Grammar-based compression of DNA sequences

Neva Cherniavsky
Richard Ladner

Background

- Motivation for DNA compression
  - DNA sequences are large
    - Single sequences are on the order of 100M symbols
    - Take up a lot of space, would like to compress them in an efficient (O(n)) way.
  - DNA structure is crucial in understanding its functionality
    - Use hierarchical modeling to identify interesting portions of the DNA sequence

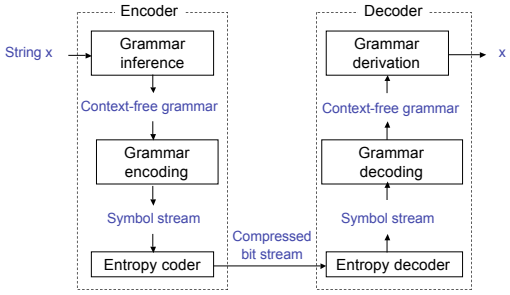
Caveats

- DNA is notoriously difficult to compress
  - Only 4 symbols, so the baseline to beat is 2 bits per symbol
- The most successful method averages only 13% compression (10% w/o outlier)
- Most standard compressors expand it (gzip, bzip2, etc.)

Grammar compression

- Grammar-based compression successful in many domains
- Uses a context-free grammar to represent a string
- The grammar is inferred from the string.
- Language of the grammar consists of only that string.
- If there is structure and repetition in the string then the grammar may be very small compared to the original string.

Overview of Grammar Compression



Our contributions

- Apply grammar-based compression to a new domain: DNA sequences
- Exploit hidden structure of DNA to improve grammar inference
- Optimize symbol stream design and entropy coding for the grammar
- Improve the efficiency of the grammar

Outline

- Introduction
- Grammar inference
  - Sequitur
  - DNA structure
  - DNA Sequitur
- Grammar encoding
- Entropy coding
- Initial experimental results
- Grammar improvement
- Conclusion

7

Grammar inference

- Sequitur: Nevill-Manning and Witten, 1996.
- Elegant, online, linear-time algorithm
- Infers grammar as it reads the string
- The language of the grammar is that string

8

Sequitur Grammar Inference

- Digram Uniqueness:
  - no pair of adjacent symbols (digram) appears more than once in the grammar.
- Rule Utility:
  - Every production rule is used more than once.
- These two principles are maintained as invariants while inferring a grammar for the input string.

9

Sequitur Example (1)

acgtcgacgt

S → a

Digrams

10

Sequitur Example (2)

acgtcgacgt

S → ac

Digrams

ac

11

Sequitur Example (3)

acgtcgacgt

S → acg

Digrams

ac

cg

12

Sequitur Example (4)

acgtcgacgt

S → acgt

Digrams

ac
cg
gt

Sequitur Example (5)

acgtcgacgt

S → acgtc

Digrams

ac
cg
gt
tc

Sequitur Example (6)

acgtcgacgt

S → acgtcg

Digrams

ac
cg
gt
tc

Enforce digram uniqueness.
cg occurs twice.
Create new rule A → cg.

Sequitur Example (7)

acgtcgacgt

S → aAtA
A → cg

Digrams

aA
cg
At
tA

Sequitur Example (8)

acgtcgacgt

S → aAtAa
A → cg

Digrams

aA
cg
At
tA
Aa

Sequitur Example (9)

acgtcgacgt

S → aAtAac
A → cg

Digrams

aA
cg
At
tA
Aa
ac

Sequitur Example (10)

acgtcgacgt

S → aAtAacg
A → cg

Digrams

aA
cg
At
tA
Aa
ac

Enforce digram uniqueness.
cg occurs twice.
Use existing rule A → cg.

19

Sequitur Example (11)

acgtcgacgt

S → aAtAaA
A → cg

Digrams

aA
cg
At
tA
Aa

Enforce digram uniqueness.
aA occurs twice.
Create new rule B → aA.

20

Sequitur Example (12)

acgtcgacgt

S → BtAB
A → cg
B → aA

Digrams

aA
cg
Bt
tA
AB

21

Sequitur Example (13)

acgtcgacgt

S → BtABt
A → cg
B → aA

Digrams

aA
cg
Bt
tA
AB

Enforce digram uniqueness.
Bt occurs twice.
Create new rule C → Bt.

22

Sequitur Example (14)

acgtcgacgt

S → CAC
A → cg
B → aA
C → Bt

Digrams

aA
cg
Bt
CA
AC

Enforce rule utility.
Bt occurs only once.
Remove B → aA.

