Matching and Significance Evaluation of combined Sequence/Structure Motifs in RNA using Algebraic Dynamic Programming

Overview

1. What is a sequence/structure motif? Relevant problems
2. Two examples: The Iron Responsive Element (IRE) and the Selenocysteine Insertion Sequence (SECIS)
3. Our approach - a short introduction to Algebraic Dynamic Programming (ADP)
4. Pattern matchers (grammars) for the IRE and the SECIS element
5. re-usable algebras for matching and significance evaluation
6. significance calibration and conclusion

What is a sequence/structure motif?

- Structure
  a) specific secondary structure (specific base pairs)
  b) specific sequence parts (regions with fixed nucleotides)
  –> for example a hairpin loop with a specific sequence
- Function
  - important regulatory functions in the cell
  - post-transcriptional processing of RNA (mRNA-localization, mRNA-degradation, efficiency of translation)

Interesting Problems

a) Can we find the motif in a sequence? (Matching)

b) How often?

c) How meaningful is the search result? (Significance evaluation)

How many hits would we expect in a random sequence with the same length and base composition as the "search" sequence?

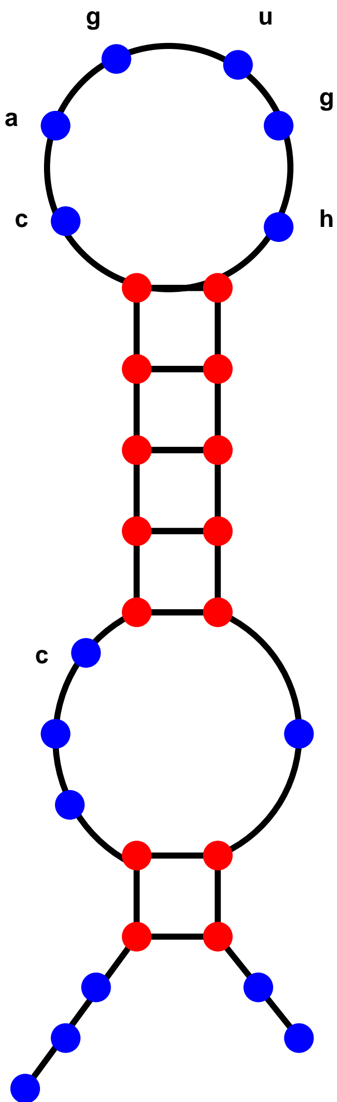
Example:

- counting: 1 hit
- expectation: 0.6 hits

–> Not very meaningful!! Motif description too unspecific!!

The Iron Responsive Element (IRE)

- specific stemloop structure; found in the UTRs of different mRNAs
- RNA dependend regulatory functions mediated by binding of iron-regulatory proteins
- e.g. in the 5'UTR of the ferritin mRNA: Depending on the amount of iron in the cell the IRE effects the translation efficiency of the ferritin-mRNA
- the structure of IREs can differ –> we need a flexible motif description



ire_hairpin:

the concrete loop sequence: hlseq

often: cagugh

ire_stack2:

the minimal number of stacked base pairs: smin

the maximal number of stacked base pairs: smax

often: 4-6 base pairs

ire_loop:

the minimal length of the left region: lmin

the maximal length of the left region: lmax

the minmal length of the right region: rmin

the maximal length of the right region: rmax

either a bulge left consisting just of a ’c’
or
an internal loop: right region length 1
left region length 3 (last nucleotide ’c’)

