ancs: approximate alignment using anchors

Ivan Tsers

July 16, 2024

Contents

1 Introduction

A local alignment of two similar nucleotide sequences can be approximated as long exact matches (anchors) separated by mismatches. The use of enhanced suffix arrays (ESA) for finding the exact matches makes such an approximation fast compared to an actual alignment. The anchor-based comparison of whole genomes was implemented in a number of programs including andi [**?**] phylonium [**?**], which are used for fast computation of evolutionary distances. In addition, the method is used for finding intersections between multiple target genomes in the program fur [**?**].

The package ancs contains functions for anchor-based approximation of local alignments.

2 Implementation

Package `ancs` provides functions for finding anchors and processing homologies.

2a ⟨*ancs.go* 2a⟩≡
```
package ancs
import (
        ⟨Imports 3c⟩
)
⟨Data structures 2b⟩
⟨Methods 2c⟩
⟨Functions 3a⟩
```

2.1 Data structure `Seg`

We declare a custom data type `Seg` for storing segments of the forward strand of the subject. A segment has a start `s` and a length `l`.

2b ⟨*Data structures* 2b⟩≡ (2a)
```
type Seg struct {
        s int
        l int
}
```

2.2 Methods

2.2.1 `End()`

Method `End()` returns an end of a segment.

2c ⟨*Methods* 2c⟩≡ (2a) 2d▷
```
func (seg *Seg) End() int {
        return seg.s + seg.l - 1
}
```

2.2.2 `NewSeg()`

Constructor method `NewSeg()` returns an empty segment.

2d ⟨*Methods* 2c⟩+≡ (2a) ◁2c
```
func NewSeg() Seg {
        return Seg{s:0, l:0}
}
```

2.3 Functions

The `ancs` package contains a number of functions.

3a ⟨*Functions* 3a⟩≡ (2a) 6a▷
⟨*SortByStart* 3b⟩
⟨*MinAncLen* 3d⟩
⟨*FindHomologies* 4a⟩
⟨*ReduceOverlaps* 7d⟩
⟨*TotalSegLen* 9d⟩
⟨*PrintSegSiteRanges* 10a⟩
⟨*SegToFasta* 11⟩

2.3.1 `SortByStart()`

`SortByStart()` accepts a slice of segments `s` and sorts the segments by their start positions in ascending order.

3b ⟨*SortByStart* 3b⟩≡ (3a)
```
func SortByStart(s []Seg) []Seg {
        sort.Slice(s, func(i, j int) bool {
                return s[i].s < s[j].s
        })
        return s
}
```

We import `sort`.

3c ⟨*Imports* 3c⟩≡ (2a) 3e▷
```
"sort"
```

2.3.2 `MinAncLen()`

An anchor is a unique non-random match. Its non-randomness is defined by its length. The threshold length, or the minimum anchor length, is found from the null shustring length distribution for a sequence with given length and GC content.

The function `MinAncLen()` accepts length, GC content of a sequence, and the threshold probability, and calculates the minimum anchor length.

3d ⟨*MinAncLen* 3d⟩≡ (3a)
```
func MinAncLen(l int, g float64, t float64) int {
          x := 1
          cq := 0.0
          for cq < t {
                  x++
                  cq = cq + sus.Prob(l, g, x)
          }
          return x
}
```

We import `sus`.

3e ⟨*Imports* 3c⟩+≡ (2a) ◁3c 4b▷
```
"github.com/evolbioinf/sus"
```

2.3.3 `FindHomologies()`

The function `FindHomologies()` accepts a `fasta` query sequence, an enhanced suffix array and length of the subject sequence, and the minimum anchor length `a`. The function returns a slice of segments (homologies) and a bool map of segregation sites.

We initialize variables to operate with in the search of homologies, conduct the search, and return its results.

4a ⟨*FindHomologies* 4a⟩≡ (3a)
```
func FindHomologies(
          query *fasta.Sequence,
          e *esa.Esa,
          subjectLen int,
          a int) ([]Seg, map[int]bool) {
          ⟨Initialize search 4c⟩
          ⟨Anchor search 5⟩
          ⟨Return the output 7c⟩
}
```

We import `fasta` and `esa`.

