fast multiple sequence alignment with PAR

- Pile Anchors on Reference

David Gmelin

September 19, 2022

Contents

1 Introduction

The goal of `par` is to read a number of FASTA-sequences and to return their multiple alignment in MAF-Format[1]. The approach for calculating the alignment is based on the program phylonium [Klötzl and Haubold, 2019].

[1] `http://genome.ucsc.edu/FAQ/FAQformat#format5`

In *andi* [Haubold et al., 2014] and its faster successor *phylonium* [Klötzl and Haubold, 2019] the authors presented the idea of *anchor distances* to estimate evolutionary distances. Anchor distances are based on micro-alignments of regions that are anchored by exact matches. Both approaches proved to be orders of magnitudes faster than classical alignment tools while still being accurate on closely related genomes. Even among similar alignment-free methods they were found to be among the fastest. However, the actual sequence alignments that are used to generate the distance matrices stay implicit and cannot be accessed. The goal of this thesis is to make the alignments accessible and to provide them in a way that makes them comparable to classical alignment tools. The section 2 explains the main implementation of the program. In section 3 the [[seqUtil]] package is explained and implemented in more detail although some parts will already be shown before. This package contains some helpers that are in some sense related to sequences.
For andi [Haubold et al., 2014] and phylonium [Klötzl and Haubold, 2019] anchor alignments are build to estimate the distance between genomes. For [[par]] we take these anchor alignments and show the actual underlying explicit multiple sequence alignment (MSA).

2 Implementation

The program `par` has the following structure.

2a ⟨*par.go* 2a⟩≡
```
package main
import (
  ⟨Main Imports 3b⟩
)

const version = "0.11"

⟨Main 2b⟩
⟨Main Helpers 4b⟩
```
As in many other programs the main function defines the entry point. Here we read the user input, calculate the alignment, build the MSA and finally output the results. The main function initiates most of the tasks but relies on other functions that are called to do the actual work. The approximate workflow looks like this.

2b ⟨*Main* 2b⟩≡ (2a)
```
func main(){
  ⟨Handle User Input 3a⟩
  ⟨Build ESA 13a⟩
  ⟨Find Anchors 26⟩
  ⟨Pile MSA 31a⟩
  ⟨Output Results 31b⟩
}
```

The first step is to handle the user input.

3a ⟨*Handle User Input* 3a⟩≡ (2b)
⟨*Define Options* 3c⟩
⟨*Read Input Sequences* 4a⟩
⟨*Choose Reference* 6c⟩
⟨*Define Threshold* 7d⟩
⟨*Set NumCPUs* 27b⟩

To interact with the user we import the `flag` package.

3b ⟨*Main Imports* 3b⟩≡ (2a) 4c ▷
```
"flag"
```

We define different options to run the program, for example choosing the reference sequence, defining a custom minimum length for anchors or only printing the version. After the flags are parsed we can access them.

3c ⟨*Define Options* 3c⟩≡ (3a)
```
var optR = flag.String("r", "", "reference sequence.\n\t"+
                "Only the first sequence of this file will be handled as reference")
var optT = flag.Int("t", 0, "threshold for minimum anchor length.\n\t" +
  "If <=0 is specified it will be calculated according to the length of the sequence.")
//add numCPU
var optC = flag.Int("c", 0, "Number of Cores to use for alignment.\n\t" +
  "If < 1 it runs with the number of logical CPUs usable by the current process (runtime.NumC

var optRevComp = flag.Bool("revComp", true, "Build reference index for reverse complement.\n"

var optV = flag.Bool("v", false, "print par version, do not run")

//TODO add seed? -> Don't Need since we have no random elements
flag.Parse()

if *optV {
  fmt.Fprintf(os.Stderr, "par v.%s\n", version)
  return
}
```

We check optR to see if any specific reference sequence was selected. In this case we add this file at the first position of the other sequence files and remove it from the list if it is included in there as well. This means that the first sequence of the reference file will be handled as reference. If the reference file contains more than one sequence, subsequent sequences are treated as regular queries. The other input files or sequences for which the alignment has to be calculated are input without any flags.

4a ⟨*Read Input Sequences* 4a⟩≡ (3a)

```
seqFiles := flag.Args()
if len(seqFiles) < 1 {
  printUsage()
  return
}

if *optR != ""{
  r:= *optR
  for i, v := range seqFiles {
    if v == r {
      seqFiles = append(seqFiles[:i], seqFiles[i+1:]...)
    }
  }
  seqFiles = append([]string{r}, seqFiles...)
}
sequences:= readSequenceFiles(seqFiles)
```

To read the sequences from these files will be more work to do. We do this in a separate function to keep the main function as simple as possible.

4b ⟨*Main Helpers* 4b⟩≡ (2a) 5b▷

```
func readSequenceFiles(seqFiles []string) []fasta.Sequence {
⟨Function readSequenceFiles 5a⟩
}
```

2.1 Reading the Sequences

Par expects the input to be one or many files in `fasta`-format. The first line of a sequence contains the sequence description and starts with 'greater than' character (>). The succeeding lines contain the actual sequence data. There is no restriction on sequence length but it is recommended that all lines are shorter than 80 characters so the sequence data can span over multiple lines. Blank lines are not allowed.
For dealing with fasta sequences, we import the fasta package from `https://github.com/EvolBioInf/fasta`.

4c ⟨*Main Imports* 3b⟩+≡ (2a) ◁3b 5c▷

```
"github.com/evolbioinf/fasta"
```

This provides useful functions for reading the input sequences as well. Reading the fasta files contains roughly two steps - opening the file and scanning its contents. After reading every file we return the containing sequences.

5a ⟨*Function readSequenceFiles* 5a⟩≡ (4b)

```
sequences := make([]fasta.Sequence, 0)
for _, file := range seqFiles {
  f := openFile(file)
  //Scan File
  sc := fasta.NewScanner(f)
  //Append to existing sequences
  for sc.ScanSequence(){
    seq := sc.Sequence()
    ⟨Modify Input Sequence 6a⟩
    sequences = append(sequences, *seq)
  }
  f.Close()
}
return sequences
```

Again we want to keep it modular with a separate function `openFile` that we add to the helpers as well as the little function `log` that writes strings to `os.Stderr` and `printUsage`.

5b ⟨*Main Helpers* 4b⟩+≡ (2a) ◁4b 8▷

```
func openFile(file string) *os.File {
  f, err := os.Open(file)
  if err != nil {
    Log(fmt.Sprintf("couldn't open %q\nError:%s", file, err))
  }
  return f
}

func Log(msg string){
  fmt.Fprintf(os.Stderr, msg)
}

func printUsage(){
  fmt.Fprintf(os.Stderr, "par v.%s\nUsage: par [OPTIONS] FILES\n\nOptions:\n", version)
  flag.PrintDefaults()
}
```

We import the corresponding package `os` for interaction with the operating system.

5c ⟨*Main Imports* 3b⟩+≡ (2a) ◁4c 6b▷

```
"os"
```

After a sequence is loaded it has to be screened. We want to modify the header to only contain an identifier without any descriptions. The identifier is always the first word after the > character. Unfortunately there is no standard definition how the identifier is seperated from optional descriptions. We use the `strings.FieldsFunc` to check for a few options that could seperate the identifier. For the sequence data we cast all characters to uppercase since we will compare bytes. Also we replace any character that is not 'A', 'C', 'G' or 'T' with 'N'.

