Identifying Hierarchical Structure in Sequences: A linear time algorithm

Craig G. Nevill-Manning · Ian H. Witten

1/25/2023

follow the presentation

https://mangoiv.github.io/sequi-slides - follow my presentation (at home)

structure

0. structure
1. motivation
2. algorithm
    1. concept
    2. implementation
    3. complexity
3. evaluation
    1. showcase
    2. comparison to other compression algorithms
    3. summary

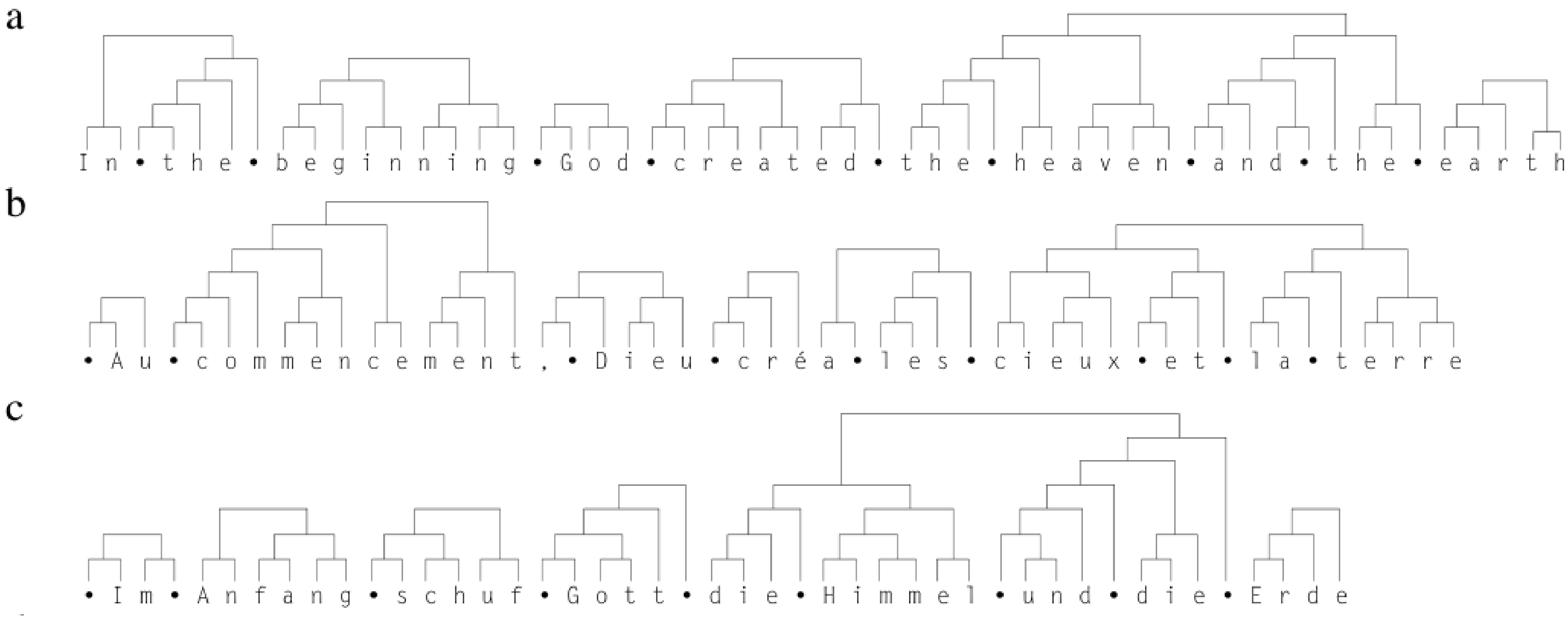
[1](3) - ..ab + A -> [illegible]

motivation

> what goal does `sequitur` pursue?

