the idea of the algorithm and the optimizations of the algorithm, which means that we lose intuitiveness of the algorithms. The algorithms are here mainly for demonstrative purposes and we will not examine these algorithms, because there is a more intuitive approach explained in Chapter 4.

Algorithm 3.1 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
```

Algorithm 3.2 Generation of most "interesting" subsequences of a set of strings

Input: Input strings $S^n = S_1, ..., S_n$, threshold K, interestingness $\mathcal{F}$
Output: Subsequences that occur in at least K sequences of S^n in the order of fitness $\mathcal{F}$

1: $S \leftarrow S_1 \# ... \# S_2$, $\# \notin \Sigma$
2: Generate a mapping $\{1, 2, ..., |S|\} \mapsto \{1, 2, ..., n\}$ for countseq(Set)
3: Root $\leftarrow$ new query
4: Root.label $\leftarrow \epsilon$
5: Root.pos $\leftarrow (1, 2, ..., |S|)$
6: enqueue($\mathcal{Q}$, Root)
7: **while** $N \leftarrow$ dequeue($\mathcal{Q}$) **do**
8: **for** $c \in \Sigma$ **do**
9: $Set(c) \leftarrow 0$
10: **for** $p \in N.pos$ **do**
11: add $p + 1$ to $Set(S[p])$ **unless** $p = |S|$ or $S[p] = \#$
12: **for** $c \in \Sigma$ **where** countseq($Set(c)$) $\geq K$ **do**
13: P $\leftarrow$ new query
14: P.label $\leftarrow c$
15: P.pos $\leftarrow Set(c)$
16: N.child(c) $\leftarrow P$
17: enqueue($\mathcal{Q}$, P)
18: enqueue($\mathcal{B}$, P, $\mathcal{F}$(P.pattern, S^n))
19: delete $N.pos$
20: **while** $(N, f) \leftarrow dequeue(\mathcal{B})$ **do**
21: output $N.pattern$ and f

Algorithm 3.3 Generation of frequent patterns with character group positions

Input: Input strings $S^n = S_1, ..., S_n$, threshold K, character groups Γ
Output: Patterns from $\Sigma \cup \Gamma$ that occur in at least K sequences ofS^n

1: $S \leftarrow S_1\#...\#S_2$, $\# \notin \Sigma$
2: Generate a mapping $\{1, 2, ..., |S|\} \mapsto \{1, 2, ..., n\}$ for $countseq(Set)$
3: Root $\leftarrow$ new query; Root.label $\leftarrow \epsilon$
4: Root.pos $\leftarrow (1, 2, ..., |S|)$
5: enqueue($\mathcal{Q}$, Root)
6: **while** $N \leftarrow$ dequeue($\mathcal{Q}$) **do**
7: Output N.pattern
8: // Construct the position list for pattern defined by node N
9: **if** N.label $\in \Sigma$ **then**
10: Pos $\leftarrow$ N.pos
11: **else**
12: Pos $\leftarrow \bigcup_{c \in \text{N.label}}$ N.sibling(c).pos
13: // Group the positions according to characters in string S
14: **for** $c \in \Sigma$ **do**
15: $Set(c) \leftarrow 0$
16: **for** $p \in$ Pos **do**
17: add $p + 1$ to $Set(S[p])$ **unless** $p = |S|$ or $S[p] = \#$
18: **for** $c \in \Sigma$ **where** countseq($Set(c)$) $\geq K$ **do**
19: N.child(c) $\leftarrow$ new query with label c
20: N.child(c).pos $\leftarrow Set(c)$
21: enqueue($\mathcal{Q}$, N.child(c))
22: **for** $g \in \Gamma$ **do**
23: **if** $\exists f(\Gamma \cup \Sigma, f \subset g, \sum_{c \in f} |Set(c)| = \sum_{c \in g} |Set(c)|$ **then**
24: **continue**
25: **if** countseq($\cup_{c \in g} Set(c)$) $\geq K$ **then**
26: N.child(g) $\leftarrow$ new query with label g
27: **for** $c \in g$ **do**
28: N.child(c) $\leftarrow$ new query with label c and positions $Set(S[c])$
29: enqueue($\mathcal{Q}$, N.child(g))
30: **if** all nodes N.sibling(c), $c \in \Sigma \cup \Gamma$ have been expanded **then**

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 4.1 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)
```

The algorithm starts by initializing the *in* pool. The *in* pool contains queries which we wish to examine further. In the simplest case this means that 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. Extending a query 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, or store the queries on disk, or even discard the queries, if needed. If we wish to get 100 best results the, then the output pool may immediately discard the bad ones.

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

4.3 Filtering

Filtering allows us to reduce the number of queries we have to examine and enables selecting 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 occurrences, metadata in sequences, metadata in dataset or configuration data.

Example 4.3.1. Whether a sequence belongs to the input or the background dataset, can be considered metadata. We count the occurences of a pattern in each origin separately and then, if the ratio between origin dataset counts is larger than some number defined in the configuration, we can add it to output pool.

Although only one filter "function" has been specified in the algorithm the filter can be a composite of multiple filters.

Example 4.3.2. Pattern length greater than three and pattern matches 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 the core of the algorithm and there are several ways of doing it. The main criteria is that the query extending should guarantee that all possible queries get eventually enumerated.

The extender is analogous to an inductive step. Our base case is formulated by the *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 its locations. If our extender generates all the queries where the patterns are larger by 1, then we are guaranteed to enumarate all the patterns.

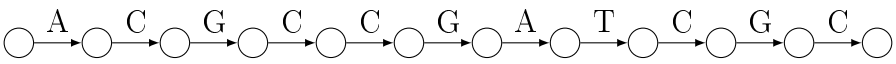
Example 4.4.2. We can start with the empty query and all queries with patterns of length 1. Now if our extender generates queries where the patterns are larger by 2 we can also examine all of the queries.

