LKR Project 3

Table des matières

1 Structural Information Theory Grammar

By studying the Structural Information Theory, we came up with the following DCG grammar for verifying it :

```
1  exp --> term.
2  exp --> term, [+], exp.
3
4  term --> factor.
5  term --> digit, [*], exp.
6
7  factor --> elem.
8  factor --> ['S'], ['['], sym, [']']. %S[(A)(B),(C)]
9  factor --> ['<'], alt, ['>'], ['/'], ['<'], alt, ['>'
       ]. %<(A)>/<(B)(C)(D)>
10 factor --> ['('], exp, [')'].
11
12 sym --> factor.
13 sym --> factor, [','], factor.
14 sym --> factor, sym.
15
16 alt --> factor.
17 alt --> factor, alt.
18
19 elem --> char.
20 elem --> char, elem.
21 char --> [D], {is_alnum(D)}.
22 digit --> [D], {is_alnum(D)}.
23 digit --> [D], {number(D)}.
```

The grammar in itself is not very complex, but we didn't use it to factorize, it was just used as an output verification step.

Instead, we implemented several algorithms to match the patterns and then, we added a step to consume characters and generate the factorization. This will be explained later on.

2 Most Frequent Consecutive Repeated Substring

The iteration rule from standard information theory aims to factorize the consecutive repeated pattern of a given string. Here we have some examples :

- AAAAA <= 6*(A)
- AXDEXDEXDEX <= AX + 3*(DEX)
- DEXDEXDEXAX <= 3*(DEX) + AX
- OKDEXDEXDEXAX <= OK + 3*(DEX) + AX

To implement this feature, we applied the Longest Repeated Substring Algorithm found in the following link plus an index checking to search only for **consecutive** patterns.

Theses are the steps of this algorithm :

1. First we have to build a suffix array.

   We ’ll use "abracadabra for this example".

   index : word

   0 : abraabraabra

   1 : braabraabra

   2 : raabraabra

   3 : aabraabra

   4 : abraabra

   5 : braabra

   6 : raabra

   7 : aabra

   8 : abra

   9 : bra

   10 : ra

   11 : a

2. Then we iteration through the array to compare all suffixes to each other. We take a suffixes and we check if it’s a prefixing the others.

   prefix(abra, abraabraabra) is true

   prefix(abra, abraabra) is true

   prefix(abra, abra) is true

   The rest is false.

3. But here we want to find only consecutives patterns :

   "abraabra" is containing a consecutive pattern "abra".

   "abraxabra" is a not containing any consecutives patterns.

   "abrabra" contains two "abra" pattern but they're overlaping so we want to get rid of it.

   We can validate the consecutivity of a pattern thanks to a index checking :

```
1  is_continuous(_, SubIdx, SuffIdx, _, _) :-
2  SubIdx == SuffIdx,
3  !.
4  is_continuous(SubLen, SubIdx, SuffIdx, Sub,
       Suff) :-
5  drop(SubLen, Suff, NewSuff),
6  prefix(Sub, NewSuff),
7  NewSuffIdx is SuffIdx + SubLen,
8  is_continuous(SubLen, SubIdx, NewSuffIdx,
       Sub, NewSuff).
```

Comparing "abra" of index 8 and "abraabra", of index 4, first check if abra is prefixing abraabra, prefix(abra, abraabra) = true, then drop as many letters than the length of "abra" in the suffix, drop(4, "abraabra", R). R = abra and increment his index, 4 + 4 = 8. The pattern is consecutive if the incermented index reach "abra"s index. Which is 8 so its valid.
Let's see for an overlaping example "abrabra" :
"abra" is prefixing "abrabra", "abrabra"'s index is 0 and "abra"'s index is 3. When we drop(4, abrabra, R) R="bra", and the incremented index is 4 which is not equal to 3. We should continue untill prefix(abra, bra) is true but this case stops here.

4. To find the most frequent consecutive pattern, we just count the number of times the routine explained above succeed.

5. But there's still some cases failing, like "cozcozcozw".
   The reason is that we don't find "coz" in the suffix array. The solution is that for each suffix, we have to compare their own prefixes. More precisely, when we iteration on "cozw", we first compare it to all suffix array's elements, then drop the last char and re-compare again :
   prefix(cozw, cozcozw) is false,
   drop_last_char(cozw, R). R=coz.
   We compute again with "coz" and we find 3*(coz).

6. At this point, we can find the most frequent consecutive repeated pattern :
   iteration("abraabra", Cnt, R). Cnt = 2, R ="abra".
   But we want this result :
   iteration("xabraabray", R). R= "x+2*(abra)+y".
   We can reach this result with a simple split string separating the repeated substring and the rest then concatenate between "+" chars.

3 Symmetry matching

To find any symmetry pattern, we extended the suffix array generation algorithm to generate a prefix array as well.

Using this prefix array, we're going to be able to match the suffixes and prefixes the following way to find our patterns :

1. Generate all possible **consecutive** sub-lists of the input array for which we want to find the pattern.
2. Match a prefix and suffix that is equal to one of the possible **consecutive** sub-lists.