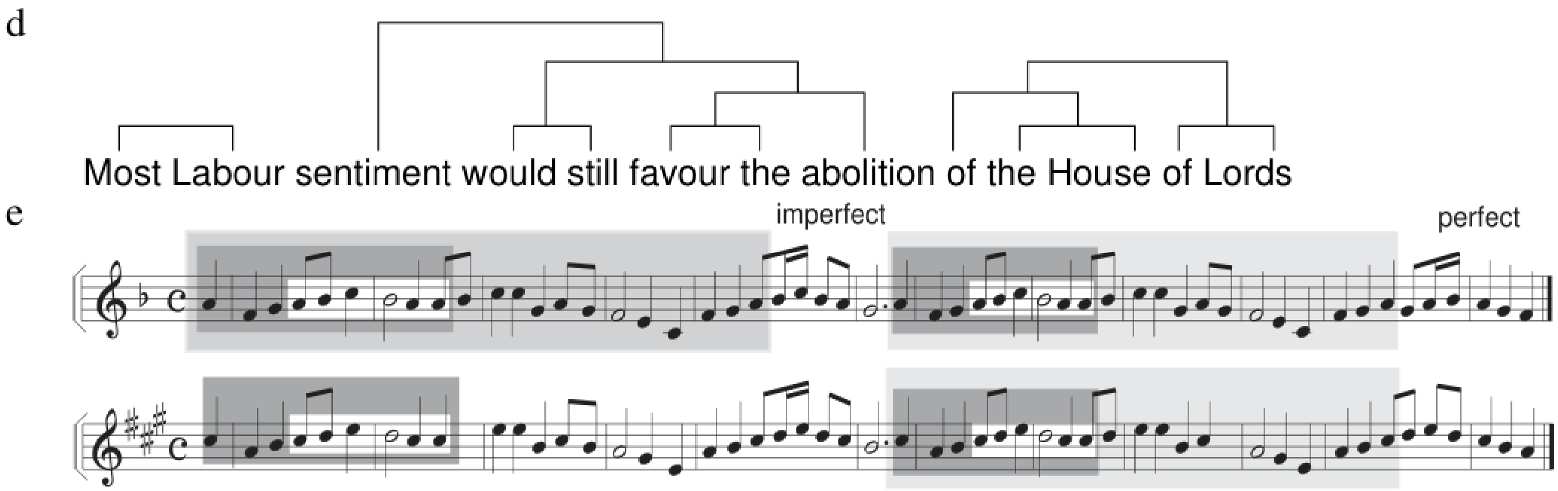
- infer structure from a *stream* of symbols
- use this structure to compress the stream in a continuous/incremental manner
- do it *fast* and *lossless*

motivation - bible



inferred structure for the same sentences in the (a) English, (b) French and (c) German bible

motivation - corpus, chorales



inferred structure for (d) a sentence in the oslo-bergen corpus (e) chorales by J.S. Bach

algorithm

> the `sequitur` algorithm

algorithm - concept: digram uniqueness

> each digram appears at most once in the grammar

observed: `abcdbc` (`a[bc]d[bc]`)

```
S → abcdbc  -- `bc` appears twice

S → aAdA    -- ensure digram uniqueness
A → bc
```

observed: `abcdbcabcdbcbc` (`[[a[bc]][d[bc]][[a[bc]][d[bc]]][bc]`)

```
S → AAbc    -- `bc` appers in `S` and `B`
A → aBdB
B → bc

S → AAB     -- ensure digram uniqueness
A → aBdB
B → bc
```

algorithm - concept: rule utility

> if a rule is only used once, we resubstitute to save space and extend the length of the rule

observed: `abcabc` (`[abc][abc]`)

```
S → AcAc     -- `Ac` appears twice
A → ab

S → BB      -- digram uniqueness
A → ab      -- `A` only appears once
B → Ac      -- namely here

S → BB
B → abc     -- resubstitute
```

observed: `abcdbcabcd` (`[a[bc]d][bc][a[bc]d]`)

```
S → CAC
A → bc
B → aA       -- `B` is used only once
C → Bd       -- namely here

S → CAC
A → bc
C → aAd      -- resubstitute `aA` for `B
```

algorithm - full example

observed: `abcdbcabc` (`[a[bc]]d[bc][a[bc]]`)

```
S → BdAB
A → bc
B → aA
```

observe `d`

observed: `abcdbcabcd` (`[a[bc]d][bc][a[bc]d]`)

```
S → BdABd    -- append `d`, `Bd` appears twice
A → bc
B → aA

A → CAC      -- digram uniqueness
A → bc
B → aA       -- `B` only appears once
C → Bd

S → CAC
A → bc
C → aAd      -- rule utility
```