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 algorithm runtime 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.

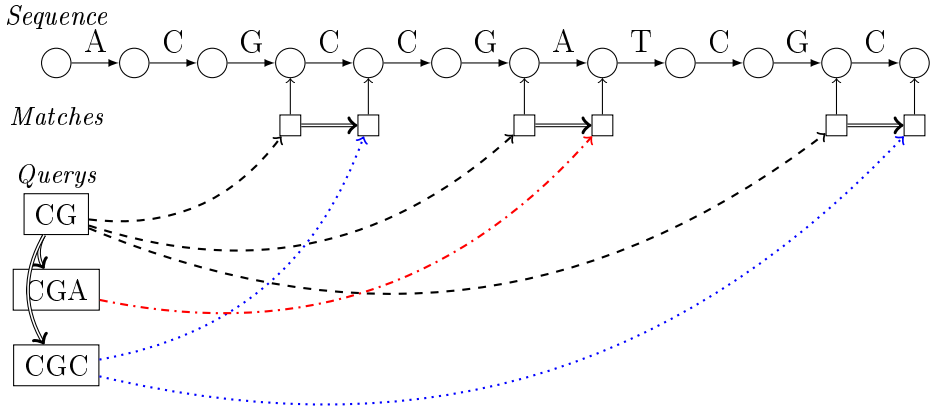
For example the sequence `ACGCCGATCGC` would look like:



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

4.4.1 Sequences

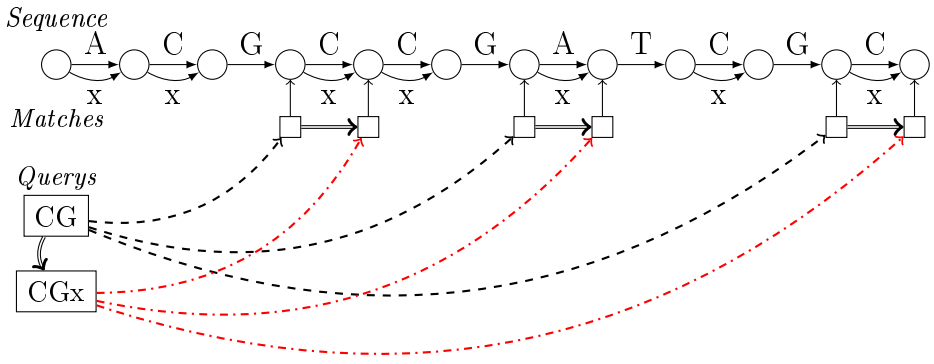
The simplest case how the *next* function behaves is when we are only looking for simple sequences. Let's consider the sequence `ACGCCGATCGC` and the pattern `CG`.



Initially we only have query `CG`. Then, by taking the *next* token from the sequence we can build up queries `CGA` and `CGC`, including the match location set.

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 a edge 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.

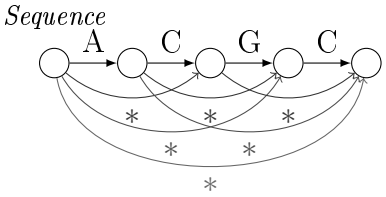
Example 4.4.3. Pattern `T.` would match patterns `TA`, `TC`, `TG`, `TT`.

4.4.3 Star

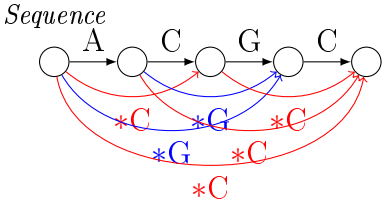
Another possible extension is capturing a run of wildcards.

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

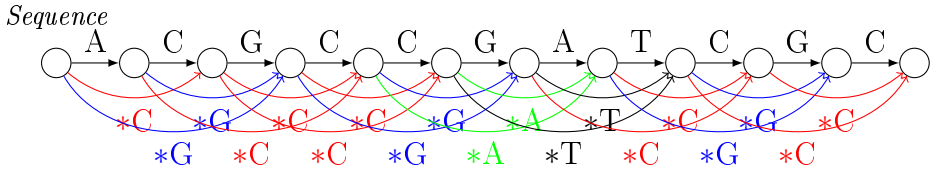
On the graph instead of `.*` we use only `*` symbol.



Constructing more complicated patterns increases the amount of queries that have to be enumerated. 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 that we avoid a 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 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 another extender and combine its results. If we have a group token γ that contains $tokens(\gamma)$ then the match positions for such group is

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

.

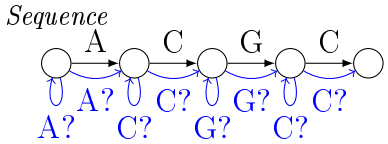
Example 4.4.5. A pattern `A[CTG]` is located in the document D at positions $positions($`AC`$, D) + positions($`AT`$, D) + positions($`AG`$, D)$.

4.4.5 Other possible extensions

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

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

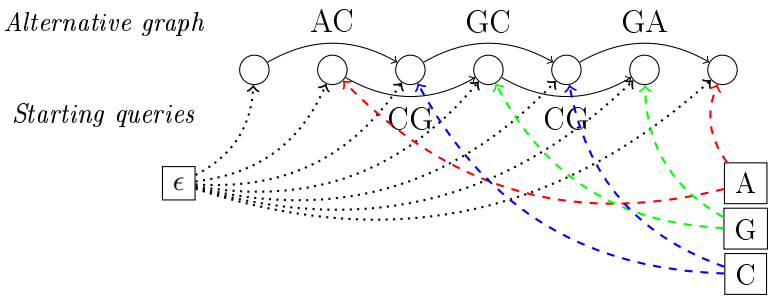
The graph for such token would look like:



Similarly, we can use the same technique that we used for the group tokens, to optimize the performance ($positions(pY, D) = positions(p) \cup positions(pY)$, where Y is a token).