6a ⟨*Modify Input Sequence* 6a⟩≡ (5a)

```
f := func(c rune) bool {
  return unicode.IsSpace(c) || c == '|' || c == ','
}
h := strings.FieldsFunc(seq.Header(), f)[0]
data := strings.ToUpper(string(seq.Data()))
fMap := func(c rune) rune{
  if !(c == 'A' || c == 'C' || c == 'G' || c == 'T'){
  return 'N'
  }
  return c
}
data = strings.Map(fMap, data)
seq = fasta.NewSequence(h, []byte(data))
```

6b ⟨*Main Imports* 3b⟩+≡ (2a) ◁5c 7b▷

```
"strings"
"unicode"
```

In this step it would also be possible to remove characters other than ACGT.

2.2 Choosing the Reference

To build the ESA we need to pick a reference sequence. If a reference is given by the user we can check that again as we did before. Furthermore we know that in this case it is the first of our sequences. Choosing this is simple.

6c ⟨*Choose Reference* 6c⟩≡ (3a) 7c▷

```
//choosing the reference
var reference fasta.Sequence
if *optR != ""{
  reference = sequences[0]
  sequences = sequences[1:]
}else{
  ⟨Pick Median Sequence 7a⟩
}
```

If no reference is given we have to select one. There are several options to do this such as randomly, taking a fixed index or according to their length. In `phylonium` a sequence with median length is picked so we do it like this.
To get the median length and the associated sequence the sequences have to be sorted by the length of their actual sequence data. In Go, we can sort anything using the `Sort` package that implements the three methods `Len`, `Less` and `Swap` for Sort to apply. In our case this would be too much since this package has a working `sort.Slice` function already that only needs a `Less`-function for sorting slices. In our case this function compares the length of the sequence data.

7a ⟨*Pick Median Sequence* 7a⟩≡ (6c)
```
sort.Slice(sequences, func(i,j int) bool{
  return len(sequences[i].Data()) < len(sequences[j].Data())
})
refIdx := len(sequences)/2
reference = sequences[refIdx]
sequences = append(sequences[0:refIdx], sequences[refIdx+1:]...)
```

We import sort.

7b ⟨*Main Imports* 3b⟩+≡ (2a) ◁6b 12e▷
```
"sort"
```

In both cases we remove the picked sequence from the list of sequences.
We notify the user about the number of sequences and the chosen reference:

7c ⟨*Choose Reference* 6c⟩+≡ (3a) ◁6c
```
go Log(fmt.Sprintf("Selected %s for reference + %d queries.\n",
        reference.Header(), len(sequences)))
```

2.3 Anchor Threshold

The threshold defines the minimum length of a match to be an anchor. If none is given by the user or the given value is zero or smaller we define it similar to phylonium according to the length of the sequence and number of GC-pairs in the `AnchorThreshold` function.

7d ⟨*Define Threshold* 7d⟩≡ (3a)
```
threshold := *optT
if threshold < 1 {
  threshold = AnchorThreshold(reference.Data())
}
go Log(fmt.Sprintf("Set threshold to %d\n", threshold))
```

The anchor threshold is defined by the probability that a common substring or shustring [Haubold et al., 2009] for two unrelated sequences is below x. Sicne we dont want to match random sequences or fragments we want the probability to find a random match to be small. We set our threshold in a way such that 97.5% are below this. Consequently the probability to find a random match is set to 2.5%.

8 ⟨*Main Helpers* 4b⟩+≡ (2a) ◁5b 27c▷

```
func AnchorThreshold(refSeq []byte) int{
  l := len(refSeq)
  gc := seqUtil.NumGC(refSeq)
  p := float64(gc) / (2*float64(l))
  pp := 1 - 0.025
  x:= 0
  for seqUtil.ShustrProb(x, l, p) < pp {
    x ++
  }
  return x
}
```

2.4 Building the ESA

After we handled the user inputs we are ready to calculate the enhanced suffix array (ESA) for the reference sequence. To build the ESA we use the already existing GO package from `https://github.com/EvolBioInf/esa/`. This is a wrapper for the C library `libdivsufsort` [Fischer and Kurpicz, 2017] that was used in phylonium as well and is upon the fastest to build the enhances suffix array. In `EvolBioInf/esa` we already have a function to get the suffix array (SA) from a string and the longest common prefix (LCP) array from a SA. However, to find maximal matches we need more than the ESA and the LCP and we want to store these structures together to only have to calculate them once. We implement the package `esaMatcher` that builds up on the wrapper `evolbioinf/esa/` and contains functions to find matches in sequences using the structure of an ESA. We start by building the ESA, the matching part is explained further below in Section 2.6.

9 ⟨*esaMatcher.go* 9⟩≡

```
package esaMatcher

import(
  "github.com/evolbioinf/esa"
  ⟨ESA Imports 19b⟩
)

⟨Type ESA 12c⟩

⟨Type Interval 13b⟩

func BuildEsa(s []byte, revComp bool) MyEsa{
  ⟨ESA Constructor 10⟩
}

⟨Build Child Table 11⟩

⟨Matching Algorithms 15a⟩
```

Since the `esaMatcher` relies on the `EvolBioInf/esa`-package we import that straight away. Thanks to this package we can quickly build the SA and LCP inside the BuildESA function. With this function we can create our own ESA-structure that we call `MyEsa`. We will see later how this looks exactly. For building the ESA, we first check whether to include the reverse complement as well. In that case we seperate both strands with the unique character #. We also append a unique identifying character to the string as last character and remember the initial length of one strand.

10 ⟨*ESA Constructor* 10⟩≡ (9) 12d ▷

```
strandSize := len(s)
if revComp {
  revStr := append([]byte{'#'}, seqUtil.RevCompDna(s)...)
  s = append(s, revStr...)
}
s=append(s, byte('$'))
sa := esa.Sa(s)
lcp := esa.Lcp(s, sa)
// Add last element to lcp if necessary
if lcp[len(lcp)-1] != -1{
  lcp = append(lcp, -1)
}
```

⟨*Add CLD* 12b⟩

2.4.1 Child Table

To search effectively we want to further enhance the esa structure with child tables. The child table contains two child pointers, a left and a right one, CLD.L and CLD.R. The two child pointers point to local minima of their current interval inside the LCP array.

TODO: show example table and super cartesian tree

Also add definitions of LCP.L and R maybe?

Minima inside the lcp array indicate boundaries between lcp intervals and hence different paths through our suffix tree. We use these boundaries to navigate through the tree structure which can be described as "guided binary search" [Frith and Shrestha, 2018]. See Ohlebusch [2013, sec. 4.3.4], for more formal details. The construction of the child table is adapted from Ohlebusch [2013, Algorithm 4.11, p. 109].

TODO: Add Algorithm here?

We can merge CLD.L and CLD.R together to reduce memory requirements. If there is a pointer at CLD[i].L, CLD[i-1].R is always empty because CLD[i].L points to the minimum of the interval that ends at i-1. Since the boundary between the two intervals must be between i-1 and i there can not be any pointer CLD[i-1].R.

11 ⟨*Build Child Table* 11⟩≡ (9)

```
func BuildCld(lcp []int) []int{
  ⟨Init Stack 12a⟩

  n := len(lcp) - 1
  cld := make([]int, n+1)
  cld[0]=n
  push(0)
  var last int

  for k := 1; k <= n; k++ {
    for lcp[k] < lcp[top()]{
      last = pop()
      for lcp[top()]==lcp[last]{
        cld[top()] = last // CLD[k].R = CLD[k]
        last = pop()
      }
      if lcp[k] < lcp[top()] {
        cld[top()] = last // CLD[k].R = CLD[k]
      } else {
        cld[k - 1] = last // CLD[k].L = CLD[k-1].R = CLD[k-1]
      }
    }
    push(k)
  }
  return cld
```

```
}
```

For this algorithm we need a structure similar to a stack. This does not exist in GO but we can help ourselves using a slice. The stack functions are not really necessary but they reduce the code in the actual algorithm. Also this is a nice example of GO's anonymous functions that can be defined at the point of their usage.

