Tree Query Language (TQL)

Référence

Doc. v 1.0 - Oct. 2021

Olivier Kraif - Université Grenoble Alpes

1. Introduction

Le langage TQL (Tree Query Language) est un formalisme dédié à l'écriture d'expressions de recherche pour la fouille de corpus arboré.

Il s'apparente à CQL, utilisé dans le Sketch Engine, CQP ou encore Frantext, mais avec une syntaxe légèrement différente (utilisation de chevrons au lieu des crochets) et permet en outre de définir des contraintes au niveau des dépendances syntaxiques. Dans sa conception, TQL cherche à être léger en termes d'écriture et puissant en termes d'expressivité.

Initialement développé pour l'interface ConcQuest en 2005, puis ScienQuest, il a ensuite été intégré à la plateforme Lexicoscope (puis Lexicoscope_2.0).
Les expressions écrites en TQL ont pour vocation de « matcher » (ou correspondre) avec des expressions en corpus et combinant des contraintes lexicales, morphosyntaxiques, syntaxiques, etc.

2. Les tokens

Les tokens correspondent aux mots graphiques, aux signes de ponctuation, aux nombres et à divers symboles. Il correspondent au niveau de segmentation le plus petit d'un fichier analysé en parties du discours (POS, ou Part-Of-Speech) et en dépendances. Bien souvent, en TAL, on représente un texte analysé avec un token par ligne (comme avec Treetagger ou dans le format CONLL).

```
1     Là          là          ADV     _ _                                                          12    advmod
2     ,           ,           PUNCT   _ _                                                          1     punct
3     écartant    écarter     VERB    _ Tense=Pres|VerbForm=Part                                   12    advcl
4     les         le          DET     _ Definite=Def|Number=Plur|PronType=Art                      5     det
5     branches    branche     NOUN    _ Gender=Fem|Number=Plur                                     3     obj
6     ,           ,           PUNCT   _ _                                                          7     punct
7     inquiet     inquiet     ADJ     _ Gender=Masc|Number=Sing                                    5     amod
8     et          et          CCONJ   _ _                                                          9     cc
9     fiévreux    fiévreux    ADJ     _ Gender=Masc                                                7     conj
10    ,           ,           PUNCT   _ _                                                          3     punct
11    il          il          PRON    _ Gender=Masc|Number=Sing|Person=3|PronType=Prs              12    nsubj
12    interrogea  interroger  VERB    _ Mood=Ind|Number=Sing|Person=3|Tense=Past|VerbForm=Fin      _     _
13    les         le          DET     _ Definite=Def|Number=Plur|PronType=Art                      14    det
14    sentiers    sentier     NOUN    _ Gender=Masc|Number=Plur                                    12    obj
15    du          de          ADP     _ _                                                          17    case
16                le          DET     _ Definite=Def|Gender=Masc|Number=Sing|PronType=Art          17    det
17    regard      regard      NOUN    _ Gender=Masc|Number=Sing                                    14    nmod
18    ,           ,           PUNCT   _ _                                                          19    punct
19    semblant    sembler     VERB    _ Tense=Pres|VerbForm=Part                                   12    advcl
20    attendre    attendre    VERB    _ VerbForm=Inf                                               19    xcomp:obj
21    quelqu'un   quelqu'un   PRON    _ Number=Sing|Person=3|PronType=Prs                          20    obj
22    avec        avec        ADP     _ _                                                          23    case
23    impatience  impatience  NOUN    _ Gender=Fem|Number=Sing                                     20    obl:mod
24    .           .           PUNCT   _ _                                                          12    punct
```

Figure 1 : Exemple de phrase analysée avec un token par ligne (format CONLL-U) (tiré de Zola, Les mystères de Marseille, 1884)

Les tokens sont représentés entre chevrons `< >`.

Les propriétés par défaut d'un token sont :

`w`→ la forme fléchie (surface) : en CONLL, cf. colonne 2

`l` → le lemme ou forme canonique : en CONLL, cf. colonne 3

`c` → la catégorie ou POS (p.ex. `VERB`) : en CONLL, cf. colonne 4 (et 5 pour les sous-catégories)

`f` → des traits additionnels (p.ex. `VerbForm=Inf`) : en CONLL, cf. colonne 6

`id` → son identifiant : en CONLL, c'est un numéro qui commence par 1 à chaque début de phrase.