23

Sequitur Example (15)

acgtcgacgt

S → CAC
A → cg
C → aAt

Digrams

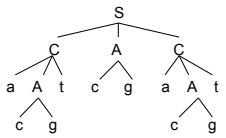
aA
cg
At
CA
AC

24

The Inferred Grammar

acgtcgacgt

S → CAC
A → cg
C → aAt



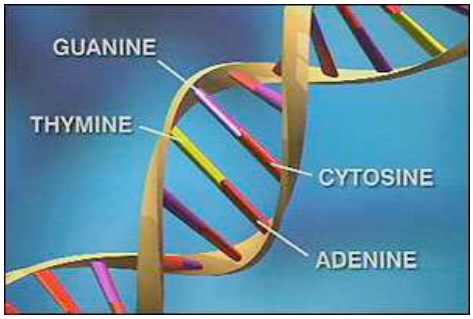
25

Outline

- Introduction
- Grammar inference
  - Sequitur
  - DNA structure
  - DNA Sequitur
- Grammar encoding
- Entropy coding
- Initial experimental results
- Grammar improvement
- Conclusion

26

The Structure of DNA



DNA vs. Arbitrary Sequences

- Only four symbols: a, t, g and c
- Each symbol has a complement form
  - a' = t; t' = a
  - c' = g; g' = c
- Reverse Complements
  - Reverse the string and complement each symbol
  - (cat)' = atg
- Exact repetitions and reverse complement repetitions appear often in DNA

28

Compressing Reverse Complements

- For any DNA sequence x and y
  (xy)' = y'x'
  (x')' = x
- Reverse complementing grammar rules
  A → aBCg
  A' → cC'B't

  C → aD'E
  C' → E'Dt

29

DNA Sequitur vs. Sequitur

- We exploit this hidden structure of DNA
- DNA Sequitur recognizes both exact matches and reverse complements matches
  - For every rule that is created, there is an implicit reverse complement.
    A → ca
    A' → tg
    B → cA
    B' → A'g

30

DNA Sequitur Constraints

- DNA Digram Uniqueness
  - No pair of adjacent symbols can appear more than once anywhere in the grammar. A pair is now defined as any of:

    (XY, XY), (XY, Y'X'), (Y'X', XY), (Y'X', Y'X')
- DNA Rule Utility:
  - Every production rule or its reverse complement is used more than once.

31

DNA Sequitur Example (1)

acgtcgacgt

S → a

Digrams	Implicit Digrams

32

DNA Sequitur Example (2)

acgtcgacgt

S → ac

Digrams	Implicit Digrams
ac	gt

33

DNA Sequitur Example (3)

acgtcgacgt

S → acg

Digrams	Implicit Digrams
ac	gt
cg	cg

34

DNA Sequitur Example (4)

acgtcgacgt

S → acgt

Digrams	Implicit Digrams
ac	gt
cg	cg

Enforce digram uniqueness.
gt occurs twice, first as the complement of ac, and then explicitly.
Create new rule A → ac.

35

DNA Sequitur Example (5)

acgtcgacgt

S → AA'
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'

36

DNA Sequitur Example (6)

acgtcgacgt

S → AA'c
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA

37

DNA Sequitur Example (7)

acgtcgacgt

S → AA'cg
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA
cg	cg

38

DNA Sequitur Example (8)

acgtcgacgt

S → AA'cga
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA
cg	cg
ga	tc

39

DNA Sequitur Example (9)

acgtcgacgt

S → AA'cgac
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA
cg	cg
ga	tc

Enforce digram uniqueness.
ac occurs twice.
Use existing rule A → ac.

40

DNA Sequitur Example (10)

acgtcgacgt

S → AA'cgA
A → ac

Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA
cg	cg
ga	tc

Enforce digram uniqueness.
gA occurs twice, first as the complement of A'c, and then explicitly.
Create new rule B → A'c.

41

DNA Sequitur Example (11)

acgtcgacgt

S → ABB'
A → ac
B → A'c

Digrams	Implicit Digrams
ac	gt
AB	B'A'
A'c	gA
BB'	BB'

42

DNA Sequitur Example (12)

acgtcgacgt

S → ABB'g
A → ac
B → A'c

Digrams	Implicit Digrams
ac	gt
AB	B'A'
A'c	gA
BB'	BB'
B'g	cB

43

DNA Sequitur Example (13)

acgtcgacgt

S → ABB'gt
A → ac
B → A'c

Digrams	Implicit Digrams
ac	gt
AB	B'A'
A'c	gA
BB'	BB'
B'g	cB

Enforce digram uniqueness.
gt occurs twice.
Use existing rule A → ac.