4.4.6 Alternative extensions

We mentioned that we can also extend by a 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 the following setup:



Since at the starting point we have all the patterns of length 0 and 1; then 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 extension need to be adjusted.

4.5 Summary

Although the extender was presented with graphs, practically it is much more reasonable to minimize the graph 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.

We can also derive the minimal requirements for the dataset from the previous results. First, we need to get the initial empty query - which means we should somehow be able to get all the positions, from where a pattern could start. The other requirement is finding the next position and the token

from a given position. Finding the next positions from a given position on sequence can be interpreted as a forward iterator.

The best way to visualize an extender was with graphs, suggesting 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 are more difficult, since we need to remove duplicates caused by extending the previous pattern graphs[Wik13b].

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 initial queries and 4. dataset implementation.

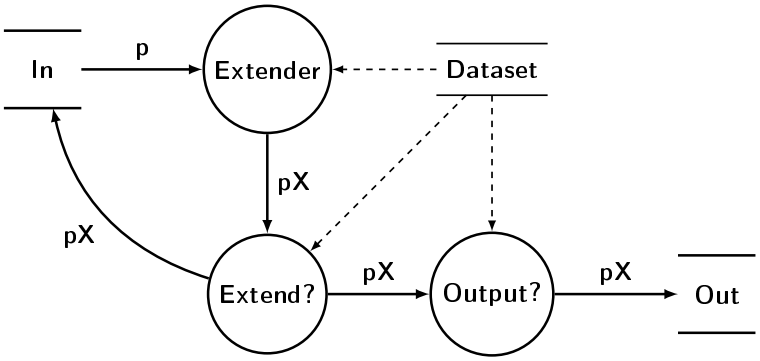
Chapter 5

Parallelization

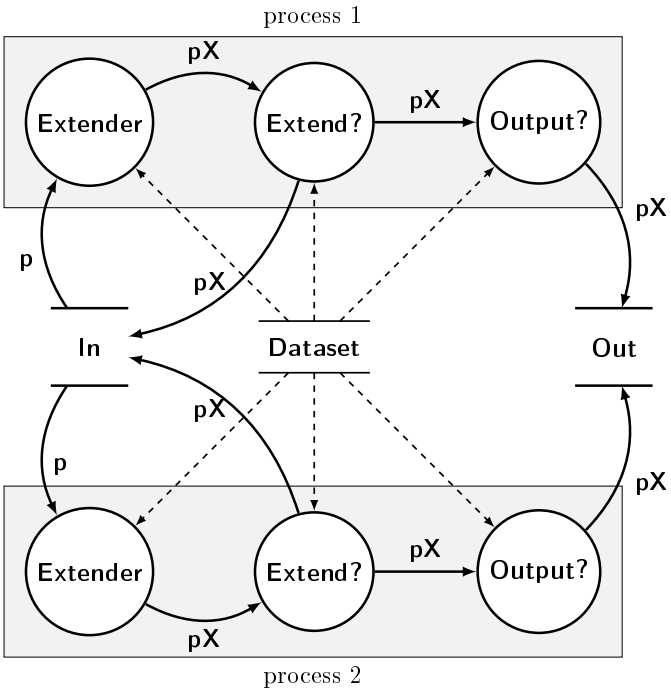
In this chapter we discuss different ways of reifying the algorithm to support parallelism. There are several ways of making a program parallel. Using parallelization means that there is a need for some communication and synchronization to make the processes reach the final result. So, it is useful to find as many possible parallelizations as possible, but it is not wise to use all of them.

5.1 Process

This is the main process of the algorithm as described by data flow diagram.[Kah74; LP95] Circles denote processes and unfinished rectangles denote data stores.



We can see that the different query extension processes do not share a dependency, except the dataset. Dataset itself is read-only in a given process, which means that we can use multiple extender processes. The same applies for extendable and output filter.



We can add more processes in a similar fashion without affecting the end result. However, 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 a previous pattern position, and then grouping those positions together to find the next queries.

The next positions can easily be found from the previous position via

mapping, by using the *next* function of the dataset. Grouping requires some extra attention - grouping is a reduction into a map by a key with joining.

