University of Tartu
Faculty of Mathematics and Computer Science
Institute of Computer Science

Egon Elbre

Parallel Pattern Discovery

Master's Thesis

Supervisor: J. Vilo, PhD

TARTU 2012

Contents

Chapter 1

Introduction

1.1 Motivation and background

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

One of main drivers of such large datasets is analysis of genomic and proteomic sequences. Regularities in such data can give new insights into how these patterns form and how they are related to the other features of the data.

More about why it's important...

In this thesis we explore an algorithm for finding patterns and show how abstractions can make it scalable and flexible, and simpler both in theory and implementation compared with non-abstract version.

1.2 Pattern Discovery

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.

Since the discovery algorithms are highly dependent on the data structures, that are being searched, the algorithm must be minimal in the requirements on the dataset to be applicable as wide range as possible. This also means that the patterns found must be dependent on the initial data.

1.3 Structure of the thesis

First we introduce definitions of our data and patterns in Chapter 2. In Chapter 3 we describe the general SPEXS algorithm as specified in Vilo et al. In Chapter 4 we generalize and abstrify the algorithm to get a more flexible and parallel algorithm. We will describe the implementation consideration of flexible algorithms in Chapter 5 using the abstract SPEXS algorithm as an example. Chapter 6 uses the implementation to show how it can be applied.

Chapter 2

Definitions

Pattern discovery is a problem of finding interesting patterns in some dataset. We discuss the algorithm in terms of DFAs and show how this can help to discover patterns in sequences.

2.0.1 Sequence

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 $S = a_1 a_2 ... a_n, \forall i \in \Sigma$ is called a *sequence over the token set* Σ*. If the length of the string is* 0*, it is called an empty string or* ϵ*.*

2.0.2 Pattern

A pattern is a structure that defines a set of structures Γ. We denote the set that a pattern defines as $x(\Gamma)$. If a structure α *matches a pattern* Γ*, it means that* $\alpha \in x(\Gamma)$*.*

The most common form of such structures are regular expressions.

2.0.3 Finite automaton

A deterministic finite automaton (henceforth *DFA) is an finite state machine (henceforth* FSM) that accepts or rejects a sequence of tokens.

A *non-deterministic finite automaton (henceforth* NFA) is an FSM that accepts or rejects a sequence of tokens.

Patterns can be represented

Both patterns and sequences can be represented by NFAs and hence is a good abstraction for both. This also means that if our algorithm works on NFAs it must work on any other dataset that is defined in terms of NFA.

2.0.4 Dataset

A *dataset is a set of NFAs. In practice it may be more comfortable to view it as a set of sequence because of it's simpler structure.*

2.0.5 Query

A query is a compound structure that has information about a pattern and it's matches in a dataset. The information about a match is the ending states where a pattern matches a DFA.

On sequences this means the ending positions of the pattern in the sequence.

2.0.6 Query interestingness

Query feature is a function of the query. It gives information about the query such as the pattern representation, length, number of matches in the dataset. Query interestingness is a function of the query whose results are well-ordered. This functions result allows to say whether one query is more interesting than the other. For convenience it is useful to represent that value in R.

2.0.7 Pattern discovery problem

Pattern discovery problem is a process of finding most interesting querys by some interestingness measure.

Chapter 3

SPEXS

SPEXS is an pattern discovery algorithm described by Vilo et al. 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.

3.1 Algorithm

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

```
TODO: format nicely
Algorithm 3.19 The SPEXS algorithm
Input: StringS, pattern class P, output criteria, search order, and fitness measur
Output: Patterns \pi from pattern class P fulfilling all the criteria, and output
of fitnessF
Method:
1. Convert input sequences into a single sequence, initiate the data structures
2.Root  new node
3.Root:label = \eps
4.Root:pos  (1; 2; : : : ; n) // Assume empty pattern to match everywhere
5.enqueue(Q; Root; order)
6. whileN  dequeue(Q)
```

```
7. Create all possible extensionsP 2 P ofN usingN :pos andS
8. foreach extensionP ofN
9. if patternP and position listP:pos fulfill the criteria
10. then
11.N :child  P
12. calculateF(P; S)
13.enqueue(Q; P; order) // Insert toQ for further extensions
14. ifP fulfills the output criteria storeP into output queueO
15. end
16. Report the list of top-ranking patterns from output queueO
```

The main idea of the algorithm is that first we generate a pattern and a query that match all possible positions in the sequence. We then put this query into a queue for extending.

