A new bioinformatics programming framework

Bioinformatics Group
School of Molecular and Biomedical Science
The University of Adelaide

BioInfoSummer 7 December 2012

Outline

- Motivation
- Go programming language
- The bíogo project
- Example use case: repeat cluster classification
- Summary

Motivation

Investigation of repetitive elements

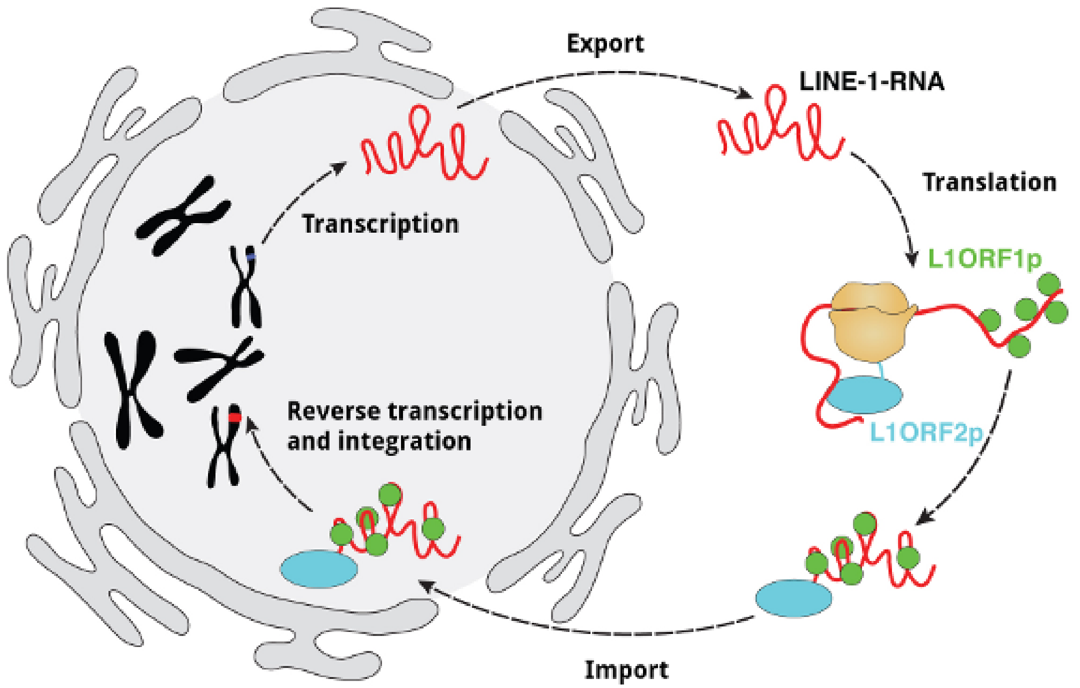
Our group investigates the evolution and function of transposable elements in animals and how they contribute to genome evolution.

Primary focus is on retrotransposable elements and their interactions with the genomic environment.

Additionally, we are interested in *de novo* repeat discovery, classification and annotation.

Retrotransposable elements

Retrotransposable elements (RTEs) are a class of mobile repetitive elements that transpose via a cut and paste reverse-transcriptase-dependent mechanism.

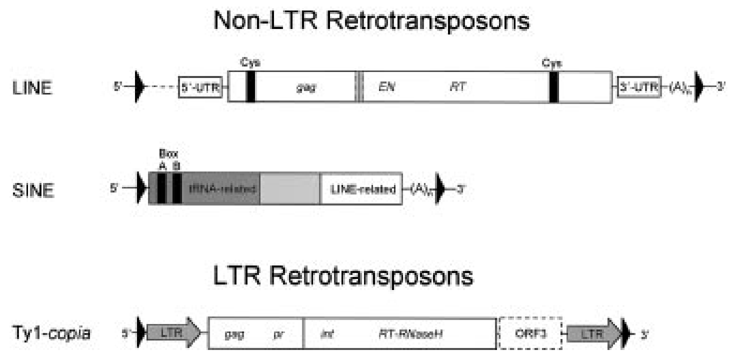


after MPG Tübingen 2009

Retrotransposable elements

Include:

- Long INterspersed nuclear Elements (LINEs)
- Short INterspersed nuclear Elements (SINEs)
- Endogenous retroviruses/LTR retrotransposons



```
after Schmidt 1999
```

We focus on LINEs and SINEs and their interactions: SINEs are replication defective and so are dependent on LINEs for transposition.