44

DNA Sequitur Example (14)

acgtcgacgt

S → ABB'A'
A → ac
B → A'c

Digrams	Implicit Digrams
ac	gt
AB	B'A'
A'c	gA
BB'	BB'

Enforce digram uniqueness.
AB occurs twice.
Create new rule C → AB.

45

DNA Sequitur Example (15)

acgtcgacgt

S → CC'
A → ac
B → A'c
C → AB

Digrams	Implicit Digrams
ac	gt
AB	B'A'
A'c	gA
CC'	CC'

Enforce rule utility.
B occurs only once.
Remove B → A'c.

46

DNA Sequitur Example (16)

acgtcgacgt

S → CC'
A → ac
C → AA'c

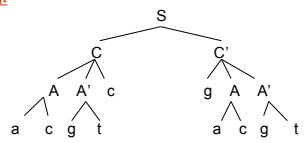
Digrams	Implicit Digrams
ac	gt
AA'	AA'
A'c	gA
CC'	CC'

47

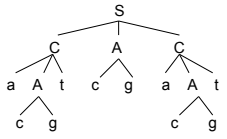
The Inferred Grammar

acgtcgacgt

S → CC'
A → ac
C → AA'c



S → CAC
A → cg
C → aAt

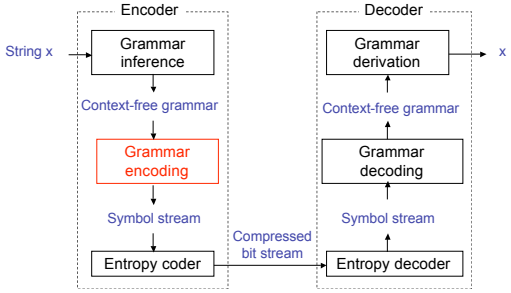


48

Outline

- Introduction
- Grammar inference
- Grammar encoding
  - Simple method
  - Marker and LZ77-style methods
- Entropy coding
- Initial experimental results
- Grammar improvement
- Conclusion

Overview of Grammar Compression



Grammar encoding

- We implemented three versions of grammar encoding
  - A simple version, mainly to use as a baseline
  - A marker method version, first proposed by Nevill-Manning and Witten, that we modified for DNA Sequitur (explained next)
  - A version based on LZ77 ideas, which works the best, but is more complicated

Basic Encoding a Grammar

Grammar

S → CC'
A → ac
C → AA'c

Send right hand sides of rules, separated by #

Grammar Code CC'#ac#AA'c

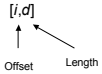
|Grammar Code| = $(s + r - 1)\lceil \log_2(r + a) \rceil$

r = number of rules
s = sum of right hand sides
a = number in original symbol alphabet

Arithmetically encode to achieve maximal compression

Encoding of the Grammar

- Nevill-Manning and Witten suggest a more efficient encoding of the grammar that uses markers as pointers
  - Send the right hand side of the S production.
  - The first time a nonterminal is sent, a marker symbol followed by its right hand side is transmitted instead.
  - The second time a nonterminal is sent as a tuple



A new production rule is then added to a dictionary.

  - Subsequently, the nonterminal is represented by the index of the production rule.

Marker method

Grammar

S → CC'
A → ac
C → AA'c

Use # as the marker symbol. Add a complement place to the tuple.

Grammar Code

Marker method (1)

Grammar
S → CC'
A → ac
C → AA'c

Send the first symbol of S. This is a rule, so send marker followed by rhs.

Grammar Code #

Marker method (2)

Grammar
S → CC'
A → ac
C → AA'c

Send the first symbol of C. This is a rule, so send marker followed by rhs.

Grammar Code ##

Marker method (3)

Grammar
S → CC'
A → ac
C → AA'c

Send rhs of A.

Grammar Code ##ac

Marker method (4)

Grammar
S → CC'
A → ac
C → AA'c

Send second symbol of C. This is the second appearance of A, so send tuple.

Grammar Code ##ac[1,2,1]

Marker method (5)

Grammar
S → CC'
A → ac
C → AA'c

Send last symbol of C.

