University of Tartu

Faculty of Mathematics and Computer Science

Institute of Computer Science

Egon Elbre

Parallel Pattern Discovery

Master's Thesis Draft

Supervisor: J. Vilo, PhD

TARTU 2013

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.

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

1.2 Structure of the thesis

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

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.

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.

2.1 Sequence and Dataset

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

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

For example $ACGTGCCATC$ is a sequence where $\Sigma = \{A, C, G, T\}$.

A *dataset* is a set of sequences. For example a document can be considered as a dataset, where the sentences are sequences and each word is a token in the alphabet.

2.2 Pattern

A *pattern* is a structure that defines a set of sequences. Pattern is usually denoted by a sequence over an extended alphabet.

We denote the set of sequences that a pattern p defines as $all(p)$. If $\alpha \in all(p)$, where α is a sequence then we say that sequence α *matches exactly* pattern p. We say that α *matches* p if any of sequences $all(p)$ is a subsequence of α.

The most commonly used pattern descriptions are regular expressions. For example $.[AT]$ denotes a set $\{AA, AT, CA, CT, GA, GT, TA, TT\}$ and this would match $CCTC$ and exactly match AT. A *match* is a location set where the pattern in the sequence ends.

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

2.3 Query

A *query* is a compound structure $< D, p, matches(p, D) >$, where D is the dataset and p a pattern.

2.3.1 Query features

Query feature is a function $f : Q \mapsto A$, where Q is the query type. It gives information about the query such as the pattern representation, length, number of matches in the dataset.

Query interestingness is a function $f : Q \mapsto Ord$ 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 $\Re$.

Query filter is a function $f : Q \mapsto Bool$ of the query whose result is a boolean.

2.4 Pattern Discovery

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

For example "Finding most common nucleotide patterns that are at least 3 nucleotides long from a shotgun sequencing output." *Most common* defines our interestingness measure. *At least 3 nucleotides* is the pattern subset criteria. *Sequencing output* is our dataset and *nucleotides* define the token alphabet.

Chapter 3

Algorithms

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

3.1 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.

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

Algorithm 1 The SPEXS algorithm

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

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

3.2 TEIRESIAS

write about it

Chapter 4

Generalization

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

4.1 Algorithm

The algorithm in a more conventional view is:

Algorithm 2 The spexs2 algorithm

```
Input: $dataset$, $in$ pool, $out$ pool, $extender$, $extendable$ filter, $outputtable$ filter, $postprocess$ function
Output: Patterns satisfying filters and $extender$ are in $out$ pool
1: $\varepsilon \leftarrow NewEmptyQuery(dataset)$
2: $in.put(\varepsilon)$
3: while $q \leftarrow in.take()$ do
4:     $extended \leftarrow extender(q, dataset)$
5:     for $qx \in extended$ do
6:         if $extendable(qx)$ then
7:             $in.put(qx)$
8:         if $outputtable(qx)$ then
9:             $out.put(qx)$
10:    $postprocess(q)$
```

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

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

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

If the query is 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 all the patterns that conform to our criteria.

4.2 Pools

Pool is an abstract datatype for a collection of queries. The pool allows queries 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 queries are stored internally and in which order they are taken out. This gives an option to persist the queries if needed.

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

4.3 Filtering

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

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

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

4.4 Extending

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

The idea of extending method is:

Algorithm 3 SPEXS2 extender

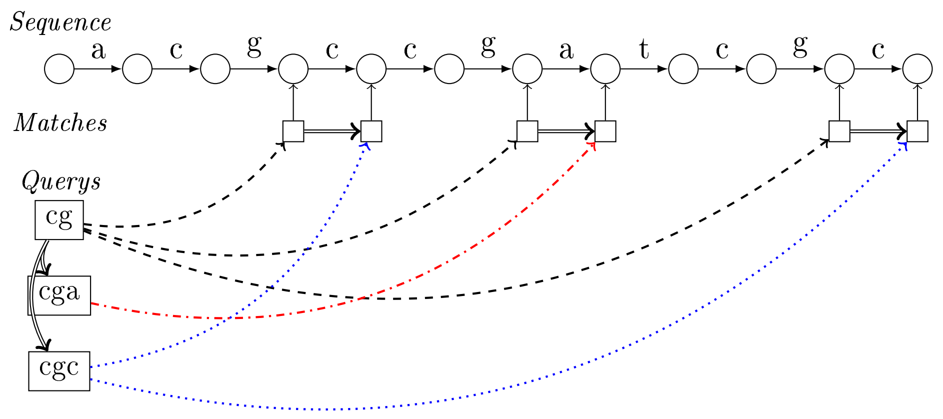
```
Input: $q$ query, $next$ function
Output: result contains queries that have been extended by one
 1: $nexts \leftarrow new\ collection$
 2: for $pos \in positions(q.matches)$ do
 3:     $(token, loc) \leftarrow next(q.document, pos)$
 4:     $nexts.put((token, pos))$
 5: $matches \leftarrow new\ map\ of\ token\ to\ position\ set$
 6: for $(token, loc) \in nexts$ do
 7:     if $matches[token]$ doesn't exist then
 8:         $matches[token] \leftarrow \{\}$
 9:     $matches[token] \leftarrow matches[token] + loc$
10: $querys \leftarrow new\ collection$
11: for $(token, positions) \in matches$ do
12:     $qx \leftarrow new\ query$
13:     $qx.document \leftarrow q.document$
14:     $qx.pattern \leftarrow q.pattern + token$
15:     $qx.positions \leftarrow positions$
16:     $querys.add(qx)$
```

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

4.4.1 Sequences

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

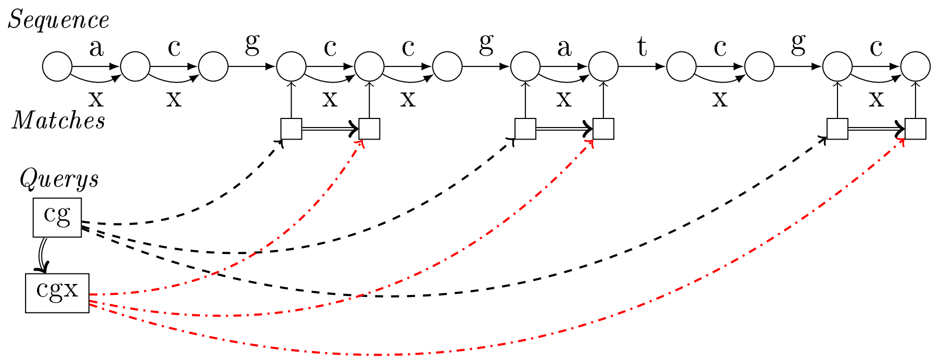
Let's consider a sequence $ACGCCGATCGC$ and a pattern CG.



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

4.4.2 Groups

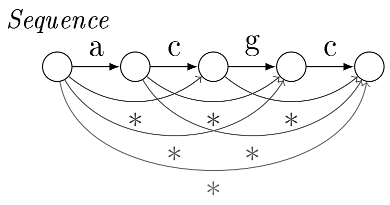
One common addition in a pattern language is capturing a group of tokens. For example we can use $X = [AC]$ to denote both tokens A, T. By adding where either one transitions we can capture such groups in the extension process.



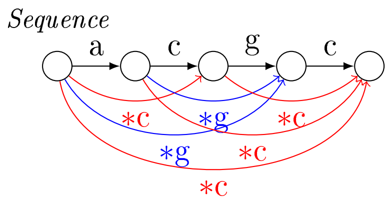
We shall omit the extension examples since the following of corresponding edges is trivial and anologous to previous examples.

4.4.3 Star

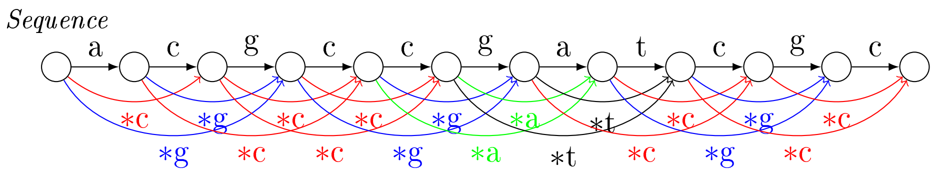
Another possible extension is the *dot-star* or more simply capturing a run of elements. Here we just show the expanded sequence with $*$ symbol.