12a ⟨*Init Stack* 12a⟩≡ (11)

```
stack := []int{}
top := func() int{
  return stack[len(stack)-1]
}
pop := func() int{
  t:= top()
  stack = stack[:len(stack)-1]
  return t
}
push := func(i int) {
  stack = append(stack, i)
}
```

We add the child array to the esa.

12b ⟨*Add CLD* 12b⟩≡ (10)

```
cld := BuildCld(lcp)
```

TODO: Add whatever is missing

We add the struct for MyEsa that holds a variable for every property of the ESA that we will use later on.

12c ⟨*Type ESA* 12c⟩≡ (9)

```
type MyEsa struct {
  s []byte
  sa []int
  lcp []int
  cld []int
  strandSize int
}

func (e *MyEsa) Sa() []int {return e.sa}
func (e *MyEsa) Lcp() []int {return e.lcp}
func (e *MyEsa) Cld() []int {return e.cld}
func (e *MyEsa) Sequence() []byte {return e.s}
func (e *MyEsa) StrandSize() int {return e.strandSize}
```

At the end of the BuildESA function we initialize and return this struct.

12d ⟨*ESA Constructor* 10⟩+≡ (9) ◁10

```
return MyEsa{s, sa, lcp, cld, strandSize}
```

We start building the ESA in our main function by calling `BuildEsa` and importing the `myesa` package

12e ⟨*Main Imports* 3b⟩+≡ (2a) ◁7b 27d▷

```
"github.com/dadidange/par/src/esaMatcher"
```

13a ⟨*Build ESA* 13a⟩≡ (2b)

```
//Construct the ESA
myesa := esaMatcher.BuildEsa(reference.Data(), *optRevComp)
go Log(fmt.Sprintf("finished building ESA\r"))
```

We continue with a helper structure for intervals in our esaMatcher package.

2.5 Interval

The type `Interval` represents an interval inside our ESA. Most importantly we need an index for its starting and ending position. For some special cases we also define its middle mid which is the end of its first child interval and the length l. The length l represents the length of the lcp at mid during `GetInterval` and `GetMatch`. When the actual match is returned we will write the length of the match to l.

13b ⟨*Type Interval* 13b⟩≡ (9)

```
type Interval struct{
  start int
  end int
  mid int
  l int
}
```

⟨*Interval Constructor* 14⟩

For most cases we can find the values for l and mid with the help of the child array. Let's take a look again at the left pointer CLD.L. CLD.L points to the first local minimum of its "left" (above) interval. Since i ends at position j, CLD[J+1].L points to the end of the first child interval of i normally. We only have to be careful for singletons and the last child interval. If the given interval {i,j} is the last child interval of some parent interval, CLD.L[j+1] might point to an minimum that starts before i since it always points to the first minimum of the interval {h,j} that ends there with $h < i \leq j$. For those two intervals that end at j, CLD[j+1].L points to the minimum of the larger interval, that is {h,j}. Hence we can not take this pointer to find the minimum of {i,j} In this case we can make use of CLD[h].R that points to the next minimum of {h,j}. Since {i,j} must be a child of {h,j} we can follow the right pointers until we will eventually arrive at the start index of {i,j}.
We add a constructor for `Interval` that does this to define the remaining variables mid and l. We also add an constructor for empty intervals. Keep in mind that both, the left and the right pointer are stored in a single list with CLD[i].L = CLD[i-1] and CLD[i].R = CLD[i] to save space.

14 ⟨*Interval Constructor* 14⟩≡ (13b)

```
func NewInterval(start, end int, e MyEsa) Interval{
  //Check for empty, invalid or singleton interval
  if (start >= end){
    //singleton
    if (start >=0){
      return Interval{start, end, start, e.lcp[end]}
    } else {
      //empty or invalid
      return EmptyInterval()
    }
  }

  m := e.cld[end] //CLD.L(m+1) = cld(m)
  for (m <= start){
    m = e.cld[m]
  }

  return Interval{start, end, m, e.lcp[m]}
}

func EmptyInterval() Interval{
  return Interval{-1, -1 , -1, -1}
}
```

2.6 Matching with ESA

In this section we will dig into the methods and functions that we use to find matches between two sequences using the ESA. Henceforth in section 2.7 we will apply these algorithms in our `main` function. Searching for matches between the sequences and the ESA that was build from the reference is the core of our program. We search for these matches using two algorithms presented in Ohlebusch [2013, Algorithm 5.1, 5.2] that were applied and further developed in Klötzl [2020, Listing 1.2, 1.3]. Furthermore these matches are selected for *Anchor Alignments*[Klötzl and Haubold, 2019].

15a ⟨*Matching Algorithms* 15a⟩≡ (9)

⟨*Get Interval* 15b⟩

⟨*Get Match* 17a⟩

⟨*Anchor Matches* 19a⟩

We start with the algorithm for `GetInterval`.

2.6.1 GetInterval

Given an interval i on the ESA and a character c `GetInterval` returns the subinterval of i that starts with c. The algorithms for `GetInterval` and `GetMatch` are crucial for our program. Together with the child array and the intervals they could be quite complicated to understand (at least that's how I felt). Hence I try to split them up to be able to explain them in more details for the next time I am (or anyone is) reading this. For a formal explanation Ohlebusch [2013, Section 4.3.1] provides some introduction.

15b ⟨*Get Interval* 15b⟩≡ (15a)

```
func (e *MyEsa)GetInterval(i Interval, c byte) (Interval){
  ⟨Check Singleton Interval 15c⟩
  ⟨Loop ChildIntervals 16a⟩
  ⟨Final Interval Check 16c⟩
}
```

The first step is to check if i is a singleton interval, i.e. it contains only one character. In this case there are two options. If i starts with c we can return this. Otherwise we return an empty interval.

15c ⟨*Check Singleton Interval* 15c⟩≡ (15b)

```
// Check Singleton Interval
if i.start == i.end{
  if(e.s[e.sa[i.start]] == c){
    return i
  } else {
    //Return empty interval
    return EmptyInterval()
  }
}
```

If i is not a singleton interval we want to loop through the subintervals one level below the given interval, lets call them child intervals. We initialize our interval by setting the upper and lower bounds. The first child interval starts where i starts and ends mid, the first local minimum.

16a ⟨*Loop ChildIntervals* 16a⟩≡ (15b) 16b▷

```
lower := i.start
upper := i.mid
l := i.l
```

We also set the variable l as the LCP of this interval. Every child interval has a common prefix with length l except the last one, so this defines our first looping condition. At the end of each loop we increment the boundaries for the interval by setting the next start to the current end. The new end of the interval can be found at CLD.R[m] which points to the next minimum right of the current interval. If the current interval starts with c we found the right child.

16b ⟨*Loop ChildIntervals* 16a⟩+≡ (15b) ◁16a

```
for e.lcp[upper] == l {
  if (e.s[e.sa[lower]+l] == c){
    //match found
    return NewInterval(lower, upper-1, *e)
  }
  //increment interval boundaries
  lower = upper
  //check for singleton
  if (lower == i.end){
    break
  }
  upper = e.cld[upper] //CLD.R(m) = cld(m)
}
```

After the loop we do a final check if the last interval starts with c or if we could not find any match.

16c ⟨*Final Interval Check* 16c⟩≡ (15b)

```
if (e.s[e.sa[lower] + l] == c){
  return NewInterval(lower, i.end, *e)
} else {
  return EmptyInterval()
}
```

2.6.2 GetMatch

`GetInterval` enables us to find matching segments between our reference and any query sequence. More precisely we can find the longest prefix of the query that matches any suffix of the reference. For every character in the query we can call `GetInterval` with the child interval returned by the previous character. Its implementation is adapted from Klötzl [2020, Section 1.6] and Ohlebusch [2013, Algorithm 5.2]. We initialize the first interval to point at the first and last element of the reference. Lets look at `GetMatch` closely.