Extending a query means finding all querys whose patterns length is longer by 1. If any of the querys is fit by some criteria it will be put into the main queue, for further extension, and output queue for possible result.

Chapter 4

Abstractification

In this chapter we discuss the general structure of the new algorithm.

4.1 Algorithm

The algorithm in a more convetional view is:

```
func SPEXS(dataset, in, out, extender, extendable, outputtable, postprocess) {
q := NewEmptyQuery(dataset)
in.Put(q)
while( !in.Empty() ) {
q := in.Take()
extended := extender(q)
foreach qx in extended {
if extendable(qx)
in.Put(qx)
if outputtable(qx)
out.Put(qx)
}
}
}
```

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

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

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

If the query is fit for output we 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 the all patterns that conform to our criteria.

4.2 Pools

Pool is an abstract datatype for a collection of querys. The pool allows querys to be put into it and taken from it, also we can ask whether the pool is empty or not.

It has no guarantees on how the querys are stored internally and in which order they are taken out.

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

4.3 Filtering

Filtering allows us to dramatically reduce the number of querys we have to look at. It also allows to select only interesting patterns by some criteria.

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

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

4.4 Extending

The extending process is at the core of the algorithm. We shall look at how we can deal with different types.

The extending method is:

```
func extender(query) {
    // find all next positions
    nexts = new collection
    foreach pos in query.Matches {
        token, set of nexts = next(pos)
        nexts.add({token, set of nexts})
    }

    // group positions by the token
    matches = new map token => pos
    foreach x in nexts {
        matches[x.token].add(x.pos)
    }

    // make new querys from the matches
    result := new query collection
    foreach token, positions in matches {
        q := NewQuery( query.Pattern + token, positions )
        result.add(q)
    }

    return result
}
```

4.4.1 Sequences

The simplest of the extensions are just sequences.

Let's consider a sequence ACGCCGATCGC and a pattern CG.
Diagram of the ACG.CCG.ATCG.C and query for that pattern.
Next positions step is finding ACG.[C]CG.[A]TCG.[C].
Grouping is [C] ==> CGC, [A] => CGA.

4.4.2 DFAs

Since we want to find more interesting patterns we can add more information to the sequence DFA. Such as a star expression.
Same sequence with star paths.

4.4.3 Groups

Although we can add the group information to the DFA, it is more performant to use the information gathered from the extension of non-groups.
A[CG].Positions = AC.Positions union AG.Positions

4.4.4 Other

There maybe several other extensions to the regular expressions.
Questionable position:
AC?.Positions = A.Positions + AC.Positions

Chapter 5

Parallelization

Here we describe full parallelization of the previous algorithm. The fully parallelized code can be seen in the appendix.

5.1 Process

The main process of the algorithm as described by dataflow:

```
     q1   => |filter|
IN POOL == take q ==> |EXTENDER| <   q2   => |filter| > => OUT POOL
|  q3   => |filter|
|       v
|< =======================      |filter|
```

The easiest thing here to parallelize is the extender since it's interaction can be seen as a separate unit.

5.2 Horizontal scaling

If instead of in/out pool we had several the algorithm can still work, if we have single take / put process to decide which actual pool to use.

We can use several extenders that work in parallel without problems since they do not need information about other querys nor input/output pools.

DIAGRAM

5.3 Filtering

This is the only place where we may need the whole information about the query.

Although most operations can be parallelized with map reduce.

5.4 Extending

We still can also do the extending process in parallel:

```
func extender(query) {
// find all next positions
nexts = (map next query.matches)
matches = (group nexts by token)
result = (map newquery matches)
other = (map #(union (matches %1)) groups)
return result union other
}
```

5.5 Dataset partitioning

If we look at where we need synchronization points if we partition the dataset. Whole parts of extension still apply for parts of dataset.

Synchronization is only needed for filtering. We don't need to know the whole query but just information about parts to make a decision whether to filter or not.

Chapter 6

Implementation

The actual implementation may need to diverge from the abstract defintion of the algorithm for practicality, simplicity or performance reasons.

Many of the operations can be optimized for some particular subset of datasets and features.

In this part we will use actual code from the program to show the implementation. For an introduction to the language Go that has been used see the appendix.