Creating a pseudo-code representation of such function compositions would be very difficult and it 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 5.1 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))))
```

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

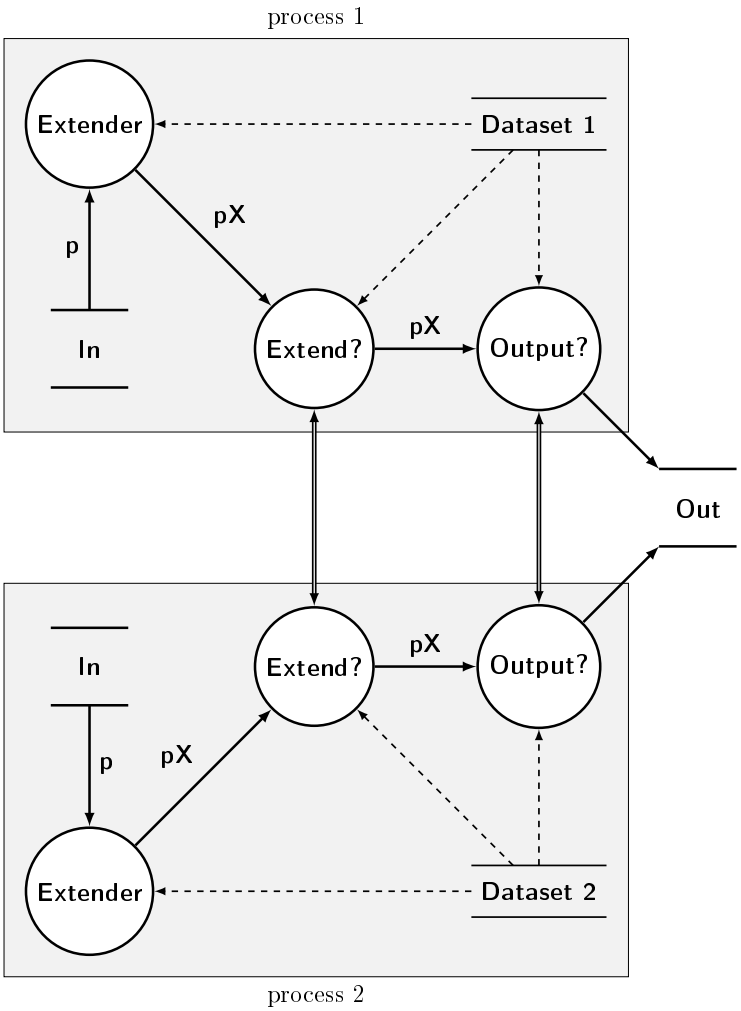
5.3 Distributed processes

Since the dataset and the process memory consumption can get quite large, it would also be beneficial 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 stay inside the sequence, which

means that we can partition the dataset by assigning sequences to separate datasets.

The whole dataset is required only for filtering queries, since simplest operations ("counting matches in dataset") require full knowledge of all matches over the dataset. We can calculate partial results and let the filters communicate the results. This could also be done in a separate process instead of using direct communication.



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 the processes exchange the partial results and add these results together locally. Depending on the local result and filter configuration, whether to discard or keep the query.

Such distribution could be used to separate the process into more manageable chunks, but, at the same time, it may add significant communication overhead.

Chapter 6

Implementation

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. The decision to use Go was based on several informal sources[Mac13; Ful13; Hun11; GPT13; Hic08; Bez+12] and the following discussion should be seen as opinions rather than facts.

This project should be a reference implementation and should be as readable as possible; languages such as C++, Java are probably not the best choices due to their complexity. We need to implement generic code; this means that Clojure, Haskell and OCaml would be useful, but they would require a lot of effort to learn for a newcomer. Languages such as Matlab, Octave or R would be ideal due to their ease of use, but their speed or memory performance is significantly 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. It is easier for new people to contribute to the project. Simplicity means that most of the language semantics can be directly translated to an other language if there are unsolvable problems. In summary, Go seemed a nice compromise between Python and C.

6.2 Architecture

The main criteria for designing program have been described in "On the Criteria To Be Used in Decomposing Systems into Modules"[Par72]. It suggests decomposing programs 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 the configuration file onto an internal data structure, this configuration structure is used as an input to the setup module. The setup module initializes (as defined

by configuration) a reader, a printer and also prepares structures for the algorithm. Then the reader reads the input files into the database. Then the algorithm is started and, finally, the printer formats the results and sends them to the output. This structure is universal for algorithm implementations and it allows adding more configuration options easily, different input formats and different output formats. By changing the configuration, reader and printer we could make it into a web service, instead of running it on command line.

6.3 Configuration

One problem with flexible algorithms is that it can be run in many different ways. This can lead to us having tens or hundreds of command-line flags for the application. To avoid this problem we decided to use a *json*[Cro06] file for the program configuration. This format is widely known and well structured. For example, a configuration file for pattern discovery in protein sequences:

```
{
    "Dataset": {
        "fore" : { "File" : "$inp$" },
        "back" : { "File" : "$ref$" }
    },
    "Reader" : {
        "Method" : "Delimited"
    },
    "Extension": {
        "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?()\t...\tHyper(fore,back)\n"
    }
}
```

In hindsight *json* for configuration may not be the best option due to its rigidity. Users can often forget to add/remove trailing commas or forget to add quotes. This suggests that formats that are less rigid, such as *yaml*[BEI01], would be a better choice.

One 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 this is to add custom syntax into the configuration file:

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

We can interpret command line flags as replacements into the configuration file. I Using

```
spexs2 -conf conf.json argument=other
```

would transform the configuration file into:

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

If no parameter was given then the default value "default" would be used instead.

Configuration files are also problematic when getting familiar with the tool. 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 instead. This means you can use the following for simpler cases:

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

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

```
"Datasets" : {
    "A" : { "File" : "$A$" },
    "B" : { "File" : "$B$" },
    "C" : { "File" : "$C$" },
    ...
```

We can use the command-line argument syntax to make it easier to use. When we are defining filters we need to specify how to exactly calculate them. For example, we can not just say ratio of occurrences since that would be ambigous if there are more than two datasets. Our solution was to use syntax similar to function calls in the configuration. This allows us to clearly see that the occurrences ratio between datasets A and B must be at least 2. Obviously we can define features that take more arguments.