17a ⟨*Get Match* 17a⟩≡ (15a) 18b ▷

```
func (e *MyEsa) GetMatch(query []byte) Interval{
  in := NewInterval(0, len(e.s)-1, *e)
  cld := EmptyInterval()
  k := 0
  m := len(query)
  for k < m{
    cld = e.GetInterval(in, query[k])
    ⟨GetMatch: Check Empty Interval 17b⟩

    ⟨GetMatch: Compare Prefix 18a⟩
  }
```

When `GetInterval` returns something, we first have to check if anything was found at all. If an empty interval gets returned no child interval starts with the character we were looking for. If this is the first iteration in `GetMatch` we know that there is no child interval at all in our ESA that starts with the charcter we are looking for, and thus no match. In case of $k > 0$ there were child intervals that match up to this position so we return them and set k as the length of the match.

17b ⟨*GetMatch: Check Empty Interval* 17b⟩≡ (17a)

```
if (cld.start == -1 && cld.end == -1){
  if (k == 0){
    return cld
  }
  in.l = k
  return in
}
```

If the interval is not empty a child interval that starts with the current query character was found. The next step is to try to extend the match but we can not just continue to call `GetInterval` for the next character. If every sequence in the subinterval has a common prefix these positions must be skipped. Since they do not act as identifier for child intervals they could lead us to wrong intervals. We know that the pointer CLD[j+1].L points to the minimum in the LCP-Interval that ends at j, except for singletons. Thus we can get the LCP at this position and only compare these sequences directly since they are the same for every element in the child interval anyway. We only have to be careful for singletons and not to exceed the query. In both cases we can loop through the rest of the sequence, not l and compare the characters. In case of a mismatch we return our current interval and set the match length to the current position. If there is not mismatch so far we try to extend our matches with the next call of `GetInterval`.

18a ⟨*GetMatch: Compare Prefix* 18a⟩≡ (17a)

```
k++ //the k-th character was matched in
in = cld
l := in.l
if(in.start == in.end || l > m){
  l = m
}

for saIdx:=e.sa[in.start]; k < l; k++ {
  if(e.s[saIdx+k] != query[k]){
    in.l = k
    return in
  }
}
```

Finally if we get through all the loops above without finding mismatches the complete query matches our reference. We return the leftover interval before we continue with the anchoring.

18b ⟨*Get Match* 17a⟩+≡ (15a) ◁17a

```
  in.l = m
  return in
}
```

2.6.3 Anchor Sequences

The anchoring step is the last one in our process of finding matches. The algorithm to find anchors is adapted from [Klötzl, 2020, Listing 3.1]. It finds a list of regions that are homologous in the reference and a query sequence and returns these homologous regions. The corresponding method `FindAnchors` relies on `GetMatch` and `GetInterval`. To find anchors and anchor pairs we iterate through the query. To be precise we iterate through every suffix of our query since `GetMatch` returns the longest prefix of a query that is contained in the reference sequence. For every match we find we check if its unique and longer than our threshold and can be added to our anchors. We add an anchor by either extending the last anchor or starting with a new one.
The structure of the function looks like this:

19a ⟨*Anchor Matches* 19a⟩≡ (15a)

```
func (e *MyEsa) FindAnchors(query []byte,
                        threshold int) []seqUtil.Homology{
  var homs []seqUtil.Homology
  ⟨FindAnchors Init Vars 20⟩

  ⟨Define isAnchor 21⟩

  for currentQ < qLen{
    // Find anchor
    if isAnchor(){
      ⟨Handle Anchor 23a⟩
    }
    //proceed in query
    advance()
  }

  // Check last
  if (lastWasRight || lastLen >= 2*threshold){
    if !currentHom.IsFwd(){
      currentHom.ReverseRefCoords(strandBorder)
    }
    homs = append(homs, currentHom)
  }
  return homs
}
```

19b ⟨*ESA Imports* 19b⟩≡ (9)

```
"github.com/dadidange/par/src/seqUtil"
```

The return type of `Find Anchors` is a slice of the type `Homology` that is implemented in the `seqUtil` package. It contains the starting position on both sequences, its length and its strand direction. Further methods help with comparing homologies.

At the top of `FindAnchors` we initialize the variables that have to be accessible throughout the method such as the current position in the query or the ending positions of the last anchor. We also add the function literal or closure advance() to define the number of positions to increment for every loop. For a complete mismatch we only advance one position, otherwise we advance to the first position we mismatched and continue our comparison there.

20 ⟨*FindAnchors Init Vars* 20⟩≡ (19a)

```
qLen := len(query)
strandBorder := e.StrandSize()

lastEndQue:= 0
currentQ := 0

lastEndRef := 0
lastWasRight := false
var matchL int
var lastLen int
var currentRef int

var currentMatch Interval
currentHom := seqUtil.NewHomology(0,0,0, true)

advance := func() {
  if matchL > 0 {
    currentQ += matchL
    return
  }
  currentQ ++
}
```

To check whether a match is an anchor is done in the closure `isAnchor`. Closures are convenient since we can access our current variables but also reduce the code inside the loop. We use another closure `naive` to check if we can extend our match naïvely. Here we check if the current match was only interrupted by a few mismatches and continue to compare the nucleotides directly without using the ESA structure.

21 ⟨*Define isAnchor* 21⟩≡ (19a)
⟨*Naive Match* 22⟩

```
isAnchor := func () bool{
  //match naïve
  if naive(){
    return true
  }

  //Find Regular Match
  currentMatch = e.GetMatch(query[currentQ:])
  matchL = currentMatch.l
  //uniqueness, minimum length
  if(currentMatch.start != currentMatch.end ||
            matchL < threshold){ return false }
  currentRef = e.sa[currentMatch.start]
  return true
}
```

22 ⟨*Naive Match* 22⟩≡ (21)

```
naive := func() bool{
  //Check if we can just continue comparing our queries after some mismatch
  if currentQ - lastEndQue - lastLen > threshold{
    return false
  }

  q := currentQ
  step := q - lastEndQue
  r := lastEndRef + step
  match := 0
  for r < len(e.s) && q < qLen {
    if query[q] == e.s[r] {
        match++
        q++
        r++
    } else {
        break
    }
  }
  if match > threshold{
    matchL = match
    currentRef = r - match
    return true
  }
  return false
}
```

If we find anchors we have two options to deal with them. They can extend the previous anchor and form a pair or group. Or, if they cannot form a pair, we store the previous anchor and continue with the current anchor to be the first of a potential new group. So how can we form anchor groups? **Two or more anchors can form a group if they are equidistant on reference and query sequence.** We also have to check that the anchors do not match different strands.

23a ⟨*Handle Anchor* 23a⟩≡ (19a)

```
isFwd := currentRef < strandBorder
// check for anchor pair
if(currentQ - lastEndQue == currentRef - lastEndRef &&
  isFwd == currentHom.IsFwd()){
  //set new End to start of current query + matchlen
  currentHom.Expand(currentQ+matchL)
  lastWasRight = true
} else {
  // We can not build anchor pair
  // Start new left one and append the old
  ⟨Start New Anchor 23b⟩
}
// set variables for next iteration
lastLen = matchL
lastEndQue = currentQ + matchL
lastEndRef = currentRef + matchL
```

If an anchor group cannot be extended we store it. We only append anchor groups or anchors that are twice as long as the threshold. Before a homology gets appended we check the direction of the strand we mapped against. If we were looking at the reverse strand we have to change the starting and ending coordinates on the reference. For details see the `ReverseRefCoords` in `seqUtil` (section 3)

23b ⟨*Start New Anchor* 23b⟩≡ (23a)

```
if (lastWasRight || lastLen >= 2*threshold){
  if !currentHom.IsFwd(){
    currentHom.ReverseRefCoords(strandBorder)
  }
  homs = append(homs, currentHom)
}
currentHom = seqUtil.NewHomology(currentQ, currentRef, matchL, isFwd)
lastWasRight = false
```

2.6.4 Sort Homologies

In practice, `FindAnchors` returns a list of homologeous segments for the reference and the query sequence. These homologies are arranged in the order they occur in the query sequence. We have to sort them in the appropriate order in the reference sequence. We use `sort.Slice` again.