Here we immediately notice how the complexity increases by introducing pattern token. Since this is a intermediary step isn't necessary we can instead extend with $*Y$, where Y is some other token. This means we avoid this single large query and have multiple smaller queries.



We can also limit the length of the run.



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

4.4.4 Optimizations

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

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

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

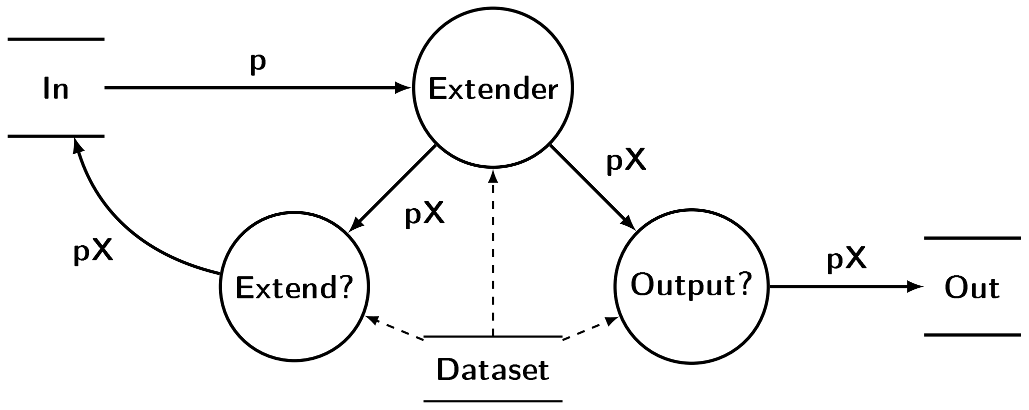
Chapter 5

Parallelization

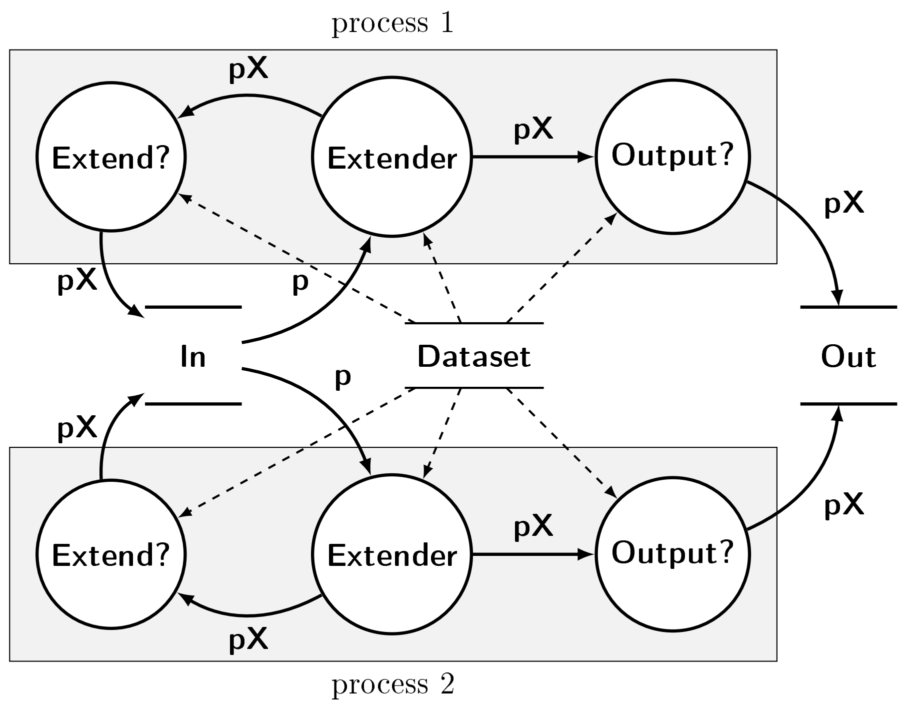
Here we describe how we can partition the algorithm to support parallelism. A proof of concept for fully parallelized code can be seen in appendix A.

