Rapport technique provisoire des algorithmes utilisés pour le parsing d'un corpus de français oral

Julie Beliao, Antoine Liutkus

Rapport technique provisoire des algorithmes utilisés pour le parsing d'un corpus de français oral annoté

Julie Beliao, Antoine Liutkus

March 24, 2012

Abstract

Ce document présente les détails techniques de l'algorithme de synthèse de l'arbre à partir du texte annoté selon le formalisme rhapsodie, ainsi que certains choix d'implémentation. Il accompagne le code source de ces algorithmes qui sera distribué gratuitement en licence LGPL sous peu.

1 Structures de données

1.1 Intro

Nous avons fais le choix de représenter les différents concepts à manipuler sous la forme d'objets. On en identifie deux types principaux : les symboles et les noeuds. Alors que les premiers correspondent aux mots eux-mêmes, les seconds correspondent aux différentes unités macrosyntaxiques et microsyntaxiques typées dans le corpus annoté.

En informatique, un objet est un conteneur symbolique, qui possède sa propre existence et incorpore des informations et des mécanismes en rapport avec une chose tangible du monde réel manipulée dans le programme. C'est le concept central de la programmation orientée objet. En programmation orientée objet, un objet est créé à partir d'un modèle appelé classe, duquel il hérite les comportements et les caractéristiques.

La classe est donc un canevas sur lequel les différents objets sont bâtis. Une classe se définit par ailleurs par ses différents attributs et méthodes. Les attributs sont des variables (comme voiture.vitesseActuelle) et les méthodes sont des fonctions, comme voiture.accelere(). On présentera ci-dessous les classes word et node.

1.2 word

Un symbole est défini comme une instance de la classe word. On entend par symbole tout mot (graphème) ou balise d'annotation rencontré dans l'échantillon annoté.

La classe word permet au mot de ne plus être simplement caractérisé par une chaine de caractères. En fait, un mot est maintenant un objet. Cet objet a plusieurs propriétés, et on peut imaginer rajouter autant de propriétés que souhaitées aux mots. Par exemple, on peut tout à fait imaginer rajouter aux mots des informations prosodiques issues de l'analyse du signal de parole.

1.2.1 propriétés

Dans l'immédiat, un mot a quatres propriétés principales:

- son contenu ou libellé (word.string) : il s'agit de la chaîne de caractère associée au mot, donc soit une balise d'annotation soit un graphème.
- sa position (word.pos) : il s'agit de la position du mot dans le texte.
- son type (word.type) : le type d'un symbole est particulier s'il s'agit d'un symbole de l'annotation syntaxique, et est "standard" si c'est un graphème. Par exemple, le type d'un symbole contenant '[' comme texte est '['. Le type d'un symbole contenant 'le' est 'standard'.
- Ses propriétés diverses. Cet attribut permet d'attacher à un symbole n'importe quel ensemble de propriétés. Cela peut être : speaker pour le locuteur associé au symbole, ou bien d'autres propriétés de type phonologique, etc.

Ainsi, une phrase n'est plus une chaîne de caractères, mais une liste d'objets word. De cette manière, si on veut obtenir le texte de la phrase, il suffit de concaténer les champs *string* des mots la constituant. Si on veut obtenir la position originale des mots de cette phrase, il suffit de même de considérer la séquence de leur champ *pos*. De cette manière, lorsqu'on déplie, on n'a plus une liste de chaînes de caractères correspondant aux différentes phrases, mais une liste de liste de mots. On peut alors tout aussi facilement obtenir les phrases dépliées que les positions correspondantes des mots dans le texte d'origine.

1.2.2 méthodes

La classe word n'a que très peu de méthodes. Sa principale est son constructeur, qui à partir du libellé du symbole et de sa position construit un objet symbole.

1.3 Node

L'arbre est implémenté comme une architecture de noeuds. Cette classe node encapsule toutes les propriétés et méthodes des noeuds de l'arbre complet. Un noeud présente des attributs et des méthodes.

1.3.1 Attributs

Un noeud se caractérise par:

- son *contenu* (node.content): il s'agit d'une liste des mots que ce noeud contient directement
- son *noeud pere* (node.father) est le noeud qui contient le noeud considéré dans l'arbre. Pour la racine, c'est None
- ses *noeuds fils* (node.sons) est une liste de noeudsfils, que le noeud considéré contient.
- son *type* (node.type) est une chaîne de caractères qui permet de typer le noeud. Par exemple, les différents types qui sont utilisés actuellement sont : 'root', 'ui', 'pile', 'layer', 'graft', etc...

- attributs divers appelés "*marques*", à la disposition de l'utilisateur pour caractériser les noeuds plus avant.