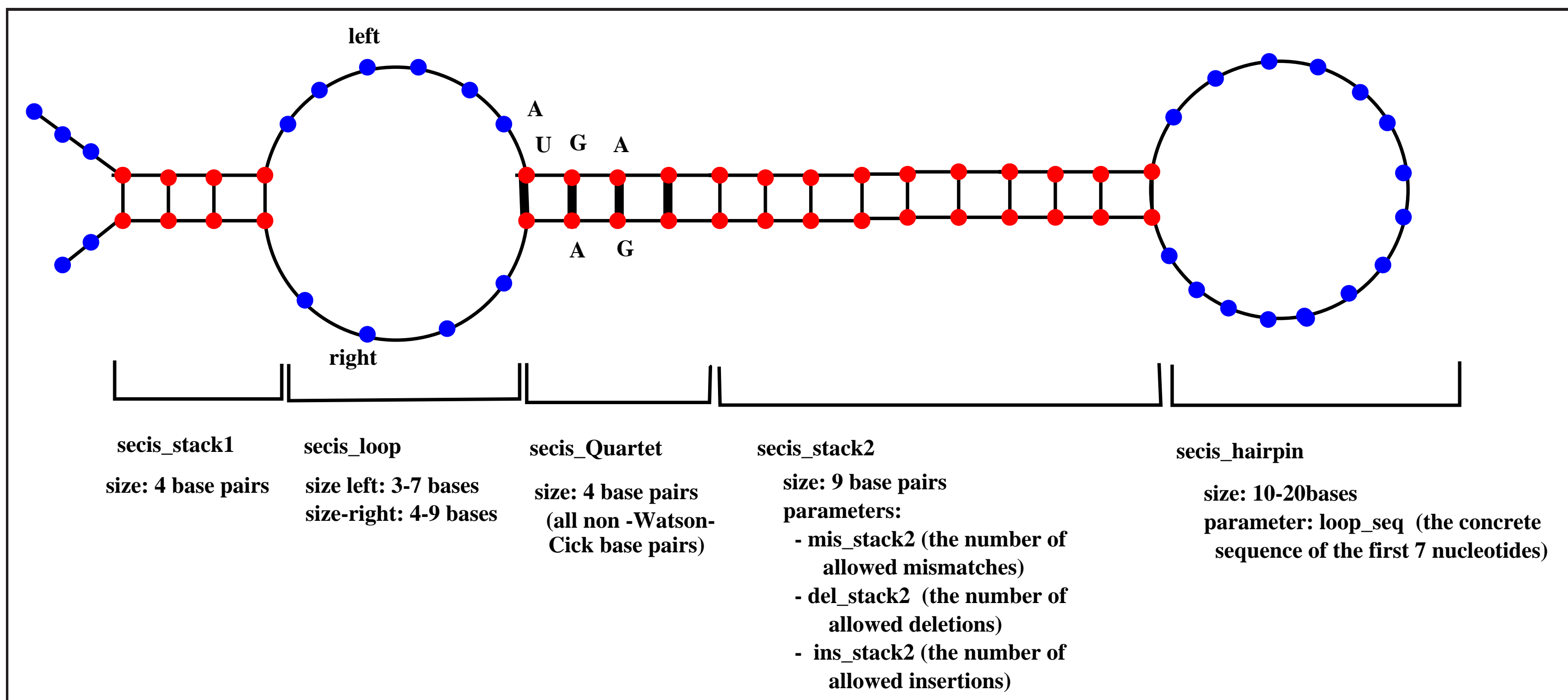
ire_stack1

at least 2 basepairs

The Selenocysteine Insertion Sequence (SECIS)

- specific stemloop structure
- found in 3'UTR of mRNAs which encode for proteins containing the aminoacid selenocysteine
- selenocysteine is encoded by UGA (normally functions as stop codon)
- SECIS-element necessary for incorporation of selenocysteine at an UGA codon

The Selenocysteine Insertion Sequence (SECIS)



our solution

- unambiguous pattern description
- unambiguous (!!!!!) pattern matcher –> no solution is found twice and no semantically equivalent solutions are computed
- we can use the same program to calculate the statistical significance of the search pattern and the pattern matcher itself
- we can compute the expected number of hits on a sequence of given length and basecomposition a priori –> systematic way to calibrate the specificity of a pattern matching algorithm

How?