```
...
"Extension": {
    "Outputtable": {
        "OccurencesRatio(A, B)" : {"min" : 2},
        ...
```

6.5 Pools

The input pool can direct the flow of extending process, which, in turn, can affect performance and memory. Performance does not get affected by chaning the poolt type as much because the algorithm is exhaustive and, hence, every query gets examined, unless it is terminated early or the filters are being tuned during runtime.

There are several ways to 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.

One major concern is running on large datasets, which means that the memory consumption is very important. Although examining the most frequent query first can reach the interesting results faster, it also uses more memory due to the unextended less frequent queries.

Examining the least frequent approach minimizes the memory use, since the least frequent query is most likely to be discarded by filters. But this requires the use of a priority queue. The depth first approach uses less memory than travsing the most frequent first and it can use a stack. Since we need concurrent access to the pool it requires fast push/pop operations. The depth-first ordering is more suitable since push/pop operations are faster than on priority queues.

For the output pool a limited size priority queue is the obvious choice, because we need to sort based on some interestingness measure and only the best results are necessary.

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 work concurrently using a simple trie was not an option, because adding a child to the parent from multiple processes can easily cause a race condition.

The first approach we tried was to flip the trie, which means that, 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, because we are working in a garbage collected environment 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 to copy any needed content to the child

and not have links between them.

The other problem was how to store the position set. 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 can be packed with any packing algorithm while they are being stored in a pool and that did have similar memory benefit.

One interesting memory optimization we found was related to storing sparsely distributed integers. Because we did not find a memory efficient set implementation for Go, we needed to implement one ourselves. A trivial way to implement a large bitset is using a hash map, where the values are bit-arrays. We use the first bits as the key and the rest store in the bit-array. We know that the occurrences of a pattern are likely to be sparse, hence it is also quite likely there are only bit arrays where only a single bit is set. In a sense, this sort of sparse data is the worst case scenario for this implementation. The solution is to swap k lower bits with some k higher bits. This means that the key bits will be more likely to collide, hence it is more likely that multiple numbers end up in the same bit-array.

For example, lets assume we have 8bit integers and we can store 3 bit numbers in the bit-array. We swap bits 2 and 1 with bits 4 and 3. We added a "." to show the key-value separation. Here we generated numbers that have a short interval between that is at least larger than 5.

number index	binary 76543.210	swapped 76521.430
7	00001.011	00001.011
12	00011.000	00000.110
19	00100.011	00101.001
26	00110.010	00101.100
32	01000.000	01000.000
37	01001.001	01000.011
43	01010.011	01001.101
50	01100.010	01101.000
56	01110.000	01100.100
61	01111.001	01100.111

In the unswapped case the data structure would have the worst case scenario, where each number would be stored in a separate bit-array. After swapping the bits we have made them "less uniformly distributed" and reduced the number of bit-arrays from 10 to 7. Of course, "the best" bit-swapping method and key/value size vary, depending on the numbers being stored. If we are using large integers we can additionally make an additional layer to reduce the memory overhead. In our case, after tuning, it used about 2x less memory than the trivial implementation.

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 implemented *features* can also be used as a interestingness measure we used a simplification for the "Feature" function definition.

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

This means that the function returns two types, a real value and a string. In the implementation, there are only a few features that return arbitrary

types so it was easier to convert them into a string than let them return a variant type. The only place, where such different features is needed, is for printing. For example, one of such features is the string representation of the pattern.

To construct an *Feature* we use a function that takes the datasets as arguments. To bind the constructor to the configuration we register it and use the function name as the name that is used in the configuration file.

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

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 limiting a feature with a minimum or a maximum value we can turn it into a filter. For example:

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

There are some filters that cannot be defined by features so there is still a possibility to make separate filters. For example, disallowing certain tokens in pattern is defined as a separate filter. In languages which do not support such composition we could use object composition and/or function pointers.

6.8 Synchronized Graph Traversal

spexs2 can be seen as a pattern tree traversal algorithm with some extra logic. Implementing parallel search over a tree requires synchronization such that there are only a certain number of workers and that they wouldn't die of starvation. For universality we describe the principle for graphs.

Without synchronization the parallel version looks like:

Algorithm 6.1 Graph 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. In the algorithm 6.2 we use *mutex* to protect variables and data structures. Semaphore *added* tracks how many items are in the *unvisited* set. If the process terminates *added* is turned into a turnstile[1] on line 30 and 13. Variable *workers* tracks how many workers are busy.

[1]Turnstile[Dow05] is a way of using a semaphore for letting through multiple waiters by first releasing a waiting worker and that worker will release another worker.

Algorithm 6.2 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()                          ▷ wait for an unvisited node
11:             mutex.lock()                          ▷ protect shared variables
12:             if terminate then
13:                 added.signal()                    ▷ let the next worker through
14:                 mutex.unlock()
15:                 break
16:             node ← unvisited.take()
17:             workers ← workers + 1
18:             mutex.unlock()                        ▷ release shared variables
19:             fn(node)                              ▷ do the actual work
20:             for child ∈ children(node) do
21:                 mutex.lock()                      ▷ protect the unvisited queue
22:                 if examine?(child) then
23:                     unvisited.put(child)
24:                     added.signal()
25:                 mutex.unlock()
26:             mutex.lock()                          ▷ protect shared variables
27:             workers ← workers − 1
28:             if workers = 0 and unvisited = {} then
29:                 terminate ← true
30:                 added.signal()                    ▷ release a waiting worker
31:             mutex.unlock()
32:
33:     wait for workers
```

6.9 Debugging

Seeing how the algorithm runs is very useful for getting an understanding how the algorithm works. This often can help to either improve the input configuration or debug the program itself. Such tracing is often implemented by adding debug statements.

For example:

```
func Spexs(s *Setup) {
    for q, ok := s.In.Pop(); ok {
        trace("started extending %v", q)
        extended := s.Extend(q)
        trace("extension result %v", extended)
        for qx := range extended {
            if s.Extendable(qx) {
                trace(" > extendable %v" qx)
                s.In.Push(qx)
                if s.Outputtable(qx) {
                    trace(" > outputtable %v" qx)
                    s.Out.Push(qx)
                }
            }
        }
    }
}
```

Such statements make it harder to read the actual code, also it's hard to modify the statements for debugging or/and provide different ways of debugging. We can use lexical closures to make this simpler:

```
type Extender func(q Query) []Query

func AddDebuggingStatements(s *Setup) {
    fn := s.Extend
    s.Extend := func(q Query) []Query {
        trace("started extending %v", q)
        extended := fn(q)
        trace("extension result %v", extended)

        for qx := range extended {
            trace(" > %v", qx)
            trace(" > extendable %v", s.Extendable(qx))
            trace(" > outputtable %v", s.Outputtable(qx))
        }
        return extended
    }
}

func Spexs(s *Setup) {
    for q, ok := s.In.Pop(); ok {
        extended := s.Extend(q)
        for qx := range extended {
            if s.Extendable(qx) {
                s.In.Push(qx)
                if s.Outputtable(qx) {
                    s.Out.Push(qx)
                }
            }
        }
    }
}

func run(){
    S := CreateSpexsSetup()
    if debugMode {
        AddDebuggingStatements(S)
    }
    Spexs(S)
}
```

As we can see the algorithm doesn't have any debugging statements. We can also define different "debug statement injectors" that provide different information. This method, of course, has a slight impact on performance due to the additional indirection, but only if it is used. This can be extended to provide user interaction and other features.