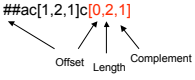
Grammar Code ##ac[1,2,1]c

Marker method (6)

Grammar
S → CC'
A → ac
C → AA'c

Send second symbol of S. This is the second appearance of C, so send tuple.

Grammar Code ##ac[1,2,1]c[0,2,1]



Marker method summary

Grammar
S → CC'
A → ac
C → AA'c

Grammar Code ##ac[1,2,1]c[0,2,1]

Any subsequent appearance of A, A', C, or C' would be sent as the nonterminal index (1, 1', 2, or 2').

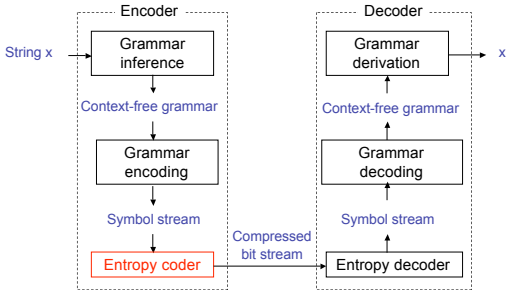
Any new rule formed would have marker offset 0.

Arithmetically encode to achieve maximal compression.

Outline

- Introduction
- Grammar inference
- Grammar encoding
- Entropy coding
- Initial experimental results
- Grammar improvement
- Conclusion

Overview of Grammar Compression



Entropy Coding

- The symbol stream has many distinct symbols.
- Arithmetic coding is a general technique that takes advantage of the statistics of the stream to achieve close to entropy performance.
  - Adapts to statistics
  - Uses context, if needed
- We treat the symbol stream as a character stream in a large alphabet and build a custom arithmetic coder for it.

Custom arithmetic encoder

- Two streams
  - Arithmetically encode terminals and nonterminals
  - Use a fixed prefix code for the tuples
  - Use an escape symbol to switch between streams

….ac3' 5' [6 g 7' % …

….001011100010101…

Custom arithmetic encoder

- Two streams
  - Arithmetically encode terminals and nonterminals
  - Use a fixed prefix code for the tuples
  - Use an escape symbol to switch between streams

….ac3' 5' [6 g 7' % …

↓

….001011100010101…

Custom arithmetic encoder

- Two streams
  - Arithmetically encode terminals and nonterminals
  - Use a fixed prefix code for the tuples
  - Use an escape symbol to switch between streams

….ac3’ 5’ [6 g 7’ % …

….001011100010101…

67

Custom arithmetic encoder

- Two streams
  - Arithmetically encode terminals and nonterminals
  - Use a fixed prefix code for the tuples
  - Use an escape symbol to switch between streams

….ac3’ 5’ [6 g 7’ % …

….001011100010101…

68

Custom arithmetic encoder

- Two streams
  - Arithmetically encode terminals and nonterminals
  - Use a fixed prefix code for the tuples
  - Use an escape symbol to switch between streams

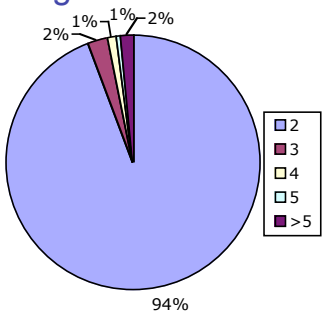
….ac3’ 5’ [6 g 7’ % …

….001011100010101…

- Escape symbols differ depending on the length - usually 2

69

Tuple Lengths in Marker Method



70

Outline

- Introduction
- Grammar inference
- Grammar encoding
- Entropy coding
- Initial experimental results
- Grammar improvement
- Conclusion

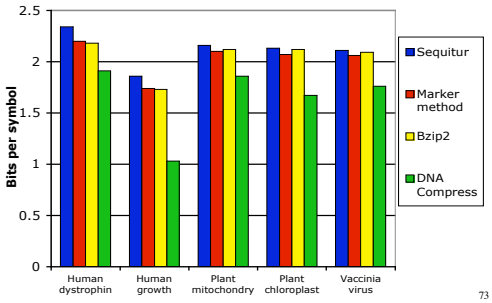
71

Grammar inference results: Sequitur vs. DNA Sequitur

Sequence length	Productions		Length RHS		Longest Repeat		Max repeats	
	Seq	DNA Seq	Seq	DNA Seq	Seq	DNA Seq	Seq	DNA Seq
38,770	1,308	1,163	10,413	10,099	16	19	104	138
66,495	2,288	2,200	13,270	13,111	225	346	113	174
100,314	2,795	2,616	23,373	22,910	114	115	183	195
121,024	3,077	2,985	27,153	26,843	21	28	185	250
191,737	4,480	4,373	41,303	40,654	560	560	238	287