If we find a match, this means that we have a symmetry pattern within our given input string.

So, what we do, is we use the `findall/3` function, we can get the whole set of possible symmetry patterns in our given input string and then, we choose the longest one.

After this, we inject the result in a function that will cut it into pieces, to be usable as an output.

For example :

`adacbbcada` yields `S[(bb),(adac)]`.

`cococo` yields `c+S[(o),(oc)]`.

4 Generating the final factorization

To be able to output a result, we probe the input list and in a recursive fashion, we check for each patterns and output them in a result list in the correct order.

Of course, we also need an heuristic to choose which factorization pattern to output over another.

Let's say a symmetry and an iteration can be seen in a string at the same place, for example :

`ooooooo` can be seen as `7*(o)` or `S[(o),(ooo)]`, or `S[(o),(3*(o))]`, or even `S[(o),(S[(o),(oo)])]`.

For this cases, we take each outputs generated by all conflicting patterns, and we compare their lengths stripped of special characters.

For instance, for an iteration, the stripped characters would be : `n*()`.

For the previous example, this would give :

`Len("7*(o)") < Len("S[(o),(ooo)]")` or $1 < 4$ which means that the iteration patterns will be chosen over the symmetry.

After, we factorize this in a recursive fashion and stop when there is no change anymore.

Finally, this will yield a good factorization of our input string.

This also means that we can factorize an expression that is only semi-factorized.

5 Other approaches we tried

5.1 With a DCG Grammar

As explained above, we didn't use Prolog's DCG grammar. This is because we found it too constraining (could be because of our lack of experience).

Even though, we tried using it. For example, here is how we tried doing the iteration :

```
1 s(N) --> a(N).
2 a(0) --> [].
3 a(R) --> [R], [*], [a].
4 a(M) --> [a], a(N), {M is N + 1}.
5
6 eval(X) :-
7    s(_, X, []).
```

This works for this pattern : kkk...k $\rightarrow$ N $*$ (k), where k is only a **single** character.

We couldn't get it to work with strings or other types (atoms, etc). Even, when using it in combination with `phrase/2` like so :

```
1 eval(X, R * (K)) :- phrase(s(R, K), X).
```

5.2 Alternative to our current iteration algorithm

We tried using only Prolog built-ins to achieve it and we came up with this :

```
1  iteration(A, N, L) :-
2    length(A,U),
3    between(1,U,N),
4    length(L,N),
5    maplist(=(_),L),
6    append(L,A).
7
8  iterations(A, N, L, Last) :-
9    findall(L, iteration(A, N, L), S),
10   append(_,[Last], S).
```

This works perfectly fine when presented with an array that contains **only** an iteration pattern. It will fail otherwise.

6 Program Usage

To use our program, you should call the `eval/2`, with an input list.

Examples of usage :

```
1  ?- eval([a, z, e,a, z, e, e,a,b,c,b,a,e,a], X).
2  X = [2, "*", "(", a, z, e, ")", "+", "S", "[", "(", c,
       ")", ",", "(", e, a, b, ")", "]", "+", a]
3
4  % With operators in the input
5  ?- eval([a,b,c,a,b,c,"+",e,d,c,a,a,a,c,d,e], X).
6  X = [2, "*", "(", a, b, c, ")", "+", "S", "[", "(", a,
       ")", ",", "(", e, d, c, a, ")", "]"]
7
8  % Semi-factorized
9  ?- eval(["S", "[", "(", 2, "*", "(", a,b,c,a,b,c, ")",
       ")", ",", "(", e, d, c, ")", "]"], X).
10 X = ["S", "[", "(", 2, "*", "(", 2, "*", "(", a, b, c,
       ")", ")", ")", ",", "(", e, d, c, ")", "]"]
```

7 Shortlog of the teamwork

Here's the Github link.

Here's our git shortlog :

```
Axel Mendoza (13):
      compare suffix
      almost finish iteration, drop not implemented
      Merge branch 'master' of github.com:cpcdoy/
          LKR_project
      almost finish iteration
      Merge branch 'master' of github.com:cpcdoy/
          LKR_project
      finish iteration
      iteration + * pattern
      fix iteration + problem
      fix reverse pattern cut string
      Merge branch 'master' of github.com:cpcdoy/
          LKR_project
      fix wrong matching pattern
      fix reverse on sym
      sit done with iteration and symmetry

Chady Dimachkie (15):
      Add: iterations function (ONLY works when the
          input list contains repetitions)
      Add: symmetry (no indices)
      Add: label_start +     Fix: sublist does not
          return empty arrays anymore
```

```
20      Merge branch 'master' of https://github.com/
            cpcdoy/LKR_project
21      Add: indices computation
22      Merge branch 'master' of https://github.com/
            cpcdoy/LKR_project
23      Add: symmetry string output
24      Add: display symmetry +     Add: get longest
            symmetry
25      Add: correctly displayed symmetry
26      Clean: function names
27      Add: SIT.pl which merges everything
28      Fix: cut_string_sym typo
29      Fix: get_symmetry returns the original array
            when nothing was found
30      Add: README.md
31      Add: grammar.pl
```