Ainsi :

<> désigne un token quelconque et matche tous les tokens de la phrase

<c=NOUN,#1> désigne un nom quelconque : dans la phrase ci-dessus, l'expression matche les tokens 5, 14, 17, 23. #1 permet d'identifier le token trouvé (de le nommer) pour exprimer des contraintes additionnelles.

<w=les,c=DET,#1> désigne le déterminant *les*. Dans la phrase ci-dessus, l'expression matche les tokens 4, 13.

<l=le,c=DET,#1> désigne le déterminant dont le lemme est *le*. Dans la phrase ci-dessus, l'expression matche les tokens 4, 13 (comme précédemment) ainsi que le token 16, qui n'a pas de forme de surface (il est issu de la décomposition de l'article contracté *du*).

Il est possible d'exprimer des suite de tokens en les concaténant :

<l=le,c=DET,#1><c=NOUN,#2> permet de matcher les tokens 4,5 (*les branches*) et 16,17 (*ø regard*)

<l=le,c=DET,#1><c=NOUN,#2><c=PUNCT,#3> permet de matcher en outre la virgule qui suit.

N.B. : pour l'écriture des formes et des lemmes, on peut utiliser des formes abrégées :

<écartant> est équivalent à <w=écartant>, et <%écarter> est équivalent à <l=écarter>

On peut également rechercher plusieurs séquences dans une même phrase, sans imposer d'ordre entre ces séquences. Pour ce faire on utilise l'opérateur && (qui signifie ET).

Par exemple :

<l=le,c=DET,#1><c=NOUN,#2><c=PUNCT,#3> && <f=Tense=Pres\|VerbForm=Part,#4>

Le token #4 peut matcher aussi bien *écartant* que *semblant*, et #1,#2,#3 peuvent matcher *les branches,* ou *ø regard,* . L'expression peut dès lors matcher les 4 combinaisons possibles pour les instanciations de #1,#2,#3 et #4.

	Tokens 4,5,6,3	Tokens 4,5,6,19	Tokens 16,17,18,3	Tokens 16,17,18,19
#1	*les*	*les*	*ø*	*ø*
#2	*branches*	*branches*	*regard*	*regard*
#3	,	,	,	,
#4	*écartant*	*semblant*	*écartant*	*semblant*

Tableau 1 : 4 instanciations possibles pour l'expression <l=le,c=DET,#1><c=NOUN,#2><c=PUNCT,#3> && <f=Tense=Pres|VerbForm=Part,#4>

3. Relations entre tokens

Le symbole && permet surtout de définir des tokens reliés entre eux non par leur séquence, mais par des relations de dépendance syntaxique (du type sujet-verbe, verbe-objet), indépendamment de leur position dans la phrase.

Les dépendances s'expriment sous la forme de triplet (REL, ID_GOUV, ID_DEP) ou REL est l'étiquette de relation, ID_GOUV et ID_DEP sont les identifiants respectifs du gouverneur et du dépendant.

Les dépendances sont listées après les tokens après le symbole ::

Par exemple :

```
<#1>&&<#2>::(nsubj,2,1)
```

permet de définir deux tokens reliés par une relation de type *nsubj* (pour sujet nominal selon le standard UD). Avec cette requête le verbe correspond à #2 et la tête du sujet à #1. Il peut se situer devant et derrière puisque les tokens sont reliés avec `&&`.

Dans la phrase précédente, cette expression matche avec deux tokens seulement (11, 12) : *il interrogea*

En revanche si on s'intéresse à la relation obj (objet direct) on a plusieurs expressions qui matchent :

```
<#1>&&<#2>::(obj,1,2)
```

écartant + *branches*

interrogea + *sentier*

attendre + *quelqu'un*

Si l'on veut se restreindre à des COD sous forme de pronoms, il faut rajouter une contrainte :

```
<#1>&&<c=PRON,#2>::(obj,1,2)
```

De la sorte, on peut définir un sous-arbre comportant plusieurs relations :

```
<l=inquiet,c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<l=fiévreux,c=ADJ,#3>::(cc,3,2)
(conj,1,3)
```

permet de matcher les tokens 7,8,9 : *inquiet et fiévreux*.