5.1 Process

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



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



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

5.2 Extending

Since much of the work is done in *extender* it could be useful to parallelize it as well:

Algorithm 4 Parallel Extender with Group optimization

```
Input: Query $q$, function $next : \mathcal{I} \to [(\mathcal{I}, \mathcal{P})]$
Output: Set of querys that have been extended by one step
1: $nexts \leftarrow map(next, positions(q.matches))$
2: $matches \leftarrow groupBy(fst, nexts)$
3: $result \leftarrow map(queryMaker(q), matches)$
4: $optimized \leftarrow map((x) \to (union(matches(x))), groups)$
5: $returnunion(result, optimized)$
```

figure out how to make it readable

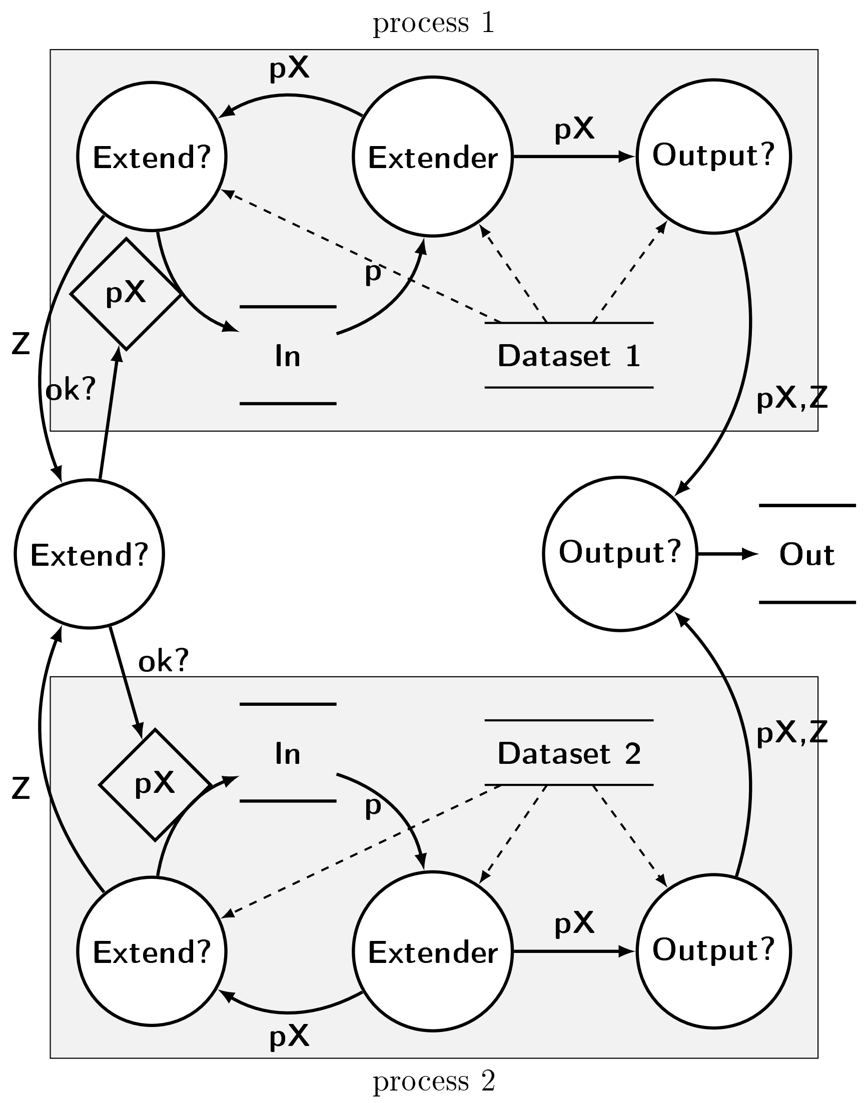
This practically may not give much improvement on conventional CPUs, but could be benefitial for highly parallel processors such as GPGPU or FPGA.