6.10 Comparison with SPEXS

The implementations of *spexs2* and *spexs* vary significantly. *spexs2* was designed to be a generic tool. In comparison: *spexs* has one input format, allows five ways of extending, allows 4 different ways for pattern traversal and allows specifying about 4 different filters; *spexs2* has two input formats, supports large alphabets[2], has five different ways to extend, has 17 different features and 19 different filters for queries and has customizable output.

It is probably worth mentioning that most of the *query feature* implementations are about 10 lines of code, every *query feature* that returns a floating point value can be used as a *filter*, and each extension method is about 30 lines of code. In summary *spexs2* can be considered a generic tool for sequence pattern discovery, where new features can be added easily.

[2] *spexs2* alphabet size is limited by the size of an integer.

Chapter 7

Applications and experimental results

7.1 Examples

Here we show several proof of concept examples for using the *spexs2* tool.

7.1.1 Genomic sequences

The original SPEXS algorithm worked on genomic sequences, so it is appropriate to show that *spexs2* also works on such datasets. Here is a problem that was presented in "Pattern Discovery in From Biosequences"[Vil02]. The search problem was to find overrepresented sequences from a cluster of co-expressed genes by comparing them to a set of random samples from upstream sequences in yeast genome.

Pattern	Cluster	Background	Ratio	Binomial Prob.
GATGAG.T	52/70	63/67	8.37	4.649e-37
G.GATGAG.T	39/49	26/29	14.979	6.926e-37
AAAATTTT	63/77	126/134	5.095	1.731e-33
AAAA.TTTT	59/86	107/138	5.617	3.952e-33
GATGAG.TG	34/42	23/23	14.74	7.414e-32
G.GATGAG	45/60	54/58	8.456	1.049e-31
AAA.TTTT	79/145	247/345	3.261	1.281e-31
GATGAG.T.A	35/44	28/29	12.551	2.690e-30
TG.AAA.TTT	53/61	93/99	5.808	8.624e-30

Almost the same results were reported by *spexs*; there are some variations due to the random background sample.

7.1.2 Event sequences

To test, whether it is plausible to analyze event sequences, we generated a dataset of 5000 sequences of length 20 to 50. The events [`A`, `B`, `C`, `D`, `F`, `G`, `H`, `I`, `X`, `Y`, `Z`] in the dataset are non-uniformly distributed and additionally there is an error event `E`, which will happen if there is a "trigger pattern" `X.*Y.*Z`. Some examples from the sequences:

```
# without errors
AIBBFCACCADAHABXCHCG
GBACDBHBDAIBHYDIHAAADAFAHFGGDBFFYFZBFBAGDIDDX
CAGZHGBAXHFIGBAFBIABDYBABBFDBAFGGAAAAHHC
CGDCHHAAAABFBDBCHBBFGICDBGGDGCDFIFADCA
# with errors
ADDDBBCYDFCCHXFDDXBAYDYBHACAZE
DXFDIHBXYDBFGGCBHAYBDHZE
IXBBXHBBACYCFHADHGFDACDHCGYABYBHADZE
AHAFFFGABIXBCAYCBBHBDCDDXZE
```

It is easy to notice the `ZE` part, but the `X.*Y.*` part of the pattern is very hard to notice - even when you know that it is there.

To prepare the dataset we extracted sequences with errors into a separate file. Since there can be a lot of patterns by chance we use the whole dataset as the background. This allows us to compare the count of matches in the

errors and the whole dataset. If the pattern occurs in both datasets at the same frequency it probably isn't an interesting pattern.

pattern	errors	all	ratio	p-value
A.*Z	330/343	1706/5000	2.818	4.952e-126
B.*Z	329/343	1676/5000	2.860	1.041e-126
X.*G.*Z	254/343	654/5000	5.659	5.511e-132
Y.*B.*Z	256/343	596/5000	6.258	1.954e-142
X.*Y	343/343	1771/5000	2.822	9.879e-147
X.*F.*Z	274/343	717/5000	5.568	2.409e-147
X.*C.*Z	285/343	758/5000	5.478	7.304e-156
X.*D.*Z	281/343	718/5000	5.701	4.570e-156
A.*A.*Y.*Z	250/343	452/5000	8.055	1.304e-158
G.*Y.*Z	265/343	515/5000	7.494	1.656e-166
F.*Y.*Z	272/343	522/5000	7.588	1.639e-174
X.*A.*Z	305/343	811/5000	5.478	1.798e-176
X.*B.*Z	304/343	785/5000	5.641	1.540e-178
D.*Y.*Z	281/343	557/5000	7.347	1.066e-180
C.*Y.*Z	285/343	584/5000	7.107	1.056e-181
B.*Y.*Z	292/343	610/5000	6.971	2.370e-187
A.*Y.*Z	300/343	616/5000	7.092	3.163e-198
X.*Z	343/343	1054/5000	4.740	1.276e-215
Y.*Z	343/343	805/5000	6.204	1.009e-249
X.*Y,*Z	343/343	343/5000	14.537	0

As we can see, this `X.*Y.*Z` pattern can be found very easily. We picked 20 best patterns based on p-value, which was calculated according to hypergeometric distribution. This example is an ideal situation and the conclusions should be verified against real world data.

7.1.3 Text patterns

As an example, how text patterns can provide useful feedback, we ran it on a chapter of this thesis. Potentially it can find overused words and phrases. To test this claim we ran the algorithm on Chapter 6.