23c ⟨*Sort Homologies* 23c⟩≡ (27a)

```
sort.Slice(h, func(i,j int) bool{
 return h[i].StartsLeftOnRef(h[j])
})
```

2.6.5 Filter for Overlaps

It is also possible that homologies occur within a sequence. This leads to overlapping segments on the reference sequence within a query. They have to be filtered since this would lead to a sequence aligning itself for the MSA. To filter for overlapping segments is more complicated than sorting. There are several ways to do this but our goal is to align as much nucleotides as possible. We do this by iterating through the homologies after they are sorted according to their start on the reference sequence. For each sequence we determine its predecessor with the highest score that does not overlap and set the score of this sequence to $predecessor.Score + own.Score$. We define the score of a sequence as the length of the longest chain that ends with this sequence. The `FilterOverlaps` function belongs to `seqUtil` as well.

24 ⟨*Filter Overlaps* 24⟩≡ (38)

```
func FilterOverlaps(homs []Homology) []Homology{
  m := len(homs)
  if m == 1 {
    // nothing to filter
    return homs
  }
  // Initialize slices for scores and predecessors
  pred := make([]int, m)
  score := make([]int, m)
  predNum := make([]int, m)
  totalMax := -1
  totalMaxIdx := 0

  ⟨Set Predecessors 25a⟩

  ⟨Select Path 25b⟩
  return reHoms
}
```

To set the predecessor for each segment we iterate through the homologies that end earlier on the reference and search for the highest scoring one. We also remember the highest total score and the number of predecessors for each homology.

25a ⟨*Set Predecessors* 25a⟩≡ (24)

```
pred[0] = -1
score[0] = homs[0].Len()

for i := 1; i<m; i++ {
    maxIdx := -1 // or NaN
    maxScore := 0
    for j := 0; j<i; j++{
    if(homs[j].EndsLeftOnRef(homs[i]) &&
                          score[j] > maxScore){
      maxIdx = j
      maxScore = score[j]
    }
  }
  pred[i] = maxIdx
  sc := maxScore + homs[i].Len()
  score[i] = sc
  if (maxIdx >= 0){
    predNum[i] = predNum[maxIdx] + 1
  }
  if (sc > totalMax){
    totalMax = sc
    totalMaxIdx = i
  }
}
```

We select the homology with the highest total score and its predecessors for the correct arrangement of non-overlapping segments for this sequence.

25b ⟨*Select Path* 25b⟩≡ (24)

```
n := predNum[totalMaxIdx]
reHoms := make([]Homology, n+1)
h := totalMaxIdx
for i:=n; i >=0; i-{
  reHoms[i] = homs[h]
  h = pred[h]
}
```

2.7 Apply Matching

In this section we will apply the functions we introduced earlier to our program. Our goal is to map (potentially) multiple queries against one reference sequence. Since these queries are independent there is potential for parallelization. Go has so-called *goroutines* that enable concurrency. They can be considered as very lightweight threads and can be started using the *go* statement before the function call. The function that is called after *go* runs in a separate goroutine. The program continues straight away and does not wait for this function to finish.

Note on Concurrency I do not want to fail to mention that even though concurrency and parallelism are similar they are not identical. This is stated regularly when dealing with goroutines. There are several posts, articles and further information about this across the internet including a conference talk by Rob Pike with the title *"Concurrency is not Parallelism"*[2]:

> "Concurrency is about dealing with lots of things at once.
> Parallelism is about doing lots of things at once."

Parallelism It is difficult to imagine our program to be actually concurrent. Most of what we do depends on the steps we did before such that we can not make them independent. But since the queries do not depend on each other we can at least process them in parallel.
To do so we have to keep two things in mind. First all sequences share the common list they belong to and the corresponding indices. We do not want to mix them up since it would be difficult to assign the homologies to the sequence later on. Also since the program does not wait after the go statement we have to make sure to wait at the end until all sequences are processed. We can wait by adding a simple channel `ch` to our function to whom we report at the end of each goroutine.

26 ⟨*Find Anchors* 26⟩≡ (2b)

```
ch := make(chan struct{})

⟨Process Queries 27a⟩

// Wait for completion
for range sequences {
  <- ch
}
go Log(fmt.Sprintf("Finished matching\r"))
```

[2] `https://www.youtube.com/watch?v=oV9rvDllKEg`, accessed June 27th 2022

For processing the queries we put them into an anonymous function that gets the current index as argument parameter. We add this as explicit argument for an important reason. The variable i is updated for each loop iteration. If we would use it similar to how we use the two slices $homs$ and $sequences$ we can not make sure that the value has not changed by the time the function is executed.

27a ⟨*Process Queries* 27a⟩≡ (26)
```
n := len(sequences)
homs := make([][]seqUtil.Homology, n)

for i := 0; i < n; i++ {
  go func (i int){
    h := myesa.FindAnchors(sequences[i].Data(), threshold)
    ⟨Sort Homologies 23c⟩
    h =  seqUtil.FilterOverlaps(h)
    homs[i] = h
    Log(fmt.Sprintf("Fin %d\r", i))
    //Report Completion
    ch <- struct{}{}
  }(i)
}
```

Using goroutines we can run in parallel but we don't have to. In the beginning we defined the number of CPUs as a user input that can be set to one core as well. We handle the input like this

27b ⟨*Set NumCPUs* 27b⟩≡ (3a)
```
SetNumCpus(*optC)
```

27c ⟨*Main Helpers* 4b⟩+≡ (2a) ◁8 28▷
```
func SetNumCpus(n int){
  max := runtime.NumCPU()
  if n > max {
    runtime.GOMAXPROCS(max)
    s := fmt.Sprintf("Could not set numCPU to %d since max is %d.", n, max)
    Log(fmt.Sprintf("Warn: %s Set to max.\n", s))
    return
  }
  if n < 1 {
    n = max
  }
  runtime.GOMAXPROCS(n)
  Log(fmt.Sprintf("Set numCPU to %d\n", n))
}
```

The function `runtime.NumCPU()` returns the number of CPUs that are usable by the current process. Using `runtime.GOMAXPROCS()` we can set the number of simultaneous executing CPUs.

27d ⟨*Main Imports* 3b⟩+≡ (2a) ◁12e 31d▷
```
"github.com/dadidange/par/src/seqUtil"
"runtime"
```

2.8 Pile Alignment

With the homologies sorted and filtered for overlaps we can start to pile up the sequences to print our sequence alignment. Again, let's think about what we already have and what we aim to achieve. We have a list of homologeous segments per sequence that store their starting and ending position on both, the reference and the query. For piling the sequences on top of each other their positions on the reference are important. Our target is to output the aligned sequences in MAF[3] format. The MAF format stores a series of MSAs in alignment blocks that contain one sequence per line. We consider one block to consist of different anchors that overlap between the sequences. We want to find segments between the queries that overlap on the reference and group them together. We add a new type `MafBlock` that helps us to deal with these overlapping segments to the main helpers as well as other functions to deal with this type. For the implementation of the type and some helper functions see 2.10. Subsequently we will implement the functions to pile and print the actual alignment.