Une requête peut être située à différents niveaux de généralité. Si l'on recherche dans le corpus toutes les occurrences de deux adjectifs coordonnés par *et*, on utilisera donc une requête plus simple et plus générale :

```
<c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<c=ADJ,#3>::(cc,3,2) (conj,1,3)
```

Nota bene :

Dans le Lexicoscope, l'identifiant `#1` marque le noyau de l'expression (choisi par l'utilisateur) : c'est lui qui figure au centre des concordances Kwic (en fonction des paramétrages choisis). L'identifiant `#co` marque, de façon facultative, la position du collocatif, si l'on veut extraire les collocatifs seulement à une position définie (par défaut, les collocatifs sont tous les tokens reliés à n'importe lequel des tokens identifiés par `#1`, `#2`, etc.).

4. Usage des expressions régulières

Les expressions régulières (ou regex) permettent de définir des classes de caractères (p.ex. `[a-z]` ou `\d` pour les chiffres) et des expressions complexes avec des disjonctions (`expr|expr2`) et des opérateurs de répétition tels que * + `{n,m}`

Les valeurs cherchées dans le lexicoscope sont des expressions régulières.

Ainsi, on peut rechercher des formes commençant par des majuscules :

```
<w=[A-Z].*>
```

Des lemmes se terminant par un certain suffixe :

```
<l=.*er,c=VERB>
```

Des lemmes commençant par un certain préfixe :

```
<l=dé.*,c=VERB>
```

Des classes de lemmes :

```
<l=inquiet|anxieux|angoissé,c=ADJ,#1>
```

Il est également possible d'appliquer les opérateurs d'expression régulière au niveau des tokens et non des caractères.

`<c=ADJ,#1><>?<c=ADJ,#3>` : un adjectif suivi d'un token facultatif suivi d'un adjectif

`<c=ADJ,#1><>*<c=ADJ,#3>` : un adjectif suivi d'un nombre de tokens quelconque suivis d'un adjectif

`<c=ADJ,#1><>{1,3}<c=ADJ,#3>` : un adjectif suivi d'un, deux ou trois tokens suivis d'un adjectif

Enfin les relations elles-mêmes peuvent être définies par les regex :

```
<#1>&&<#2>::(.*obj,1,2)
```

Ici on pourra avoir une relation `obj` ou `ccomp:obj` ou `xcomp:obj`.

De ce fait, si l'on a besoin d'exprimer un des métacaractères suivants : `()[]^.*+ ?${}|` , il est nécessaire de le faire précéder par un backslah \ afin qu'il ne soit pas interprété en tant qu'opérateur de regex. C'est pourquoi on ajoute `\` devant le signe `|` dans l'expression ci-dessous :

```
<f=Tense=Pres\|VerbForm=Part,#4>
```

Quand une valeur ne contient qu'un caractère, celui est considéré par défaut comme caractère littéral (le backslash est optionnel dans ce cas) :

`<c=PUNCT,l=?,#1>` : matche le signe `?`

`<c=PUNCT,l=.,#1>` : matche le signe `.`

Symboles et ponctuations

Si l'on doit chercher à exprimer les métacaractères `< > &` et `,` qui jouent un rôle dans la syntaxe TQL, on utilisera les entités (respectivement) `< > & &comma;`

`<c=PUNCT,l=&comma;,#1>` : matche la virgule.

Pour les caractères `( ) { } [ ] .` et `?` qui sont des métacaractères de regex, il faut les faire précéder d'un \. Par exemple :

```
<l=\?,c=SENT,#1>
```

5. Traits positionnels

Il est aussi possible de définir des contraintes sur des traits positionnels inscrits dans les tokens.

Par exemple, on peut rechercher une certaine position dans la phrase :

`<c=VERB,n=1,#1>` : matche tous les verbes en position initiale de la phrase (les positions commencent à 0).

Par ailleurs, pour identifier la position relative dans la phrase, on découpe toutes les phrases en 5 tranches, numérotées de 1 à 5. On peut alors rechercher un verbe dans la première, la deuxième, etc. jusqu'à la dernière tranche.

`<c=VERB,sentPos=5,#1>` : matche un verbe en position finale (tranche 5) de la phrase.