4b ⟨*Imports* 3c⟩+≡ (2a) ◁3e 10b▷
```
"github.com/ivantsers/fasta"
"github.com/evolbioinf/esa"
```

We declare variables to operate with during the search. These are:

- current and previous positions in the query;
- current match: its length and start in the subject;
- previous match: its length, start and end the subject;
- current segment;
- a slice of segments to store the output in;
- a map to store positions of segregation sites (mismatches);
- an indicator of right anchor;
- length of a `single strand` of the subject;
- length of the query.

As one might notice, we calculate the full subject's length before calling `FindHomologies()`, and we calculate the query's length inside the function. This design is intentional, so the same subject can be compared to multiple queries.

4c ⟨*Initialize search* 4c⟩≡ (4a)
```
var qc, qp int
var currLen, currStartS int
var prevLen, prevStartS, prevEndS int
var seg Seg
var h []Seg
n := make(map[int]bool)
rightAnchorFound := false
subjectStrandLen := subjectLen/2
queryLen := query.Length()
```

We perform anchor search for each prefix in our query. We get a prefix that starts at the current position and ends at the last byte of the query. Then we search for a long unique match using the function `anchorLongMatch()`, which we still have to write.

If such a match is found, we proceed with calculating the end positions of the previous match in the query and the subject. Then we analyze the current match and decide, whether the current segment can be extended with it. If yes, we extend the current segment. If it cannot be extended and the right anchor is found, we open a new segment and save the current one in `h`. After these operations, we remember the current match and jump in the query by at least one nucleotide (a guaranteed mismatch) before proceeding with the next prefix.

5 ⟨*Anchor search* 5⟩≡ (4a)

```
for qc < queryLen {
        queryPrefix := query.Data()[qc:queryLen]
        if anchorLongMatch(&currStartS, &currLen,
                n, a, queryPrefix, e) {
                prevEndQ := qp + prevLen
                prevEndS = prevStartS + prevLen
                ⟨Analyze the match 6b⟩
                if segCanBeExtended {
                ⟨Extend the current segment 6c⟩
                } else {
                        if rightAnchorFound || prevLen / 2 >= a {
                                ⟨Close the current segment 6d⟩
                        }
                        ⟨Open a new segment 7a⟩
                }
                ⟨Remember the current match 7b⟩
        }
        qc = qc + currLen + 1
        fmt.Println()
}
//Close the last segment if open:
if rightAnchorFound || prevLen / 2 >= a {
        if seg.s > subjectStrandLen {
                seg.s = subjectLen + 1 - seg.s - seg.l
        }
        h = append(h, seg)
}
```

The function `anchorLongMatch()` accepts pointers to length of the current match and its start in the subject, a map of segregation sites, minimum anchor length, current query prefix, and a pointer to the subject ESA. The function returns a boolean and updates the start and the length of the current match.

A match is unique if it starts and ends at the same position in an ESA. In terms of a suffix tree, a unique match ends on a leaf.

6a ⟨*Functions* 3a⟩+≡ (2a) ◁3a 9b▷

```
func anchorLongMatch(
          currStartS, currLen *int,
          n map[int]bool,
          a int,
          queryPrefix []byte,
          e *esa.Esa) bool {
          mc := e.MatchPref(queryPrefix)
          newStartS := e.Sa[mc.I]
          newCurrLen := mc.L
          *currStartS = newStartS
          *currLen = newCurrLen
          //Add the guaranteed mismatch to the segsite map
          n[newStartS + newCurrLen] = true
          lu := (mc.J == mc.I) && (newCurrLen >= a)
          return lu
}
```

We analyze if the match

- starts in the subject after the previous match;
- is equidistant with the previous match in the subject and query;
- is located on the same strand in the subject as the previous match.

If these criteria are met, we qualify the current segment as extendible.

6b ⟨*Analyze the match* 6b⟩≡ (5)

```
afterPrev := currStartS > prevEndS
areEquidist := qc - prevEndQ == currStartS - prevEndS
onSameStrand := (currStartS < subjectStrandLen) ==
          (prevStartS < subjectStrandLen)
segCanBeExtended := afterPrev && areEquidist && onSameStrand
```

We extend the segment to the end of the new anchor and add positions between the end of the last left anchor and the start right anchor to the map of mismatches as keys. We also remember that we have just found a new right anchor.