28 ⟨*Main Helpers* 4b⟩+≡ (2a) ◁27c 31c▷
⟨*Maf Section* 29⟩

[3] `http://genome.ucsc.edu/FAQ/FAQformat#format5`

In here we store the homologies that belong to a common group or MAF-block, that is they overlap. We use the ending positions later for printing the results correctly. The piling process is implemented in the function `pileBlocks` in the main package. It returns a slice of MafBlocks from *seqUtil*.

29 ⟨*Maf Section* 29⟩≡ (28) 37 ▷

```
func pileBlocks(homs *[][]seqUtil.Homology, numSeqs int) ([]MafBlock, bool) {

  finElements := 0
  nextElement := make([]int, numSeqs)

  //Check not to extend homologies
  checkNext := func (next, i int) bool{
    if next >= len((*homs)[i]){
      //reached end of this sequence
      finElements ++
      return false
    }
    return true
  }

  for i, next := range nextElement{
    checkNext(next, i)
  }
  if finElements >= numSeqs {
    return nil, false
  }

  var blocks []MafBlock
  var minStart, minStartIdx int

  ⟨Pick Earliest Start 30a⟩

  ⟨Pile Blocks 30b⟩

return blocks, true
}
```

The slice *nextElement* points to the index of the next homology to be processed for every sequence. Using `checkNext` we can notice when we have processed every homology for the respective sequence. If this happens before we even start there are no homologies and we return an empty slice. In this case there is no alignment at all for the given sequences and reference.
To build an alignment block we pick the segment with the earliest start with `SetNextMin` on the reference sequence and add this to our block.

30a ⟨*Pick Earliest Start* 30a⟩≡ (29)

```
SetNextMin := func(){
  minStart = math.MaxUint32
  for i, next := range nextElement{
    if next < len((*homs)[i]){
      if((*homs)[i][next].StartR() < minStart){
        minStart = (*homs)[i][next].StartR()
        minStartIdx = i
      }
    }
  }
}
```

Next we iterate through the homologies that are left and try to extend our block. As long as we find an overlapping segment we add this to the block and increment the counter in *nextElement*. If no segment overlaps with the block any more we stop this round. We start again with a new block and the remaining sequences from which we pick the earliest. This is repeated until no more homologies are left.

30b ⟨*Pile Blocks* 30b⟩≡ (29)

```
for finElements < numSeqs {
  SetNextMin()

  h := (*homs)[minStartIdx][nextElement[minStartIdx]]
  nextElement[minStartIdx]++

  b := NewMafBlock(minStart, minStartIdx, h)
  added := true
  for added {
    added = false
    finElements = 0
    for i, next := range nextElement{
      if !checkNext(next, i){
        continue
      }
      if b.Contains((*homs)[i][next]){
        b.AddItem(i, (*homs)[i][next])
        nextElement[i]++
        added = true
      }
    }
  }
  blocks = append(blocks, b)
}
```

We call `pileBlocks` in `main()` and check if we could find any homologies. If we ddid we can proceed and print our results otherwise we terminate.

31a ⟨*Pile MSA* 31a⟩≡ (2b)

```
blocks, foundAny := pileBlocks(&homs, n)
if !foundAny {
    Log("no match found, stopping with empty alignment.\nTry a different reference sequence.\
    return
}
```

2.9 Output Results

We print our results in the multiple alignment format (MAF) to the standard output. MAF-files start with a mandatory header line and an optional comment line indicated by '##' and '#' respectively. We add a comment line that contains the parameters we used to run this program. To print the actual alignment we iterate through our alignment blocks. The method `MafString` returns a string that contains their alignment already in correct format for printing.

31b ⟨*Output Results* 31b⟩≡ (2b)

```
//MAF Header
fmt.Printf("%s\n",
  "##maf version=1 scoring=none")
fmt.Println(ToMafInfo(reference.Header(), threshold, *optRevComp))
numBlocks := len(blocks)
for i, b := range blocks{
  go Log(fmt.Sprintf("Building Block %d of %d\r", i+1, numBlocks))
  fmt.Println(b.MafString(reference, &sequences))
}
```

We add the `ToMafInfo` function to our helpers.

31c ⟨*Main Helpers* 4b⟩+≡ (2a) ◁28

```
func ToMafInfo(ref string, thres int, includeReverse bool) string{
  s := fmt.Sprintf("#par v.%s, command: par -r %s -t %s -revComp=%t\n",
                   version, ref, strconv.Itoa(thres), includeReverse)
  return s
}
```

31d ⟨*Main Imports* 3b⟩+≡ (2a) ◁27d 45▷

```
"strconv"
```

Internally `MafString` breaks down a `MafBlock` into multiple blocks if necessary. It returns its content in correct formatting for MAF. For the structure of a MAF-file we used the MAF specification [4] and the scheme from Dutheil et al. [2014, Fig. 1] for orientation. We break a multiple alignment block down into multiple blocks according to the ends of the contained homologies. The first block starts at the start of the first homologeous position and ends at the earliest end of one of the contained homologies. The remaining homologies continie at the next position as well as some possible new homologies that overlap the next segment.

To do this we first have to sort the ending indices and remove possible duplicates. The order in which this is more effective depends on the number of duplicates. For many duplicates it is more effective to remove them before sorting. If there are only a few duplicates we rather sort them first since this speeds up the removing process. Let's sort them first but remember this for possible performance tests. `RemoveDuplicateInt` is a private function inside `seqUtil`. See section 3 for the actual code.

32a ⟨*remove Duplicates* 32a⟩≡ (32b)

```
sort.Ints(b.ends)
b.ends = seqUtil.RemoveDuplicateInt(b.ends)
```

The structure of `MafString` looks like this. The resulting text is stored using a `strings.Builder`. This helps to build the blocks efficiently instead of concatenating all the strings.

32b ⟨*MAF Printing* 32b⟩≡ (37) 34a▷

```
func (b *MafBlock) MafString(refSeq fasta.Sequence, queries *[]fasta.Sequence) string{

  ⟨remove Duplicates 32a⟩

  var blockstr strings.Builder
  items := &b.items
  start := b.start
  refName := refSeq.Header()
  refData := refSeq.Data()

  //Iterate through all end points
  for _,end := range b.ends{
    ⟨Iterate Block Intervals 33a⟩
    }
    return blockstr.String()
}
```

[4] urlhttps://genome.ucsc.edu/FAQ/FAQformat.html#format5

To build the different blocks we iterate through all the ending points. At the end of each loop we set the new start to the current end to proceed at the next interval. For MAF a new block starts with an 'a' and an optional alignment score which we set to zero. We also print the reference sequence as first sequence line. Each sequence line starts with a 's' followed by six fields that describe the aligned sequence with the following order and meaning:

1. name of the sequence
2. starting index of the alignment in the sequence, zero based
3. alignment size, that is the number of aligned characters and is equal to the number of non-dash characters
4. strand; '$+$' for forward strand or '$-$' for alignment with the reverse complement
5. total size of the sequence. This is always the same if the sequence appears in multiple blocks.
6. the actually aligning nucleotides. Dashes indicate Gaps/ insertions.

To get the other aligning homologies we check if it overlaps with the current interval. If it does `GetMafLine` returnes the line for the given parameters.

33a ⟨*Iterate Block Intervals* 33a⟩≡ (32b)

```
//take next homology from all items
s := fmt.Sprintf("a score=0\ns %-10s\t %d %d + %d %s\n",
      refName, start, end-start, len(refData), refData[start:end])
blockstr.WriteString(s)
for idx, homs := range *items{
  h := homs[0]
    if h.StartR() < end{
      s := GetMafLine(h, start, end, (*queries)[idx])
      blockstr.WriteString(s)
      //remove from items if ends
      ⟨Check Remove Homology 33b⟩
    }
  }
  blockstr.WriteString(fmt.Sprintln())
  start = end
```

When we add a homology to a block we check if it exceeds the block as well. If it does not we remove this so we can continue with the next homology from this sequence. If it was the last one we remove the complete sequence from our items.