[1](3) - ..ab + A -> [illegible]

implementation

> running `sequitur` on a machine

[1](3) - ..ab + A ->

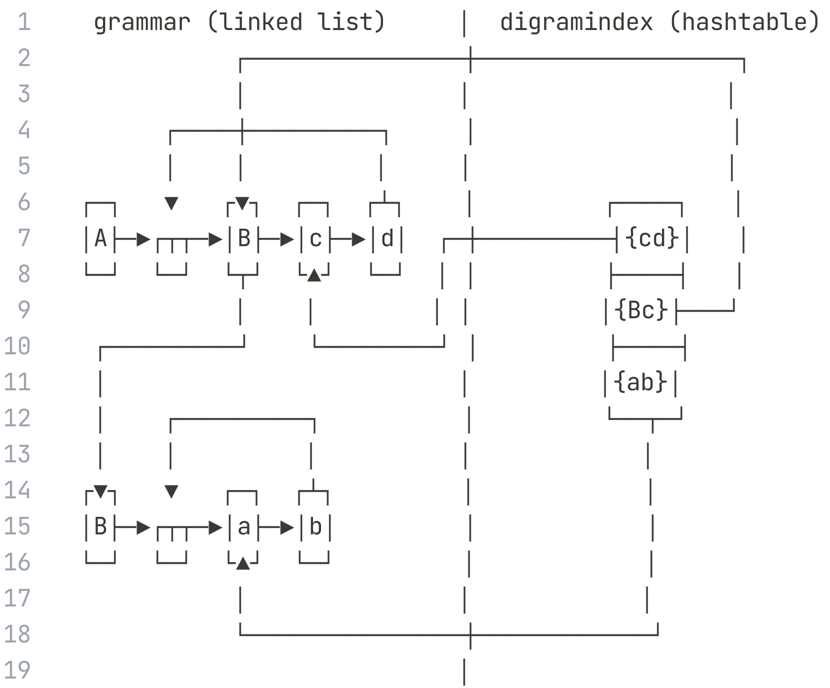
implementation - constraints

- *append* to `S`
  - we need fast `snoc`[1]
- *use* a rule
  - substitute a non-terminal by any digram (this *shortens* the rule)
- *create* a rule
  - non-terminal on LHS
  - digram on RHS
- *delete* a rule
  - move RHS to replace a non-terminal
  - delete LHS

[1] (3) - ..ab + A -> [illegible]

implementation - datastructures

grammar and digramindex



[1](3) - ..ab + A ->

implementation - example

observed: `abcdbc` (`a[bc]d[bc]`)

```
S → abcdbc  { ab, bc, cd, db }

S → abcdbc  { ab, bc, cd, db }  -- create rule that produces `bc`
A → bc

S → aAdbc   { bc, db, aA, Ad }  -- update `ab`, `cd`;  update `S` rule
A → bc

S → aAdA    { bc, dA, aA, Ad }  -- update `db`; update `S` rule
A → bc
```

[1](3) - ..ab + A -> ...

implementation - complexity

```
upon observation append symbol to `S` - Rule                   (1)

entry in grammar rule is made:                                 (2)
  if digram is repetition then
    if other occurence is rule then
      replace digram by non-terminal of that rule              (3)
    else
      form new rule                                            (4)
  else
    insert digram into index

digram replaced by a non-terminal:
  if either symbol is non-terminal that only occurs once then
    remove rule substituting its LHS for observed non-terminal (5)
```

- `(1)` performed exactly n times
- `(2)` performed upon link creation
- `(3)`, `(4)`, `(5)` with savings $1, 0, 1$[1] respectively

[1] (3) - ..ab + A -> [illegible]

implementation - complexity

- n - size of input
- o - size of final grammar
- r - number of rules in final grammar
- $a_1 - a_5$ actions `(1)`-`(5)` respectively
- $n - o = a_3 + a_5$ *- savings are the amount of times a_3, a_5 are performed*
- $r = a_4 - a_5 \equiv a_4 = r + a_5$ *- formed- minus deleted rules, a_5 bounded by 1.*
- $r < o \equiv r - o < 0$ *- less rules than symbols (per rule ≥ 2 symbols)*
- $a_5 = n - o - a_3 < n$ *- see first rule*

$$\sum_{i=1}^{5} a_i = n + a_2 + (n - o) + (r + a_5) \tag{1}$$

$$= n + a_2 + n + (r - o) + a_5 \tag{2}$$

$$< 3n + a_2 \tag{3}$$

[1] (3) - ..ab + A -> ...