6c ⟨*Extend the current segment* 6c⟩≡ (5)

```
seg.l = seg.l + qc - prevEndQ + currLen
rightAnchorFound = true
```

To close the current segment is to project it onto the forward strand (if necessary) and append it to the slice of homologies..

6d ⟨*Close the current segment* 6d⟩≡ (5)

```
if seg.s > subjectStrandLen {
          seg.s = subjectLen + 1 - seg.s - seg.l
}
h = append(h, seg)
```

To open a segment is to declare its start and length, and forget that the right anchor was found.

7a ⟨*Open a new segment* 7a⟩≡ (5)
```
seg.s = currStartS
seg.l = currLen
rightAnchorFound = false
```

We update previous position in the query and the previous match.

7b ⟨*Remember the current match* 7b⟩≡ (5)
```
qp = qc
prevLen = currLen
prevStartS = currStartS
```

We return the output of `FindHomologies()` if the slice of homologies is not empty. If it is, we notify the user and exit.

7c ⟨*Return the output* 7c⟩≡ (4a)
```
if len(h) == 0 {
        fmt.Fprintln(os.Stderr, "No homologous regions found\n")
        os.Exit(0)
}
return h, n
```

2.3.4 `ReduceOverlaps()`

`ReduceOverlaps()` accepts a sorted slice of segments (homologies) and returns a slice of segments, which contains the longest chain of non-overlapping homologies.

If `h` contains less than two elements, we return early. Otherwise, we reduce overlapping stacks of homologies to the longest chain of co-linear non-overlapping homologies using the algorithm for two-dimensional fragment chaining. We start with initializing variables, then we calculate chain scores, find links of the longest chain through backtracking, and return the chain.

7d ⟨*ReduceOverlaps* 7d⟩≡ (3a)
```
func ReduceOverlaps(h []Seg) []Seg {
        hlen := len(h)
        if hlen < 2 {
                return h
        }
        ⟨Initialize chaining 8a⟩
        ⟨Calculate chain score 8b⟩
        ⟨Backtrack the chain 9a⟩
        ⟨Return the chain 9c⟩
}
```

We declare variables describing the chain. For each homology `i` we initialize:

- `predecessor`—the previous homology in the longest chain ending before `i`);
- `score`—the length of the longest chain ending at `i`. The initial score of the chain is the first homology's length;
- `visited`—whether `i` was visited in backtracking of chain links.

We also initialize the first elements for `score` and `predecessor`.

8a ⟨*Initialize chaining* 8a⟩≡ (7d)
```
predecessor := make([]int, hlen)
score := make([]int, hlen)
visited := make([]bool, hlen)
score[0] = h[0].l
predecessor[0] = -1
```

We calculate the chain score to maximize the number of non-overlapping segments in `h`. Starting from the second homology `h[1]`, we traverse each homology `h[i]` in the sequence. For each `h[i]`, we find the preceding homology `h[k]` that can form the longest chain ending before `h[i]` starts, ensuring no overlap.

8b ⟨*Calculate chain score* 8b⟩≡ (7d)
```
for i := 1; i < hlen; i++ {
          maxScore := 0
          maxIndex := -1
          for k := 0; k < i; k++ {
                  if h[k].End() < h[i].s {
                          if score[k] > maxScore {
                                  maxScore = score[k]
                                  maxIndex = k
                          }
                  }
          }
          predecessor[i] = maxIndex
          if maxIndex != -1 {
              score[i] = h[i].l + score[maxIndex]
          } else {
              score[i] = h[i].l
          }
}

// Debug messages. Will be removed in the future
//fmt.Println("***Homologies:")
//for _, el := range(h) {
//      fmt.Printf("***(%d, %d)\n", el.s, el.End())
//}
//fmt.Println("***Scores:", score)
```

Now that we have scores for all homologies, we can find `s`, the index of an element of `h` with the highest score. This element is the final link of the chain we are looking for, and its score is the total score of it. To find `s`, we use the function `argmaxMapInt()`, which we still have to write. Then we backtrack through the predecessor links and mark visited homologies.