To prepare the text we separated each sentence to a separate sequence. Then we removed all the non-textual characters and replaced them with spaces. The text was then converted to lowercase. We additionally tried stemming the text, but it didn't provide any useful improvements for this

case. We use a group "*bind*" $= [and, or, if, then, else, the, a, an, my]$ to define words that do not carry much meaning. First we searched patterns that can have group symbols and are at least 4 tokens long. We limited the output to 10 results:

```
matches  pattern
2        this means that the
2        a practical implementation spexs2 for
2        a practical implementation spexs2
2        discuss a practical implementation spexs2
2        discuss a practical implementation
2        discuss a practical implementation spexs2 for
2        discuss (bind) practical implementation spexs2 for
2        discuss (bind) practical implementation
2        discuss (bind) practical implementation spexs2
2        for pattern discovery in
```

Repetition of such long word sequences looks peculiar. Further investigation revealed that there was a sentence that was rewritten and the previous version hadn't been removed. After removing the repeating sentence we reran and also lowered the pattern length limit to 3.

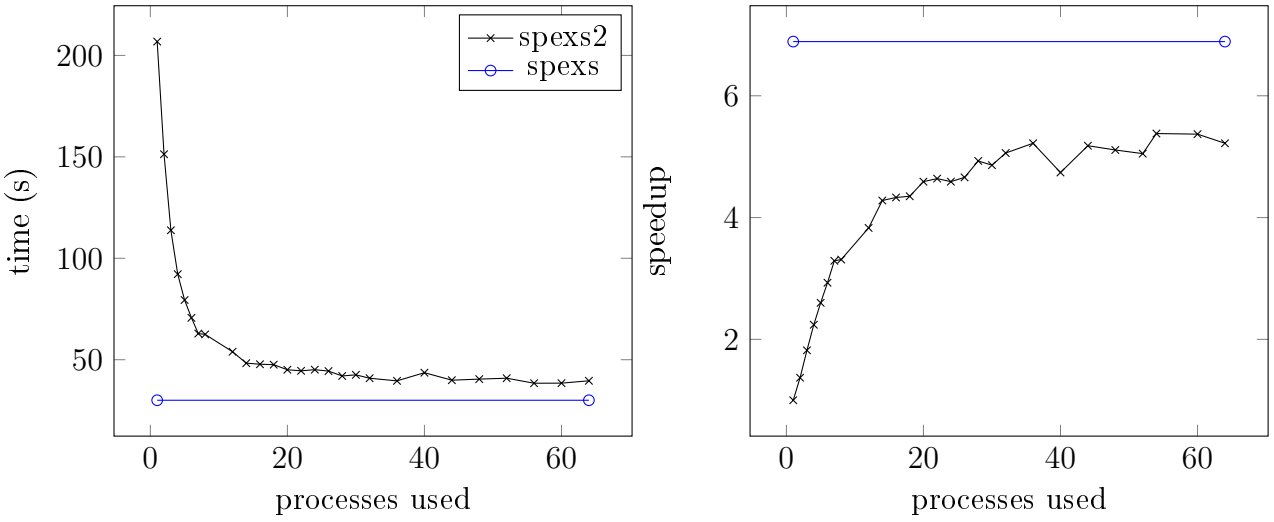
```
matches  pattern
5        the configuration file
4        a lot of
3        this means that
3        in the configuration
3        means that the
3        there are only
3        in (bind) configuration
3        pattern discovery in
3        we can use
3        of (bind) pattern
2        into (bind) configuration file
2        in (bind) configuration file
2        pattern discovery in sequences
2        into the configuration file
2        for pattern discovery in
2        this means that the
2        in the configuration file
2        be the best
2        go is a
2        the algorithm is
```

We see repetitions such as "this means that", "in the configuration", "we can use" and "pattern discover in", which suggests we can improve the text at those places. Usefulness of the algorithm for such natural language processing tasks should be further examined.

7.2 Performance measurements

To verify that the parallelization improves the performance we need to see a speedup when using multiple cores. As a comparison we also compare with the original *spexs*. The exact versions we tested were *spexs2@6b14edd* and *spexs.0.2.a01*.

For performance analysis we used 400,000 random protein sequences with length 12. Limited number of wildcards to 3 and searched all patterns with at least 5 matches in the dataset. We calculated the average of 5 runs for *spexs* and average of 3 runs for *spexs2*.



(a) (b)

The original *spexs* works faster than *spexs2* which is to be expected due to the runtime differences. At single core *spexs2* performance is about 7 times

slower than *spexs*. We can expect 2x performance difference due to the Go runtime and compiler. The rest of the difference 3.5x are most likely due to more general algorithm and some simplifications that were made to increase code readability.

The parallelization has significant benefit up to 20 cores. Amdahl's Law [Gus88] states that the the parallel algorithms will be limited by their sequential parts; so this falloff is to be expected. The parallelization is effective, although the runtime has significant impact on the performance. *spexs2* performance can be significantly improved with optimizations[1] that weren't implemented due to time constraints.

[1]The focus currently has mainly been simplicity, readability and memory usage.

Chapter 8

Conclusions

In this thesis we analyzed how to develop a parallel pattern discovery algorithm. We showed how we can take an already existing 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 ways of interpreting it. Decomposing the algorithm allows us to talk about separate parts of the 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 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 analyzed the performance gained from parallelization.

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

Paralleelne Mustriotsing

Egon Elbre

Magistritöö

Selles töös uurisime, kuidas arendada paralleelset mustrituvastus algoritmi. Näitasime, kuidas võtta olemasolev algoritm ning paralleliseerida see üldistades, liigendades ja reifitseerides. Algoritmi üldistamine võib tuua esile intuitiivse algoritmi interpretatsiooni. Liigendatud algoritmis on võimalik iga osa eraldi käsitleda algoritmi tulemust muutmata. Asendades iga osa paralleelsete struktuuride ja algoritmidega, saamegi paralleelse algoritmi.