Problems with working with mobile sequences

How to find new elements?

When is a chimera just a TinT event and when is it a new type of element?

How to deal with interrupted elements?

Finding repeats to work with

Repeat annotation approaches fall into two main categories:

- homology-based repeat-library search *e.g.* RepeatMasker/RepBase
- whole-genome self-alignment *e.g.* PALS/PILER

We use both methods to complement each other.

Neither approach seems adequate in isolation, with current tools.

RepeatMasker/RepBase

www.repeatmasker.org(http://www.repeatmasker.org)

RepeatMasker takes a curated library of known repeats and uses this to identify regions of a genome that are similar to these elements.

Annotation of regions determined by similarity score with library elements.

Limitations:

- unable to find novel repeats
- members of large divergent families often confusingly annotated
- chimeric repeats are not identified as single units

PALS/PILER

www.drive5.com/pals/ (http://www.drive5.com/pals/)

www.drive5.com/piler/ (http://www.drive5.com/piler/)

The PALS/PILER *de novo* annotation pipeline consists of two phases:

- whole-genome self-alignment — PALS
- clustering of identified repeat intervals into families — PILER

Limitations:

- limit on the number of repeat regions identifiable
- significant limitation on identification of divergent/sparse families
- problems with identifying interrupted/chimeric elements
- code stability

Addressing limitations of PALS/PILER

PALS

Design decisions in PALS result in a limit on the number of regions that are searchable:

PALS is written in C/C++ with 32 bit integers as the index type

- revise to make use of 64 bit integers on 64 bit architectures: ✔

Candidate repetitive regions are stored in memory for sorting, merging and analysis

- revise to do disk-based sort: ✘ (PALS' architecture makes this unviable)

PALS is single threaded:

- revise to use at least concurrent processing of each strand: ✔/✘

PILER

PILER condenses aligned regions (images) into non-contiguous intervals (piles).

A graph-based clustering of piles is then performed.

The approach used is fast and simple.

Possibly too simple.

PILER-DF (dispersed family) algorithm

```
Type image:
    genomic interval.
```

```
Type pile:
    list of images covering a maximal contiguous region of copy count > 0.
```

```
Function pile-of(Q image):
    Return pile containing Q.
```

```
Function is-global-image(Q image, g real):
    If |Q ∩ pile-of(Q)| ≥ |pile-of(Q)|·g:
        Return true
    Return false.
```

```
Function PILER-DF(P piles, t integer):
    Let G be a graph where nodes represent repeat instances
    For each pile p in P:
        For each image Q in p:
            Let T = partner(Q)
            If is-global-image(Q) and is-global-image(T):
                Add edge p—pile-of(T) to G
    Return connected components of G of order ≥ t.
```

Improving PILER-DF

There are two obvious areas for improvement for our use case:

- construction of piles - unrelated repeats may fall into the same pile

```
Type pile:
    list of images covering a maximal contiguous region of copy count > 0.
```

- family size - for divergent or rare repeats, t may equal 2 (default = 3)

```
Function PILER-DF(P piles, t integer):
    Let G be a graph where nodes represent repeat instances
    For each pile p in P:
        For each image Q in p:
            Let T = partner(Q)
            If is-global-image(Q) and is-global-image(T):
                Add edge p—pile-of(T) to G
    Return connected components of G of order ≥ t.
```

Family size parameter

Family size - for divergent or rare repeats, t may equal 2:

- trivial test: run piler with -famsize=2

```
Function PILER-DF(P piles, t integer):
    Let G be a graph where nodes represent repeat instances
    For each pile p in P:
        For each image Q in p:
            Let T = partner(Q)
            If is-global-image(Q) and is-global-image(T):
                Add edge p—pile-of(T) to G
    Return connected components of G of order ≥ t.
```

- result ↝ ~100% increase in number of families returned — many are LINE/SINE

Identified repeats may require further classification.

Essentially not interesting for this discussion, but something worth following up.

Pile construction

Construction of piles - unrelated repeats may fall into the same pile

- less trivial: within each pile, cluster images to give a set of clusters

```
Type cluster:
    a pile containing images with similar end points.
```

```
Function clusterise(P pile):
    Return list of clusters.
```

```
Function PILER-DF-NEW(P piles, t integer):
    Let G be a graph where nodes represent repeat instances
    For each pile p in P:
        Let C = clusterise(p)
        For each cluster c in C:
            For each image Q in c:
                Let T = partner(Q)
                If is-global-image(Q) and is-global-image(T):
                    Add edge c—cluster-of(T) to G // by analogy to pile-of()
    Return connected components of G of order ≥ t.
```

Rewrite: What language?