5.3 Distributed process

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

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

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



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

Chapter 6

Implementation

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

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

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

6.1 Architecture

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

We chose the following decomposition for the application:

Configuration structure for holding the configuration data
Setup based on the configuration initializes data-structures and functions for the algorithm
Reader reads in the data from files
Database a collection of datasets
Algorithm the SPEXS2 algorithm
Printer prints the result queries
Debugging utilities for the program

The algorithm was decomposed based on the generalization:

Pattern represents a pattern
Query stores the pattern with database and pattern
Set stores matching positions
Pool stores queries
Extender drives algorithms extending step
Feature computes a value from a query
Filter filters a query

6.2 Configuration

One problem with flexible algorithms is that they a lot of ways to be run. This often leads to having tens or hundereds of program flags. To avoid this problem we first decided to use a *json* file for the program configuration. Using *json* requires exact placement of commas and quotes that caused a lot of problems, hence we started using *yaml* that is more suitable for human input.

Short example of a configuration file.

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

For example, inside the configuration file:

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

Using *spexs2 –conf conf.json input=other* would replace the "File" value with "other". If there is no flag from command line is given, it will be replaced with "main".

6.3 Input and Output

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

6.4 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.

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

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

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

link to lock-free queues

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

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

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

Algorithm 5 priority queue push

```
1: $worst \leftarrow current\ worst$
2: **if** $worst > new\ element$ **then return**
3: $acquire\ lock$
4: $queue.put(new\ element)$
5: **if** $queue.length > queue.limit$ **then**
6:     $current\ worst \leftarrow queue.pop()$
7: $release\ lock$
```

6.5 Features and Filters

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

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

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

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

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

Example:

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

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

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

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

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

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

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

6.6 Synchronized Graph Traversal

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

The simplest form without synchronization is:

Algorithm 6 Graph traversal

```
Input: $graph$, $start$, $fn$
Output: All nodes in $graph$ get processed with $fn$
 1: $unvisited \leftarrow \{start\})$
 2: $visited \leftarrow \{\}$
 3: spawn
 4:     while $unvisited$ not empty do
 5:         $node \leftarrow unvisited.take()$
 6:         $visited.put(node)$
 7:         $fn(node)$
 8:         for $next \in neighbors(node)$ do
 9:             if $next \notin visited$ then
10:                 $unvisited.put(next)$
11:
12: $wait\ for\ workers$
```

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

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

Algorithm 7 Synchronized graph traversal

```
Input: $graph$, $start$, $fn$
Output: All nodes in $graph$ get processed with $fn$
 1: $added \leftarrow Semaphore(0)$
 2: $terminate \leftarrow false$
 3: $mutex \leftarrow Mutex()$
 4: $workers \leftarrow 0$
 5: $unvisited \leftarrow \{start\}$
 6: $visited \leftarrow \{\}$
 7: $added.signal()$
 8: spawn
 9:     while $true$ do
10:         $added.wait()$
11:         $mutex.lock()$
12:         if $terminate$ then
13:             $added.signal()$
14:             $mutex.unlock()$
15:             break
16:         $node \leftarrow unvisited.take()$
17:         $visited.put(node)$
18:         $workers \leftarrow workers + 1$
19:         $mutex.unlock()$
20:         $fn(node)$
21:         for $next \in neighbors(node)$ do
22:             $mutex.lock()$
23:             if $next \notin visited$ then
24:                 $unvisited.put(next)$
25:                 $added.signal()$
26:             $mutex.unlock()$
27:         $mutex.lock()$
28:         $workers \leftarrow workers - 1$
29:         if $workers = 0$ and $unvisited = \{\}$ then
30:             $terminate \leftarrow true$
31:             $added.signal()$
32:         $mutex.unlock()$
33:
34: $wait\ for\ workers$
```

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

6.7 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 picking")
        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 and we can inject different debugging statements without actually changing the algorithm.

Also we can now change the ways how to add debug info. One of would be a full stepwise debugger.

```
func debuggable(fn PickFunc) PickFunc {
    return func() Thing {
        mutex.lock()
        p := fn()
        while not continue
            interact with user
        mutex.unlock()
        return p
    }
}
```

Chapter 7

Applications and experimental results

7.1 Examples

Here we show examples for the program:

7.1.1 DNA sequences

make an example

7.1.2 Protein sequences

make an example

7.1.3 Text mining

make an example

7.1.4 Code mining

make an example

7.2 Performance

make an example

Chapter 8

Conclusions

results 1

results 2

context, compare

strength + limitations

so what? why is it important

what is next?

strong conclusions

take home message

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

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

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.