- the smallest RNA motifs are sequence patterns and basepairs

  –> the significance can be computed from the basecomposition

- larger motifs are build from smaller motifs in an unambiguous fashion
  –> significance of larger motifs can be computed using the significance of smaller motifs

  –> Dynamic Programming can be used to solve this problem

  –> we use **Algebraic Dynamic Programming (ADP)**. This technique allows us an unambiguous description.

Dynamic Programming (DP)

- classical programming technique; very important in computational biology
- applicable if the optimal solution can be computed recursively using the optimal solutions of subproblems
- by using DP a search space of exponential size can be evaluated in polynomial time and space
- example edit distance of two strings
- DP = recursion + tabulation (Re-use of previously computed results for subproblems)

Algebraic Dynamic Programming (ADP)

- programming method to generate systematically DP programs
- DP approach is splitted into a structure recognition phase (grammar) and an evaluation phase (algebra)
- to achieve an ADP program it is necessary to describe the relevant structure with a grammar
- here: we need ADP grammars for the sequence/structure motifs
- the grammars are evaluated by different evaluation algebras
- grammar + evaluation algebra = executable prototype

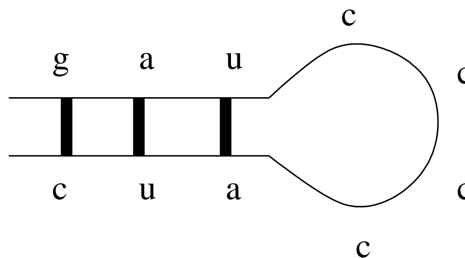
- a more efficient version in `C` can be derived systematically
- at the moment a compiler is developed that generates `C` code automatically

advantages

- implementation is easy since the ADP framework can be used
- not necessary to know the implementation of the used ADP framework
- ADP programs are not so error-prone
- ADP programs are readable
- re-usable structures

ADP-example: RNA structures –> hairpin structures

- hairpin structure: any number of basepairs (here: at least 1) followed by a singlestranded region (here: at least 3 bases)
- example sequence: gaucccccauc



- we use an algebraic representation: a RNA structure consists of a list of structural components
- components: stacking region (SR), singlestranded region (SS), hairpin loop (HL), internal loop (IL), bulge left (BL), bulge right (BR), multiloop (ML)

- term representation of example hairpin:

  SR ’g’ (SR ’a’ (HL ’u’ "cccc" ’a’) ’u’) ’c’

- the tree grammar for hairpin structures:

An executable notation for tree grammars

- we use 3 combinators:
  - <<< : denotes application of a tree constructor (e.g. SR) to its arguments
  - ~~~ : separates the arguments
  - ||| : separates alternative righthand sides
- syntactic restrictions can be associated by a with-clause
- the grammar in the new notation:

```
hairpin_structure =
(sr <<< base ~~~ hairpin_structure ~~~ base) with basepairing |||
(hl <<< base ~~~ (region with minsize 3) ~~~ base) with basepairing
```

- in the Haskell ADP framework the combinators are defined as parser combinators –> we get an executable prototype
- we need an objective function `h` to evaluate the results

  notation `... h` means: apply function `h` to the results
- tabulation: just add the clause `tabulated` to a production
- the grammar now looks like this:

```
hairpin_structure = tabulated((
(sr <<< base ~~~ hairpin_structure ~~~ base) with basepairing |||
(hl <<< base ~~~ (region with minsize 3) ~~~ base) with base-
pairing) ... h)
```

- we need an algebra which gives meaning to sr, hl and the objective function `h`
- example energy minimization: objective function –> minimum

  algebra functions –> energy functions

basepair algebra –> counting the maximal number of basepairs