6.1 Algorithm

The algorithm was implemented as:

```
type Setup struct {
Db  *Database
Out Pooler
In  Pooler

Extender Extender

Extendable  Filter
Outputtable Filter
PostProcess PostProcess
}

func Run(s *Setup) {
prepareSpexs(s)
for {
p, valid := s.In.Take()
if !valid {
return
}

extensions := s.Extender(p)
for _, extended := range extensions {
if s.Extendable(extended) {
s.In.Put(extended)
}
if s.Outputtable(extended) {
s.Out.Put(extended)
}
}

if s.PostProcess(p) != nil {
break
}
}
}
```

Setup is a structure designed to hold the full setup of the algorithm. The algorithm is very similar to the algorithm described in the theoretical part, with the slight addition of PostProcess.

6.2 Architecture

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

We chose the following decomposition:

```
Configuration
Dataset reader
Setup
Algorithm
Sets
Pools
Extenders
Features
Filters
Printer
```

6.2.1 Configuration

One problem with flexible algorithms is that they a lot of ways to be run. This often would need having tens or hundereds of program flags. To avoid this problem we decided to use a json file for the program configuration.

Example Configuration.

The problem with only using a json file is that when running from command line it may be more comfortable using flags. To solve this problem we added replacement strings into the json files that can be given in as a program argument.

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

When using spexs2 –conf conf.json input=filename the input will be replaced by filename. Also there is optional default value if one wasn't given.

6.2.2 Dataset reader

One of the most likely things to change is the input format. As an example we can have different delimiters. By adding this separation immediately we can avoid the future problem of trying to support multiple formats.

6.2.3 Datasets

We also may need to process several files. This means that we should treat each file as a separate entity and allow calculate features only on part of the dataset.

6.2.4 Setup

There are a lot of ways to setup the program. This allows us to separate the initialization all of the structures necessary for the algorithm from the algorithm itself.

6.2.5 Algorithm

This is an obvious separation. This is the main driver of the algorithm and shouldn't know anything about the rest of the setup. Also there are two versions of the algorithms one parallel the other not.

6.2.6 Sets

Since we need a collection how to store the matching positions it suggests the need for a set datatype.

6.2.7 Pools

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

For the pools we have several choices: lifo, fifo or priority.

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

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

A priority queue suits for the out pool since we can easily then use some feature to sort the queue and choose only a limited amount.

6.2.8 Extenders

As we saw in the theoretical part there are a lot of ways to extend the query. Hence it is reasonable to decompose into a separate piece from the algorithm.

6.2.9 Features

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

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

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

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

Example:

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

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

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

6.2.10 Filters

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

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

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

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

6.2.11 Printer

Although we use a general formatting there may be a need to save into several files or output some extended information of the query.

6.3 Debugging

Debugging is a important part of development process hence the need for more information how the algorithm is working.

Often this is resolved by adding some debug statments:

```
for i := 0; i < 100; i += 1{
printf("picking")
a := pick()
printf("picked %v", a)

printf("picking")
b := pick()
printf("picked %v", b)

out.put(a + b)
}
```

This can be harmful to the readability of the code. To fix this debugging problem we use closures.

```
type PickFunc func() Thing
func debuggable(fn PickFunc) PickFunc {
return func() Thing {
printf("start")
p := fn()
printf("picked %v" p)
return p
}
}

pick = debuggable(pick)
for i := 0; i < 100; i += 1{
a := pick()
b := pick()
out.put(a + b)
}
```

As we can see the algorithm implementation is much more readable.

Chapter 7

Applications and experimental results

Here we show example uses for the program:

7.1 DNA sequences

Make an example.

7.2 Protein sequences

Make an example.

7.3 Text mining

Make an example.

Chapter 8

Conclusions

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

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

We also demonstrated how to make the algorithm more concrete and work well on some particular datasets. This also took into consideration further development and flexibility of the algorithm.

Bibliography

[1] J. Vilo, "Pattern discovery from biosequences," 2002.

Appendix A

spexs2 command line utility

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

A.1 Configuration

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

A.2 Running

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo conse-

quat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Appendix B

SPEXS Source Code

The source code for the implementation of SPEXS is available at `github.com/egonelbre/spexs`.

The source folder "src" has the following structure:

Appendix C

Go language

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.