72

Basic Compression Results



73

Grammar Improvement

- Basic idea: remove inefficiencies in the grammar
- Two approaches:
  - Kieffer-Yang: sound theory, no practical evaluation until now
  - Cost measure: our grammar efficiency yardstick

74

Kieffer-Yang Improvement

- Kieffer and Yang developed a theoretical framework for studying these types of grammars in 2000.
  - KY is universal; it achieves entropy in the limit
- Add to Sequitur Reduction Rule 5:

	S → AB		S → AA	
	A → CD		A → CD	
	B → *a*E	⇒	B → *a*E	Adding this constraint makes Sequitur universal.
<C> = *ac*	C → *ac*		C → *ac*	
<D> = *gt*	D → *gt*		D → *gt*	
<E> = *cgt*	E → *c*D		E → *c*D	

<A> = <B> = *acgt*

75

Implementation of Kieffer-Yang

- Add a string table that contains the derivation <A> of rule A
- Before a new rule is formed, check table
- Leads to smaller grammar sizes but can *increase* entropy:

S → …E't…A…aE		S → …B…A…B'
A → CD		A → CD
C → *ac*	⇒	B → E't
D → *gt*		C → *ab*
E → *c*D		D → *cd*
		E → *b*D

<A> = <B> = *acgt*

Digrams	Implicit Digrams
…	…
E't	aE
…	…

76

Implementation of Kieffer-Yang

- Add a string table that contains the derivation <A> of rule A
- Before a new rule is formed, check table
- Leads to smaller grammar sizes but can *increase* entropy:

S → …E't…A…aE		~~S → …B…A…B'~~
A → CD		~~A → CD~~
C → *ab*	⇒	~~B → E't~~
D → *cd*		~~C → *ab*~~
E → *b*D		~~D → *cd*~~
		~~E → *b*D~~

<A> = <B> = *abcd*

77

Implementation of Kieffer-Yang

- Add a string table that contains the derivation <A> of rule A
- Before a new rule is formed, check table
- Leads to smaller grammar sizes but can *increase* entropy:

S → …E't…A…aE		S → …A'…A…A
A → CD		A → CD
C → *ab*	⇒	C → *ab*
D → *cd*		D → *cd*
E → *b*D		E → *b*D

<A> = <B> = *abcd*

78

Our cost measure

- Sometimes it is more expensive to code the rule than send the right-hand side as is

  $I(s)$ = information of symbol s
  $N(s)$ = # times s appears in the symbol stream
  T = total number of symbols in the stream

  Let $A \rightarrow x$, $x = x_1 \ldots x_n$. Then:
  $I(s) = -\log_2(N(s)/T)$
  $I(x) = \sum I(x_i)$

79

Cost measure continued

- The cost of replacing a rule is the number of time the rule appears times its information:

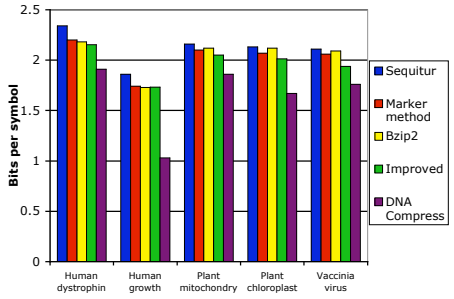
  $R(A) = N(A)I(x) + N(A')I(x')$

- The cost of using a rule is the cost of its right hand side, plus its fixed code cost, plus the cost of its subsequent appearances:

  $U(A) = I(x) + C + (N(A) - 2)I(A) + N(A')I(A')$

80

Total Results



81

Conclusions

- We've taken a good general compression technique and tried it on a new, important domain
  - We've created a new algorithm that exploits the structure of DNA to infer better grammars
  - We've optimized each step of the grammar compression process and improved the final grammar

82

Future Work

- Bottom line: DNA compression is hard.
  - Best method for DNA compression not great (close to simple arithmetic coding)
  - Uses inexact matches (i.e., repeat + insert a character, replace a character, etc)
- Edit grammars could be a good way of capturing exact and inexact matches
  - Grammar rules include A → X[editop]
  - [editop] is insertion, replacement, or deletion

83

Questions?

84