```
sr lb x rb = x + 1          (x: maximal number of basepairs
hl lb _ rb = 1                  in the included substructure)
h          = maximum
```

- example sequence: gaucccccauc –> result : 3 (maximal number of basepairs)
- 3 different hairpin structures:
  - basepair gc and a loop of length 8 –> score 1
  - basepairs gc and au and a loop of length 6 –> score 2
  - basepairs gc, au and ua and a loop of length 4 –> score 3

counting algebra –> counting the number of different hairpin structures

```
sr lb x rb = x                (x: number of hairpin structures
hl lb _ rb = 1                  in the included substructure)
h  xs      = sum xs
```

- example sequence: gauccccauc –> result : 3 (number of possible hairpin structures)

patterns	**meaning**
`rp lb x rb`	required pair of specific bases `lb` and `rb`
`unp lb t rb`	(unpaired) The bases `lb` and `rb` cannot form a feasible base pair.
`loop us x _`	Internal loop: left singlestranded region is a sequence motif. right region is arbitrary.
`lr _ us`	(left region) arbitrary region at the $5'$ end and a specific sequence motif at the $3'$ end (`us`).
`rr us _`	(right region) a specific sequence motif at the $5'$ end (`us`) and arbitrary region at the $3'$ end.
`skip_left _ x`	skips one base at the $5'$ side
`skip_right x _`	skips one base at the $3'$ side

The unambiguous IRE grammar

```
IRE alg lmin lmax rmin rmax smin smax hlseq inp = axiom (p lcomps) where
  (str,ss,hl,sr,lr,skip_left,skip_right,loop,h) = alg
  lcomps         = tabulated (
    str <<<  (skip_left <<<  base -~~ p lcomps ||| p rcomps      ... h))
  rcomps         = tabulated (
    skip_right <<<  p rcomps ~~- base ||| p ire_stack1           ... h)
  ire_stack1     = tabulated (
    (sr <<< base -~~ ire_stack1b ~~- base)  with` basepairing)
  ire_stack1b    = (sr <<< base -~~ p ire_loop  ~~- base) `with` basepairin
  usinglestrand = ss <<< uregion
  ire_loop       = tabulated ((loop <<< (lr <<< usinglestrand  ~~-

    (fbase "C")) ~!+~ p ire_stack  ~!!+~  usinglestrand)           ... h)
  stackscheme r = (sr <<< base -~~ r ~~- base) `with` basepairing
  ire_stack      = tabulated ((upto (smax-smin) stackscheme
    (rep (smin-1) stackscheme ire_hairpin))                      ... h)
  ire_hairpin    = (hl <<< base -~~ (iupac hlseq) ~~- base) `with` basepair
```

The SECIS grammar

```
SECIS alg mis_stack2 del_stack2 ins_stack2 loop_seq inp = axiom (p lcomps)
    where
  (str,ss,hl,sr,st,unp,lr,rr,skip_left,skip_right,loop,h) = alg
  lcomps           = tabulated (
         str <<< ((skip_left <<<  base -~~ p lcomps ||| p rcomps) ... h))
  rcomps           = tabulated (
         (skip_right <<< p rcomps ~~- base ||| secis_stack1)      ... h)
  stackscheme r    = (sr <<< base -~~ r ~~- base)  `with` basepairing
  mismatch r       = (unp <<< base -~~ r ~~- base) `with` mispairing
  deletion r       = skip_left  <<< base -~~ r
  insertion r      = skip_right <<< r ~~- base
  secis_stack1     = tabulated (
   (rep 4 stackscheme 0 mismatch 0 deletion 0 insertion (p secis_loop)) ..
  usinglestrand    = ss <<< uregion
  secis_loop       = tabulated (
           (loop <<< (lr <<< usinglestrand   ~~- (fbase "A"))   ~!+~
           (secis_quartet)  ~!!+~  usinglestrand)                   ... h)
```