`<c=VERB,sentPos=3,#1>` : matche un verbe en position médiane (tranche 3) de la phrase.

Il est également possible d'interroger les positions relatives par rapport aux textes, aux divisions ou aux paragraphes :

`<c=VERB,textPos=1,#1>` : matche un verbe en position initiale (tranche 1) par rapport au texte

`<c=VERB,divPos=1,#1>` : matche un verbe en position initiale (tranche 1) par rapport à la division

`<c=VERB,paraPos=1,#1>` : matche un verbe en position initiale (tranche 1) par rapport au paragraphe

N.B. : Quand une phrase (ou un paragraphe, ou une division) ne contient qu'un seul token, les traits de position sont fixés à 0.

6. Contraintes additionnelles et négation

Il est possible de formuler une contrainte négative dans la présence d'une relation : auquel cas, on préfixe la relation du symbole `!`.

Par exemple, si l'on veut un verbe, avec un sujet mais pas de complément d'objet (afin d'avoir des occurrences intransitives), on pourra écrire une relation vers un token non instancié (ici 3) :

```
<#1>&&<#2>::(nsubj,1,2)(!obj,1,3)
```

Mais dans certain cas, il ne suffit pas de formuler des contraintes négatives sur la relation.

Par exemple, si l'on veut éliminer les formes négatives, il ne suffit pas d'écrire :

```
<c=VERB,#1>::(!advmod,1,2)
```

car cela supprime toutes les relations adverbiales, et pas seulement la négation.

Et si l'on écrit :

```
<c=VERB,#1>&&<f=Polarity=Neg,#2>::(!advmod,1,2)
```

on introduit une contradiction : à gauche, on recherche un adverbe négatif, et à droite, on interdit la relation avec cet adverbe. On ne trouvera donc que des phrases intégrant une négation sur un autre verbe que celui ciblé par #1.

Pour traiter ce type de contrainte négative, on peut exprimer des contraintes sur le token impliqué dans la relation, sans que ce token n'apparaisse dans la partie gauche.

On écrira donc :

```
<c=VERB,#1>::(!advmod,1,<f=Polarity=Neg>)
```

Ce qui se traduit par : « chercher un verbe quelconque qui n'entretient pas de relation avec un adverbe de négation. »

Notons que dans ce cas, le token `<f=Polarity=Neg>` n'est pas identifié avec #2, car il ne joue pas de rôle en tant que token recherché, mais simplement dans la définition de la relation.

Il est par ailleurs possible d'exprimer des contraintes additionnelles, sous une forme négative ou positive, en utilisant des fonctions. Celles-ci doivent figurer dans le membre de droite (après ::) et peuvent s'ajouter aux contraintes relationnelles avec l'opérateur `&&`.

Par exemple, si l'on veut se limiter à des verbes dont le lemme fait 4 caractères, on peut écrire :

```
<c=VERB,#1>:: length(l(#1))==4
```

Voici une liste non exhaustives des fonctions utilisables :

`w(#n)` -> permet d'obtenir la forme du token `#n`

`l(#n)` -> permet d'obtenir le lemme du token `#n`

`c(#n)` -> permet d'obtenir la catégorie du token `#n`

`f(#n)` -> permet d'obtenir les traits du token `#n`

`simil(#i,#j)` -> permet de calculer un score de similarité distributionnel en #i et #j (un cosinus FastText compris entre 0 et 1)

`levenshtein(w(#i),w(#j))` -> permet de calculer la distance d'édition entre les formes de #1 et #2

`prec(#i,#j)` -> renvoie vrai si `#i` est situé avant `#j` dans la phrase

`hasCorresp(#i)` -> renvoie vrai si le token est resté invariant dans la comparaison de deux versions (quand on travaille sur des versions que l'on compare avec la fonction diff).

`deleted(#i)` -> renvoie vrai si le token a été supprimé dans la version courante.