Now that we have calculated the scores for all homologies, we need to find `s`, the index of the homology in `h` with the highest score. This homology represents the final link in the chain we are looking for, and its score reflects the total score of the chain. To find `s`, we use the function `argmax()`, which we still have to write. After identifying `s`, we backtrack through the `predecessor` links to trace the entire chain and mark the corresponding homologies as visited.

9a ⟨*Backtrack the chain* 9a⟩≡ (7d)
```
s := argmax(score)
for s != -1 {
        visited[s] = true
        s = predecessor[s]
}
```

The function `argmax()` returns the index of the maximum value in the input slice of integers.

9b ⟨*Functions* 3a⟩+≡ (2a) ◁6a 10c▷
```
func argmax(x []int) int {
    maxIdx := 0
    for i := 1; i < len(x); i++ {
        if x[i] > x[maxIdx] {
            maxIdx = i
        }
    }
    return maxIdx
}
```

We extract only visited elements of `h` and return the reduced slice.

9c ⟨*Return the chain* 9c⟩≡ (7d)
```
var hred []Seg
for i := 0; i < len(h); i++ {
        if visited[i] {
                hred = append(hred, h[i])
        }
}
return hred
```

(TotalSegLen()) accepts a slice of segments and returns their total length.

9d ⟨*TotalSegLen* 9d⟩≡ (3a)
```
func TotalSegLen(segments []Seg) int {
        sumlen := 0
        for _, s := range(segments) {
                sumlen += s.l
        }
        return sumlen
}
```

2.3.5 `PrintSegSiteRanges()`

`PrintSegSiteRanges()` accepts a bool map of Ns (segregation sites), a pointer to an output file, and prints their coordinate ranges.

10a ⟨*PrintSegSiteRanges* 10a⟩≡ (3a)

```
func PrintSegsiteRanges(m map[int]bool, file *os.File) {
        if len(m) == 0 {
                fmt.Fprintf(file, "No segregation sites found\n")
        } else {
                k := append([]int{-1}, getSortedIntKeys(m)...)
                k = append(k, -1)
                for i := 1; i < len(k) - 1; i++ {
                        prev := k[i] == k[i-1]+1
                        next := k[i] == k[i+1]-1
                        if prev && next {
                                continue
                        }
                        if next {
                                fmt.Fprintf(file, "[%d", k[i]+1)
                        } else if prev {
                                fmt.Fprintf(file, ":%d] ", k[i]+1)
                        } else {
                                fmt.Fprintf(file, "%d ", k[i]+1)
                        }
                }
                fmt.Fprintf(file, "\n")
        }
}
```

We import `os` and `fmt`.

10b ⟨*Imports* 3c⟩+≡ (2a) ◁4b

```
"os"
"fmt"
```

We define `getSortedKeys()`.

10c ⟨*Functions* 3a⟩+≡ (2a) ◁9b

```
func getSortedIntKeys(m map[int]bool) []int {
        keys := make([]int, 0, len(m))
        for k, _ := range m {
                keys = append(keys, k)
        }
        sort.Ints(keys)
        return keys
}
```

2.3.6 `SegToFasta()`

`SegToFasta()` converts a slice of segments into actual fasta sequences. It accepts a slice of segments, a pointer to the corresponding ESA, and a map of Ns. It returns a slice of pointers to fasta entries (type fasta.Sequence).

11 ⟨*SegToFasta* 11⟩≡ (3a)

```
func SegToFasta(segments []Seg,
          e *esa.Esa,
          n map[int]bool) []*fasta.Sequence {
          var segfasta []*fasta.Sequence
          for i, s := range(segments) {
                  start := s.s
                  end := s.End()
                  var data []byte
                  for j := start; j < end + 1; j++ {
                          if n[j] {
                                  data = append(data, 'N')
                          } else {
                                  data = append(data, e.T[j])
                          }
                  }
          segname := fmt.Sprintf("Segment_%d (%d..%d)", i+1, start+1, end + 1)
          converted := fasta.NewSequence(segname, data)
          segfasta = append(segfasta, converted)
          }
          return segfasta
}
```

References