Praktilise osana implementeerisime paralleliseeritud algoritmi *spexs2* võttes aluseks algoritmi SPEXS[Vil02]. Seejärel arutlesime erinevate probleemide üle, mis tekkisid algoritmi implementeerimisel. *spexs2*-te on võimalik laiendada erinevate sisendandmetega, otsingufiltritega ja huvitavuskriteeriumitega. Pakkusime välja erinevaid algoritmi kasutusvõimalusi ning analüüsisime paralleliseerimise tulemusel saavutatud kiirusevõitu.

Selles töös tutvustatud ideid saab kasutada algoritmide üldistamisel ja paralleliseerimisel. Algoritmi üldistamisel on võimalik leida uusi viise kuidas algoritmi optimeerida ning avastada uusi algoritme ja leida uusi kasutusvaldkondi sellele algoritmile. Üldistuste implementeerimisel saame programmi, mida on lihtne laiendada ning mida saab kasutada erinevate probleemide lahendamiseks.

Bibliography

[BEI01] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. "YAML Ain't Markup Language (YAML™) Version 1.1". In: *Working Draft 2008-05* 11 (2001).

[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.

[BM11] M Baena-Garcia and Rafael Morales-Bueno. "New data structures for analyzing frequent factors in strings". In: *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*. IEEE. 2011, pp. 900–905.

[Cro06] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. RFC 4627 (Informational). Internet Engineering Task Force, July 2006. URL: http://www.ietf.org/rfc/rfc4627.txt.

[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.

[Dow05] A.B. Downey. *The Little Book of Semaphores*. 2005. URL: http://books.google.ee/books?id=pojKZwEACAAJ.

[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/.

[Gus88] John L Gustafson. "Reevaluating Amdahl's law". In: *Communications of the ACM* 31.5 (1988), pp. 532–533.

[HAC98] Jacques van Helden, Bruno André, and Julio Collado-Vides. "Extracting regulatory sites from the upstream region of yeast genes by computational analysis of oligonucleotide frequencies". In: *Journal of molecular biology* 281.5 (1998), pp. 827–842.

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

[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.

[How+08] Doug Howe et al. "Big data: The future of biocuration". In: *Nature* 455.7209 (2008), pp. 47–50.

[HS99] Gerald Z Hertz and Gary D. Stormo. "Identifying DNA and protein patterns with statistically significant alignments of multiple sequences." In: *Bioinformatics* 15.7 (1999), pp. 563–577.

[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.

[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.

[S+00] Saurabh Sinha, Martin Tompa, et al. "A statistical method for finding transcription factor binding sites". In: *Proc Int Conf Intell Syst Mol Biol*. Vol. 8. 1553-0833. 2000, pp. 344–54.

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

[Thi+01] Gert Thijs et al. "A higher-order background model improves the detection of promoter regulatory elements by Gibbs sampling". In: *Bioinformatics* 17.12 (2001), pp. 1113–1122.

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

[WDD99] Andreas Wespi, Marc Dacier, and Hervé Debar. *An Intrusion-Detection System Based on the Teiresias Pattern Discovery Algorithm*. 1999.

[Wik13a] Wikipedia. *Regular expression — Wikipedia, The Free Encyclopedia*. [Online; accessed 12-May-2013]. 2013. URL: `http://en.wikipedia.org/wiki/Regular_expression`.

[Wik13b] Wikipedia. *Subgraph isomorphism problem — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-May-2013]. 2013. URL: `http://en.wikipedia.org/wiki/Subgraph_isomorphism_problem`.

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

Concise Implementation

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

Algorithm B.1 Definitions

```
(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))
```

Algorithm B.2 Extenders

```
; simple extension function
(defn walk-extend [dataset positions]
  (let [steps (mapcat #(walk dataset %) positions)]
   (group-map-by :token :position steps)))

; group extender
(defn- select-merged [m ks]
  (mapcat second (select-keys m ks)))

(defn extend-grouper [extend groups]
  (fn [dataset positions]
    (let [ extended (extend dataset)
           groupings (for [[token items] groups]
                       [token (select-merged extended items)])]
      (apply merge extended groupings))))

; function to combine multiple extension functions
(defn combine-extenders [extenders]
  (fn [dataset positions]
    (apply merge-with concat (map #(% dataset positions) extenders))))
```

Algorithm B.3 The main algorithm

```
; finally the algorithm itself:
(defn- spexs-step [ds q extend]
  (map #(child-query q %) (extend ds (:positions q))))

(defn spexs [{
    ds  :dataset ; dataset
    in  :in      ; input coll
    out :out     ; output coll
    extend  :extend  ; position extender function
    extend? :extend? ; query filter for further extension
    output? :output? ; query filter for output
  }]
  (let [e (empty-query ds)]
    (loop [in (conj in e)
           out out]
      (if-not (empty? in)
        (let [[q & qs] in
              querys (spexs-step ds q extend)
              new-in  (concat qs  (filter extend? querys))
              new-out (concat out (filter output? new-in))]
          (recur new-in new-out))
        out))))
```

Algorithm B.4 Sequence Dataset

```
; here is an example how to implement a dataset
(defn- posify [row-index row-item]
  (map (fn [pos] [row-index pos]) (range (count row-item))))

(defrecord SequenceDataset [items]
  (token [this [row pos]]
       (nth (nth (:items this) row) pos))

  Dataset ; satisfy dataset interface
  (all   [this]
         (mapcat posify (range) (:items this)))
  (walk  [this [row i]]
         (let [row-item (nth (:items this) row)]
           (if (> (count row-item) i)
             [(Step. (token this [row i]) [row (inc i)])]
             []))))
```

Algorithm B.5 Example how to use

```
(def simple-dataset (SequenceDataset. ["ACGT" "CGATA" "AGCTTCGA" "GCGTAA"]))

(spexs { :dataset simple-dataset :input [] :output []
         :extend walk-extend
         :extend? #(> (count (:positions %)) 3)
         :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 20, 2013