```
  secis_quartet    =
             (rp <<< fbase "U" -~~ secis_quartet2 ~~- fbase "U"  |||
              rp <<< fbase "U" -~~ secis_quartet2 ~~- fbase "C")     ... h
  secis_quartet2   = st <<< fbase "G" -~~ secis_quartet3 ~~- fba-
se "A"
  secis_quartet3   = st <<< fbase "A" -~~ secis_quartet4 ~~- fba-
se "G"
  secis_quartet4   = (unp <<< base -~~ secis_stack2 ~~- base)
                                                   `with` mispairing
  secis_stack2     = tabulated (
    (rep 8 stackscheme mis_stack2 mismatch del_stack2 deletion
                      ins_stack2 insertion secis_stack2end)       ... h)
  secis_stack2end  = (sr <<< base -~~ secis_hairpin ~~- base)
                                                   `with` basepairing
  secis_hairpin    =
     (((hl <<< base -~~ (hairpin `with` minloopsize 10 `with` max-
loopsize 20)
                    ~~- base)  `with` basepairing)               ... h)
              where
     hairpin  = rr <<< (iupac loop_seq) ~!++~ usinglestrand
```

Efficiency of the grammars

- Space efficiency (size of the tables) = $O(n^2)$
- Time complexity:
  - efficiency of ADP grammars = $O(n^{2+w})$
  - w = width of the grammar
  - width = maximum number of ~~~-combinators in a single parser of the grammar
  - no ~~~-combinators in the IRE and in the SECIS grammar
    –> time efficiency: $O(n^2)$

Counting Algebra

The counting algebra computes how often the motif occures in the sequence.

The objective function sums over the structure counts:

```
h []              = []          h  xs             = [sum xs]
```

The evaluation functions count the substructures.

```
skip_left  _ t   = t            skip_right t _   = t
lr _ us          = us           rr us _          = us
ss _             = 1            rp lb t rb       = t
loop us t _      = t            sr lb t rb      = t
unp lb t rb      = t            hl lb us rb     = 1
nwc lb t rb      = t
```

Expectation Algebra

Given the base composition and the length of a sequence the algebra computes the expected number of appearances of the motif based on probabilities!

The algebra does not look inside the concrete sequence !!!!

–> this enables us to calculate the significance a priori

The objective function sums over the probabilities of the structure constituents:

```
h []                 = []          h  xs                = [sum xs]
```

Expectation Algebra (2)

The evaluation functions calculate the probabilities of the substructures:

The probabilities of a string is computed based on the base composition of the sequence.

```
skip_left  _ t  = t          skip_right t _  = t
lr _ us         = us         rr us _         = us
ss _            = 1
rp lb t rb       = t * ubasecomp!lb * ubasecomp!rb
loop us t _      = t * product[ubasecomp!u | u <- us]
sr  lb t rb      = t * pair_prob
unp lb t rb      = t * (1-pair_prob)
hl lb us rb      = pair_prob * product [ubasecomp!u | u <- us]
nwc lb t rb      = t * (1-watcr_pair_prob)
```

Significance calibration

- the number of hits (computed using the counting algebra) is only meaningful, if it lies significantly above the expected number of hits (computed using the expectation algebra)
- calibration of the specificity of the search pattern is possible by choice of the parameters
- e.g. decreasing of the allowed loop sizes or increasing of the desired number of base pairs result in a higher specificity
- e.g. increasing of the allowed number of mismatches or bulges in stacking region leads to a lower specificity
- if the motif description is too unspecific, we get some hits in random sequences

- Using the ADP approach a new non-ambiguous pattern matching algorithm can be designed and tested within a few hours.
- Its efficiency is high enough for systematic testing of hypotheses.
- For screening large data sets, a more efficient version in `C` can be derived systematically. –> soon: just use the Compiler to generate `C` Code
- Grammars for other motifs can be formulated in a similar way
- The same algebras can be used
- new algebras can be implemented (e.g. energy minimization)
- For more details –> paper: Zeitschrift für Physikalische Chemie; C. Meyer and R. Giegerich; Matching and Significance Evaluation of combined Sequence/Structure Motifs in RNA (to appear)