Neither PALS nor PILER are friendly to third party revisions due to their architecture and coding styles.

Enough justification for a rewrite. But which language?

My C and C++ were not good enough to pursue them and gaining proficiency would take too long.

My native languages (perl/python) were not fast enough for the problem.

I was looking for a performant language that facilitated:

- ease of coding
- checking for correctness in development and particularly in peer review
- with an already existing bioinformatics library

The Go programming language: 2 out of 3

Go

golang.org (http://golang.org)

Go is a programming language that blurs the lines between a statically typed, compiled language and a dynamically typed scripting language.

It is the first, but feels like the second.

Design principles (according to one of the lead developers):

Simple	concepts are easy to understand
Orthogonal	concepts mix cleanly, easy to understand and predict what happens
Succinct	no need to predeclare every intention
Safe	misbehavior should be detected

Looked like a good option for implementing our rewrites in the absence of any other reasonable competitor.

The bíogo project

Need for a basic set of tools

There are a number of bioinformatics libraries available in a variety of languages.

- BioPerl
- BioPython
- PyCogent (python)
- BioJava
- SeqAn (C++)
- BioConductor (R)
- *etc.*

None exists in Go, though people are using it for this kind of work.

Since I'm writing a set of tools that interact with sequence and interval data I may as well make the components reusable.

bíogo

Plan to develop a light-weight high-performance library for prototyping new tools:

```
Be easy to learn and use
Clearly document approaches in code and comments
```

- These principles aim to facilitate more widespread publication of source code by making good code easier to write, as discussed in other places:

sciencecodemanifesto.org (http://sciencecodemanifesto.org)

```
Code  All source code written specifically to process data for a published
      paper must be available to the reviewers and readers of the paper.
```

'Publish your computer code: it is good enough.' Nature 2010.

(http://www.nature.com/news/2010/101013/full/467753a.html)

But also that the library should:

```
Not try to do everything
Try to compete with (at least) Java in terms of performance
```

Implementing the library

Initially written with significant influence from BioPerl — just get it working.

Start with basic types — sequences and features and their I/O.

Increasingly adopting Go's coding idioms — increase elegance and readability.

Expand feature set — genomic arrays, interval stores, graphs, external tools *etc.*

Have a reasonably featureful library now (reflects my use needs though).

PALS/PILER implementation with bíogo

Comparison of CPALS and GoPALS

Performance: 71Mbp test genome:

- CPALS — 1m 25s
- GoPALS — 2m 20s / 1m 40s (dual threaded)

GoPALS is more flexible with respect of alignment parameters:

- allows more divergent alignment before catastrophic failure

GoPALS code is clearer and more reusable:

- can now use PALS components as modules within other code

GoPALS has been used in our lab to identify repeat regions in horse, cow, elephant, mouse and human.

Comparison of CPILER and GoPiler

PILER has a function to apply RepeatMasker annotations to PALS-identified features.

- useful in our work to identify novel repeat classes and chimeric repeats

The Go implementation of this functionality beats CPILER's performance by ~100x.

- purely due to choice of algorithm — easier to do things properly using bíogo

Comparison of CPILER GoPiler

GoPILER not yet complete...

- uses a flexible equivalent of PILER's algorithm to build piles
- break piles into clusters using *k* - means
- builds cluster connection graphs

```
Function PILER-DF-NEW(P piles, t integer):
    Let G be a graph where nodes represent repeat instances
    For each pile p in P:
        Let C = clusterise(p)
        For each cluster c in C:
            For each image Q in c:
                Let T = partner(Q)
                Add edge c—cluster-of(T) to G
    Return connected components of G of order ≥ t.
```

Comparison of CPILER and GoPiler

- pile clustering with *k* - means

```
func main() {
    km := ClusterFeatures(feats, 0.15, 5)
    for ci, c := range km.Clusters() {
        fmt.Printf("Cluster %d:\n", ci)
        for _, i := range c {
            fmt.Println(feats[i])
        }
        fmt.Println()
    }

    var within float64
    for _, ss := range km.Within() {
        within += ss
    }
    fmt.Printf("betweenSS / totalSS = %.6f\n", 1-(within/km.Total()))
}
```