33b ⟨*Check Remove Homology* 33b⟩≡ (33a)

```
if h.EndR() <= end {
  if len(homs) > 1{
    (*items)[idx] = (*items)[idx][1:]
  } else {
    delete(*items, idx)
  }
}
```

The GetMafLine function returns the corresponding segment from a homology within given borders, that is the interval $[start, end)$. Unfortunately it is necessary to convert between different positions on the Sequences which makes this code not very handsome.
We start by initializing some of the properties we need to print a correct sequence line in MAF. Next we check if we are looking at the forward or reverse strand and reverse the sequence if necessary.

34a ⟨*MAF Printing* 32b⟩+≡ (37) ◁32b

```
func GetMafLine(h seqUtil.Homology, start, end int, seq fasta.Sequence) string{
  //Init fields for MAF sequence line
  src := seq.Header()
  size := end - start
  startQ := h.StartQ()
  endQ := h.EndQ()

  var seqStr []byte
  var text string
  var strand string

  srcSize := len(seq.Data())

  overlap := h.StartR() - start
  gap := ""

  ⟨MafLine: Find Correct Segment 34b⟩

  ⟨MafLine: Return 36⟩
}
```

To find the correct nucleotide sequence on the query we have to distinguish between the forward and the reverse strand since this changes the indeces but the principle is the same. We have to consider that for the reverse strand the sequence does not start at the beginning of our interval but at the end. Hence if we want to skip some characters at the front we have to actually remove them from the end and vice versa. There is a way to avoid that distinction by computing the reverse complement before we think of the correct indices but this is not effective. We would compute the reverse complement multiple times for the same area and only actually use a part of it.

34b ⟨*MafLine: Find Correct Segment* 34b⟩≡ (34a)

```
⟨Handle Forward Strand 35a⟩
⟨Handle Reverse Strand 35b⟩
```

Lets look at the closure handleFwd as example fot the general procedure to finding the correct indices. We will see that this does not differ that much for the reverse part.
We use $overlap$ to check where a homology starts relative to our current interval. If the overlap is negative it starts before the interval and we have to skip some of its characters at the beginning. The characters to be skipped is the difference between our interval and the actual start of the homology, i.e. the positive value of *overlap*. For a positive overlap we have to introduce a gap before the actual start. We can use the overlap again to find out the length of the gap. Note that only for a negative overlap is the size of the aligning region $end - start$.

35a ⟨*Handle Forward Strand* 35a⟩≡ (34b)

```
handleFwd := func ()  {
  if overlap < 0{
    //skip positions -> h begins before current interval
    startQ = startQ - overlap
    endQ = startQ + size
  }else{
    //insert gap and reduce size of actual aligning region
    gap = strings.Repeat("-", overlap)
    size = size - overlap
    endQ = startQ + size
  }
  seqStr = seq.Data()[startQ: endQ]
  strand = "+"
}
```

We do the same for the reverse strand only that we come from the end of the homology this time and take the reverse complement at the end. This way the reverse complement is only calculated for the section we actually need.

35b ⟨*Handle Reverse Strand* 35b⟩≡ (34b)

```
handleRev := func ()  {
    if overlap < 0{
      //skip positions at the end this time since they get reverted
      end= h.EndQ() + overlap
      startQ = end - size
    }else{
      gap = strings.Repeat("-", overlap)
      size = size - overlap
      end = h.EndQ()
      startQ = end-size
    }
    s := seq.Data()[startQ: end]
    seqStr = seqUtil.RevCompDna(s)
    strand = "-"
  }
```

Finally we check the homology direction to decide which strand direction we are dealing with. We append the gap, which is empty for $overlap < 0$ to the sequence and return the corretly formatted line.

36 ⟨*MafLine: Return* 36⟩≡ (34a)

```
if h.IsFwd(){
  handleFwd()
}else{
  handleRev()
}

text = gap + string(seqStr)

return fmt.Sprintf("s %-10s\t %d %d %s %d %s\n",
            src, startQ, size, strand, srcSize, text)
```

2.10 MafBlock

A `MafBlock` contains overlapping homologies that form a common group. This helps us to provide a structure for piling the sequences to form a MSA that can be written to a file or the standard output. The homologies are stored in the map *items* with type `map[int][]Homology`. This map stores a slice of homologies for a given index. We also store all ends of the homologies, the maximum end and the earliest start. This enables us to iterate through the ends for printing the block in correct MAF format.

37 ⟨*Maf Section* 29⟩+≡ (28) ◁29

```
type MafBlock struct{
  items map[int][]seqUtil.Homology
  ends []int
  start int
  maxEnd int
}

func NewMafBlock(start, idx int, h seqUtil.Homology) MafBlock {
  b := MafBlock{make(map[int][]seqUtil.Homology), []int{}, start, 0}
  b.AddItem(idx, h)
  return b
}

func (b *MafBlock) AddItem(idx int, h seqUtil.Homology){
  b.items[idx] = append(b.items[idx], h)
  e:= h.EndR()
  b.ends = append(b.ends, e)
  if e > b.maxEnd{
    b.maxEnd = e
  }
}

func (b *MafBlock) Contains(h seqUtil.Homology) bool{
  return h.StartR() >= b.start && h.StartR() < b.maxEnd
}

func (b *MafBlock) String() string{
  return fmt.Sprintf("MafBlock: Start=%d, End=%d, ends=%v",
                                          b.start, b.maxEnd, b.ends)
}
```

⟨*MAF Printing* 32b⟩

3 seqUtil: Sequence Utils Package

The `seqUtil` package implements helpers to deal with sequences and similar structures. It is sort of a companion for most of what we do during this program. For example it contains the `Homology` type.

3.1 Homology

38 ⟨*seqUtil.go* 38⟩≡ 41 ▷

```
package seqUtil

import (
  ⟨seqUtil Imports 39b⟩
)

⟨Type Homology 39a⟩

⟨Filter Overlaps 24⟩
```

The type Homology holds the relevant properties of an homologous region. Internally we only need to have four variables to describe the homology - the starting positions at the respective sequences, the length of the match and the direction. Apart from the getter functions and a constructor, we add the Expand function as well. This allows us to extend an homology which is useful in the process of finding the anchors. String() allows us to properly print the homology if it is necessary.

39a ⟨*Type Homology* 39a⟩≡ (38) 40a▷

```
type Homology struct{
  startQ int
  startR int
  homLen int
  isFwd bool
}

func (h *Homology) Len() int{ return h.homLen}
func (h *Homology) StartR() int{ return h.startR}
func (h *Homology) EndR() int{ return h.startR + h.homLen}
func (h *Homology) StartQ() int{ return h.startQ}
func (h *Homology) EndQ() int{ return h.startQ + h.homLen}
func (h *Homology) IsFwd() bool{ return h.isFwd}

func NewHomology(startQ, startR, homLen int, isFwd bool) Homology{
  return Homology{startQ, startR, homLen, isFwd}
}

func (h *Homology) Expand(newQEnd int){
  h.homLen = newQEnd - h.startQ
}

func (this *Homology) String() string{
  dir := "fwd"
  if !this.isFwd{
    dir = "rev"
  }
  s := fmt.Sprintf("Homology: startR=%d, startQ=%d, len=%d, dir=%s",
        this.startR, this.startQ, this.homLen, dir)
  return s
}
```

39b ⟨*seqUtil Imports* 39b⟩≡ (38) 42c▷

```
"fmt"
```

We also add some functions to compare different homologies. For example it can be useful to know which starts earlier on the reference or query sequence or to know if they overlap.