implementation - complexity

- a_2: check for duplicate digrams
- with occupancy $< 80\%$ lookups are $\mathcal{O}(1)$
- hashtable size smaller than grammar (which itself is linearly bounded by the input)
- a_2 can only be executed, when either of $a_1, a_3 - a_5$ are run (bounded by $\mathcal{O}(n)$)

$$\implies a_2 \in \mathcal{O}(n)$$

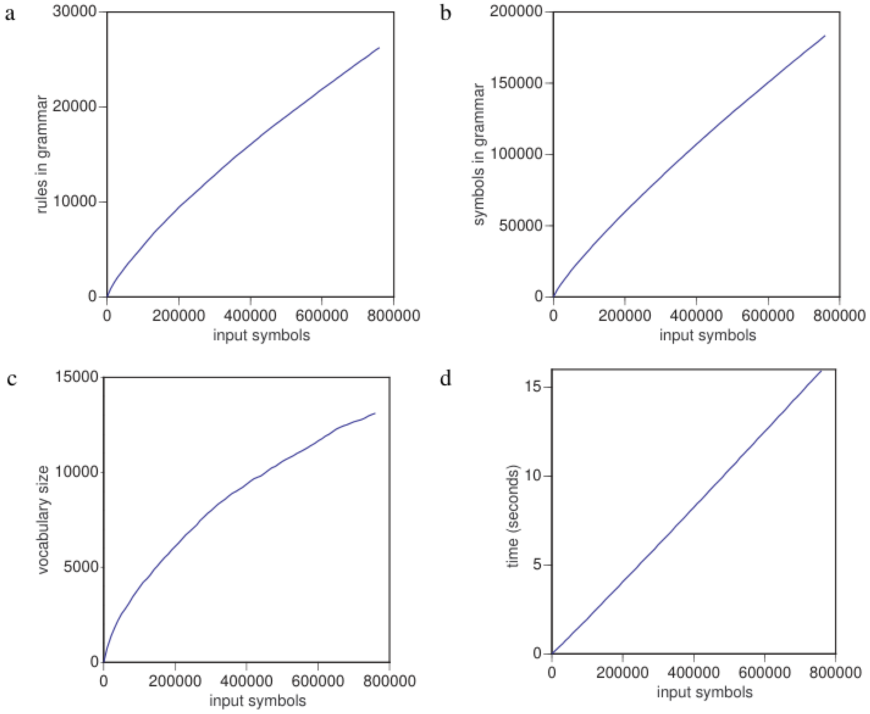
$$\implies$$ `sequitur` runs in $\mathcal{O}(n)$

but

- in theory the hashing and hence addressing will be $\mathcal{O}(\log n)$, remains stable up to 10^{19} symbols on 64bit archs (10 Exabytes if 1 Byte per symbol is assumed)
- `sequitur` is also linear in memory

[1](3) - ..ab + A -> [illegible]

implementation - complexity



behaviour on English text; rules-symbols (a); grammar-symbols (b); vocabulary-symbols (c); time-symbols (d)

evaluation

> how does `sequitur` perform?

evaluation - showcase

http://sequitur.info - JS-implementation by C. Nevill-Manning

[1](3) - ..ab + A ->

evaluation - comparison

> `sequitur` [...] outperfoms other schemes that achieve compression by factoring out repetition, and approaches performance of schemes that compress based on probabilistic predictions

- performance in compressing macromolecular sequences is better than `PPM`
- generally (in most cases) performs better than other generic compression algorithms like `gzip`
- compresses the bible (King James version) best
- *linear in time* (cf. `Mk10`, $\mathcal{O}(n^2)$, Wolff, 1975)

[1](3) - ..ab + A -> [illegible]

evaluation - comparison

but

- linear in space
  - split input; merge grammars
  - $\mathcal{O}(\log n)$ memory
  - sacrifices digram uniqueness
- issues with hashtable
  - resizing (to maintain amortized[1] $\mathcal{O}(1)$ `lookup` and `insert`) is costly

[1] [illegible]

evaluation - summary

Use two simple rules:

- **digram uniqueness**
- **rule utility**

to achieve algorithm that compresses...

- in $\mathcal{O}(n)$ space and **time**
- **losslessly**

which can be implemented...

- by the use of **doubly linked lists**
- and **hash tables**

[1] (3) - ..ab + A -> [illegible]

Thank you for your attention

> questions welcome