Comparison of CPILER and GoPiler

- generates multiple alignments (via interface to MUSCLE)

www.drive5.com/muscle/ (http://www.drive5.com/muscle/)

code.google.com/p/biogo.external/muscle (http://go.pkgdoc.org/code.google.com/p/biogo.external/muscle)

Input data:

```
    fmt.Print(s)
```

Run MUSCLE as external process:

```
    m, err := muscle.Muscle{Quiet: true}.BuildCommand()
    if err != nil {
        panic(err)
    }
    m.Stdin = strings.NewReader(s)
    m.Stdout = &bytes.Buffer{}
    m.Run()
    fmt.Print(m.Stdout)
```

Comparison of CPILER and GoPiler

- and consensus sequences for families...

```
var (
    r = fasta.NewReader(m.Stdout.(io.Reader), &linear.Seq{
        Annotation: seq.Annotation{Alpha: alphabet.DNA},
    })
    ms = &multi.Multi{
        ColumnConsense: seq.DefaultQConsensus,
    }
)
for {
    s, err := r.Read()
    if err != nil {
        break
    }
    ms.SetName(s.Name())
    ms.Add(s)
}
c := ms.Consensus(false)
c.Threshold = 42
c.QFilter = seq.CaseFilter
fmt.Printf("%60a\n", c)
```

Results from test data

Run GoPALS/GoPILER on 71MB test genome.

- reasonably fast
- produces generally good consensus sequences — need validation

```
>Consensus:Family_627 (132 members)
gatcagggatggaaaaatgtccggcccGGCCCGGCCCGGnCCCnGCgCnCCGGCCCGGCC
CGGCCCGGGCCGGGCCGGGAGTGAAAATTAGAGTCCGGCCCAGCCGGCCCAATTTTTCGG
GAAAAAGTTTTTTTTTTCTTTCTGGAATTGGGGCAGTTAGTCCTGCAGAAAATCTCGCCC
GGTCCCAACTCTTTTTAGAAGTAAAAATTTTAAGATTCTGATCAATTTTGGACAAGTTAA
TTTTTGATAAAAACGTTCATATTATCATAATCTCTATGACTCGAAACTGGGCATCGTACC
CCTTGACTCTAAGTGTGCATTTTATGCACATTTTCATTTTCCGTTGAGAATCTCAACTTG
AGATAACCCAACTTGAAAATATAAAATTATAAAAAGTTGATTACGCTGATTCTTTTTGGT
AGTCTCGGATTATCTATATCATTCCTATTAATGTAAATAATTAAATTAACCCTTGCGAGT
AAAATTTACTTTCCACCCCTGCAGGTAAAACGACCTAAAAAATGGTCTCCTGAAAAATCt
TTTTTTCCTGAGCGACTGGTTCTTAAAATATCATCGTATTTGTGTGCAgAAAGAATAAAG
AAAAGATTGACCTTATCGATGGACATGTTGTAATGTCCAAAAAATATCCACATTAACAGA
AAGAAATGTCCATCAATAAACACTTCGAAATGCTCTTTTCTCCCGTAGAGAACATTTTTT
GCTCCTGCAAATTGTATTATACAATGAATTAGGCGACATCAAAATATTCAAAAATTGCTT
TCTTTTTTTTCTCACGGGTAACCTTAGTTTTTAGACGTCGAATTTGAAACAACCGATAAA
AAGTAGCTCGATAAAGTGAGGTCAGATTCTAAAATTGTCATGTTATCCGGTTAATAGAAA
TTCAATATCCTTAAACTTAnTTTCTCAGGCATAAAGnnnnc
```

Real world use

Next step run GoPiler on human GoPALS data.

- waiting for some quiet server time

Experience writing GoPiler with bíogo tools

Easier than I imagined.

- parts of the implementation almost arose by accident
- move load from reasoning about code to reasoning about algorithms
- modular nature of Go's design allows graceful expansion of approaches
- type safety saves lives
- rapid compilation and strict typing makes playing with ideas less onerous

Would recommend.

Summary

bíogo: a new bioinformatics development framework

Easy to learn.

Competitive environment for large scale data analysis.

Needs modules in a number of areas (contributions welcomed).

Don't try this at home.

Thank you

Bioinformatics Group
School of Molecular and Biomedical Science
The University of Adelaide

(mailto:dan.kortschak@adelaide.edu.au)

(http://code.google.com/p/biogo/)

BioInfoSummer 7 December 2012