Ces fonctions peuvent permettre d'élaborer des conditions booléennes (évaluée à Vrai ou Faux) en utilisant les opérateurs de comparaison du langage Perl :

`==`	égal
`!=`	différent
`<=`	inférieur ou égal
`>=`	supérieur ou égal
`eq`	(equal) égal pour des chaines de caractères
`ne`	(not equal) différent pour des chaines de caractères
`=~/regex/`	matche avec la regex
`!~/regex/`	ne matche pas avec la regex

Ces conditions peuvent se combiner avec les opérateurs booléens classiques, ainsi qu'avec les parenthèses :

`not` négation

`and` ET logique

`or` OU logique

Si l'on veut exprimer une condition négative sur un token, c'est donc dans le membre de droite qu'on le fera : supposons que l'on recherche des coordinations avec adjectifs qui ne se terminent pas par le suffixe *-el*

```
<c=ADJ,#1>&&<l=et,c=CCONJ,#3>&&<c=ADJ,#3>::(cc,3,2)(conj,1,3)
&& l(#1)!~/el$/ && l(#2)!~/el$/
```

De la sorte on élimine les occurrences du type « riche et industriel »

(N.B. : pour une écriture plus lisible, il est conseillé de passer à la ligne avant chaque &&)

Autre exemple : supposons que l'on recherche, dans la comparaison de deux versions, les occurrences d'adjectifs ayant été supprimés.

```
<c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<c=ADJ,#3>::(cc,3,2)(conj,1,3)
&& deleted(#1)
```

Si l'on s'intéresse à la position de l'adjectif supprimé, on peut imposer une contrainte supplémentaire sur l'ordre d'apparition :

```
<c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<c=ADJ,#3>::(cc,3,2)(conj,1,3)
&& deleted(#1) && prec(1,3)
```

7. Suggestions par l'exemple

La confection des requêtes est chose difficile : elle requiert la connaissance du formalisme précédemment exposé, ainsi que le jeux d'étiquette morphosyntaxique associé à un corpus donné (il existe de nombreux jeux d'étiquettes différents).

Pour éviter cet écueil, il est recommander de construire ses requête à partir d'exemples.

Pour cela, le Lexicoscope propose le bouton 'Suggérer'. Si vous entrez une expression telle qu'elle apparait en surface dans le corpus, la suggestion construira la (ou les) requêtes correspondantes, en fonction des occurrences trouvées dans le corpus.

Ainsi, il l'on cherche des expressions coordonnant deux adjectifs, on pourra s'appuyer sur un exemple. On trouve, dans le roman de Zola duquel nous avons tirés les précédents exemples, l'expression « lèvres fortes et épaisses ». Si l'on entre « fortes et épaisses » et que l'on demande une suggestion, on obtient la requête :

```
<l=fort,c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<l=épais,c=ADJ,#3>::(cc,3,2) (conj,1,3)
```

A partir de cette requête, il est aisé d'obtenir une expression plus générale en supprimant les lemmes. La requête simplifiée devient :

```
<c=ADJ,#1>&&<l=et,c=CCONJ,#2>&&<c=ADJ,#3>::(cc,3,2) (conj,1,3)
```

On peut même chercher d'autres conjonctions que « et », avec une requête encore plus simple :

```
<c=ADJ,#1>&&<c=CCONJ,#2>&&<c=ADJ,#3>::(cc,3,2) (conj,1,3)
```

Par défaut, les suggestions sont lemmatisées. Mais dans certains cas, on peut être amené à rechercher des flexions particulières. Dans ce cas, on décoche la case « Lemmatiser la suggestion ». Ainsi, si l'on cherche des participes présent, on pourra entrer *écartant* (ou tout autre exemple dont on sait qu'il apparairait dans le corpus), et l'on obtiendra :

```
<w=écartant,l=écarter,c=VERB,f=Tense=Pres\|VerbForm=Part,#1>
```

A partir de cette requête, on peut construire une requête simplifiée permettant de cibler toutes les occurrences de participes présent :

```
<c=VERB,f=Tense=Pres\|VerbForm=Part,#1>
```

On peut ensuite raffiner la requête à loisir. P.ex. si l'on cherche les participes présent qui apparaissent en début de phrase, on pourra ajouter un trait positionnel :

```
<c=VERB,f=Tense=Pres\|VerbForm=Part,n=1,#1>
```

Références

Autres langages de requête

CQL : https://www.sketchengine.eu/documentation/corpus-querying/

Toundra : https://weblicht.sfs.uni-tuebingen.de/Tundra/help