40a ⟨*Type Homology* 39a⟩+≡ (38) ◁39a

```
func (this *Homology) StartsLeftOnRef(other Homology) bool{
  return this.startR < other.startR
}

func (this *Homology) StartsLeftOnQ(other Homology) bool{
  return this.startQ < other.startQ
}

func (this *Homology) EndsLeftOnRef(other Homology) bool{
  return this.EndR() <= other.startR
}

func (this *Homology) Overlaps(other Homology) bool{
  if this.startR == other.startR {return true}

  if this.StartsLeftOnRef(other) {
    return !this.EndsLeftOnRef(other)
  } else {
    return other.EndsLeftOnRef(*this)
  }
}
```

⟨*ReverseRefCoords* 40b⟩

The method `ReverseRefCoords` helps us to remap the coordinates on the reference sequence when we deal with a homology which is known to be matching the reverse strand but the coordinates are still flipped. This happens because we usually match the forward query to the reverse reference sequence. For our MSA we rather want the query sequence in reverse and the reference sequence forward. Therefore we invert the position where we start at the reference. The starting and ending position on the query are still correct since we store the forward strand as well. But we have to remember this when we are aligning the sequences later on and reverse the query if we need the reverse strand.

40b ⟨*ReverseRefCoords* 40b⟩≡ (40a)

```
func (h *Homology) ReverseRefCoords(refLen int){
  startR := h.startR
  h.startR = refLen - (startR - refLen) - h.homLen + 1
}
```

TODO: Insert sketch

We add the `RemoveDuplicateInt` function to seqUtil. This function removes duplicates from a list of integers. This is useful when we use the ends of our homologies to print the MAF.

41 ⟨*seqUtil.go* 38⟩+≡ ◁38 42a▷

```
func RemoveDuplicateInt(intSlice []int) []int {
    allKeys := make(map[int]bool)
    sl := []int{}
    for _, item := range intSlice {
        if _, value := allKeys[item]; !value {
            allKeys[item] = true
            sl = append(sl, item)
        }
    }
    return sl
}
```

3.2 ShustrProb

We use the function `ShustrProb` to define the threshold for the minimum anchor length. This is adapted from Haubold et al. [2009, Eq. 4]. We get the probability that a match between two random, unrelated sequences of lenght l is below x with $2p$ as the GC-content of the sequence.

$$P(X^* \leq x) = \sum_{k=0}^{x} 2^x \binom{x}{k} p^k \left(\frac{1}{2} - p\right)^{l-k} \left(1 - p^k \left(\frac{1}{2} - p\right)^{l-k}\right)^l \tag{1}$$

42a ⟨*seqUtil.go* 38⟩+≡ ◁41 42b▷

```
⟨GODOC ShustrProb 44c⟩
func ShustrProb(x, l int, p float64) float64{
        xFl := float64(x)
        lFl := float64(l)

        lExp2 := math.Pow(2,xFl)
        bin := func (n,k int) float64 {
                v := Binomial(int64(n), int64(k))
                return float64(v)
        }

        var sum float64 = 0
        for k:= 0; k<=x; k++{
                kFl := float64(k)
                tmp := math.Pow(p,kFl) * math.Pow(0.5 - p, xFl - kFl)
                b := bin(x,k)
                sum += lExp2 * b * (tmp * math.Pow(1 - tmp, lFl))


                if sum >= 1.0 {
                        return 1.0
                }
        }
        return sum
}
```

For calculating the binomial coefficient we have to use the `math.big` package.

42b ⟨*seqUtil.go* 38⟩+≡ ◁42a 43▷

```
func Binomial(n, k int64) int64{
        z := new(big.Int).Binomial(n,k)
        return z.Int64()
}
```

We import `math` and `math.big`.

42c ⟨*seqUtil Imports* 39b⟩+≡ (38) ◁39b 44b▷

```
"math"
"math/big"
```

NumGC is helpful for setting the anchor threshold as well. This returns the molecules in the DNA-sequence that are either guanine (G) or cytosine (C). For this we just iterate through the sequence.

43 ⟨*seqUtil.go* 38⟩+≡ ◁42b 44a▷

```
//Counts the number of nucleotides that are either 'G' or 'C'
func NumGC(seq []byte) int{
        c := byte('C')
        g := byte('G')
        count := 0
        for _,char := range seq {
                if char == c || char == g {
                        count ++
                }
        }
        return count
}
```

3.3 Reverse Complement

We also want to be able to find matches with the reverse strand. `RevCompDna` enables us to compute the reverse complement of a sequence. We use the `bytes.Map` function that allows to chance certain characters according to a mapping function.

44a ⟨*seqUtil.go* 38⟩+≡ ◁43

```
func RevCompDna(seq []byte) []byte{
  n := len(seq)
        revSeq := make([]byte, n)

        //Reverse
        for i, j := 0, n-1; i < j; i, j = i+1, j-1 {
                                        revSeq[i], revSeq[j] = seq[j], seq[i]
        }

        //Complement
        f := func (r rune) rune  {
          switch{
          case r == 'A':
                return 'T'
          case r == 'T':
                return 'A'
          case r == 'G':
                return 'C'
          case r == 'C':
                return 'G'
          default:
                return '-'
          }
        }
        return  bytes.Map(f, revSeq)
}
```

44b ⟨*seqUtil Imports* 39b⟩+≡ (38) ◁42c

```
"bytes"
```

4 GODOC

In this section we only define the headers for some crucial functions and methods. GODOC is the documentation tool for Go.

44c ⟨*GODOC ShustrProb* 44c⟩≡ (42a)

```
//Given the length l of a sequence and x ShustrProb returns probabilityt hat a
// match between two random, unrelated sequences of lenght l is below x with 2p
// as the GC-content of the sequence.
```

5 MAF format

We import `fmt` for printing in main.

45 ⟨*Main Imports* 3b⟩+≡ (2a) ◁31d

```
"fmt"
"math"
```

References

Julien Y. Dutheil, Sylvain Gaillard, and Eva H. Stukenbrock. Maffilter: a highly flexible and extensible multiple genome alignment files processor. *BMC Genomics*, 15, 2014. doi: 10.1186/1471-2164-15-53.

Johannes Fischer and Florian Kurpicz. Dismantling divsufsort, 2017. URL `https://arxiv.org/abs/1710.01896`.

Martin C. Frith and Anish M. S. Shrestha. A simplified description of child tables for sequence similarity search. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 15(6):2067–2073, 2018. doi: 10.1109/TCBB.2018.2796064.

Bernhard Haubold, Peter Pfaffelhuber, Mirjana Domazet-Los ̆o, and Thomas Wiehe. Estimating mutation distances from unaligned genomes. *Journal of Computational Biology*, 16(10):1487–1500, 2009. doi: 10.1089/cmb.2009.0106. URL `https://doi.org/10.1089/cmb.2009.0106`. PMID: 19803738.

Bernhard Haubold, Fabian Klötzl, and Peter Pfaffelhuber. andi: Fast and accurate estimation of evolutionary distances between closely related genomes. *Bioinformatics*, 31(8):1169–1175, 12 2014. ISSN 1367-4803. doi: 10.1093/bioinformatics/btu815. URL `https://doi.org/10.1093/bioinformatics/btu815`.

Fabian Klötzl. *Fast Computation of Genome Distances*. PhD thesis, University of Lübeck, Oct 2020.

Fabian Klötzl and Bernhard Haubold. Phylonium: fast estimation of evolutionary distances from large samples of similar genomes. *Bioinformatics*, 36(7):2040–2046, 12 2019. ISSN 1367-4803. doi: 10.1093/bioinformatics/btz903. URL `https://doi.org/10.1093/bioinformatics/btz903`.

Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*. Oldenbusch Verlag, 2013.