1.3.2 Méthodes

Un arbre est donné par le noeud racine, et par tous les noeuds et les mots liés à ce noeud racine. Il s'agit d'une hiérarchie objet, ce qui signie qu'en plus de ses propriétés, chaque noeud dispose d'un certain nombre de méthodes qu'on peut appeler. En particulier :

- un noeud peut être déroulé (node.unroll) : c'est la fonction qui fait le dépliage. Elle renvoit une liste de phrases (entendues comme des listes de word cf plus haut) correspondant au dépliage de l'arbre à partir de ce noeud. L'usage typique est de l'appeler sur le noeud racine. La force de ce système est que le dépliage devient assez facile lorsque l'arbre est consitué.

- un noeud peut être converti en texte ou XML (node.prettyPrint ou node.XMLPrint): ce sont les fonctions qui convertissent en texte ou en XML la hiérarchie des noeuds en dessous du noeud considéré. Ces fonctions sont récursives, dans le sens où un noeud appelle les fonctions de ses enfants, de manière à ce que toute la hiérarchier en dessous de lui soit traitée. Une fois de plus, disposer d'une structure objet en arbre permet de changer facilement de convention pour le XML. On peut noter que l'ensemble des exemples donnés dans cette étude ont été obtenus par l'appel de la méthode prettyPrint sur des arbres réels.

- On peut créer des enfants à un noeud,

- déplacer des noeuds,

- détacher/attacher des noeuds d'un père donné vers un autre père,

- rechercher des noeuds descendants, de l'ensemble des symboles contenus dans la descendance d'un noeud etc...

D'une manière générale, l'essentiel est de comprendre qu'à partir des noeuds hierarchisés en arbre, tous les traitements deviennent instinctifs. Par exemple:

- On peut très facilement extraire tous les symboles prenoyaux d'un arbre

- On peut très facilement déterminer quels symboles sont à la fois dans des piles et énoncés par un locuteur donné

- On peut facilement ne conserver d'un arbre que les piles

- etc ...

2 Algorithme

2.1 Introduction

Cette section présente le principal algorithme de cette étude, celui qui permet de construire un arbre complet à partir d'un texte annoté. C'est lui qui est responsable de la mise en forme du texte en un arbre exploitable par la suite et présenté dans les section précédentes. D'autres algorithmes importants sont celui qui procède au dépliage et celui qui réalise les projections.

2.2 Algorithme

Initialization

- root = node(type='root')
- A=root.createSon(type='ui')
- currentNode←A
- currentPos = 0
- words = text.split()

while currentPos < len(words)

- switch words[currentPos].type
  - case "standard" : **add word to current node**
    - currentNode.content.extend(words[currentPos])
  - case [,(,(+: **create a new son [] or () and mark it if integrated**
    - A=currentNode.createSon(type = '[]' or '()' respectively)
    - currentNode←A
    - if (+ then A.mark('integrated')
  - case {: **create a new pile and its first layer**
    - A=currentNode.createSon(type = '{}')
    - currentNode←A.createSon(type = 'layer')
  - case < or <+: **if currentNode is a *kernel*, it should be a prekernel, and focus goes to father. If currentNode is not a *kernel*, then create a new prekernel, copy content of currentNode except *kernel* stuff, and move it to this new prekernel**
    - if 'kernel' in currentNode.type:
      - set currentNode.type to 'prekernel', 'integratedprekernel' respectively
      - currentNode←currentNode.father
    - else:
      - A←currentNode.content
      - reset all content of currentNode except nodes of type 'intro', 'kernel', 'prekernel', 'postkernel', 'integratedprekernel', 'integratedpostkernel'
      - B=currentNode.createSon(type='prekernel' or 'integratedprekernel' respectively)
      - copy all content of A into B except nodes of type 'intro'
  - case > or >+: **go to container parent, and create a new postkernel as currentNode**
    - while currentNode.type not in ['[]', 'layer', '()', 'ui']:
      - currentNode←currentNode.father
    - currentNode←currentNode.createSon(type = 'post kernel' or 'integratedpost kernel' respectively)
  - case ": **find next " and insert all content in between to a new discursivemarker**
    - currentPos ← currentPos+1
    - beginPos ← currentPos
    - while words[currentPos].type != ":
      - currentPos← currentPos+1
    - newNode=currentNode.createSon('discursivemarker')
    - newNode.content.append(words[beginPos:currentPos])
  - case ^: **add next word to a new intro node**
    - A=currentNode.createSon(type='intro')
    - add next word to A
    - currentPos←currentPos + 1
  - case //: **go to first container parent, and if its sons are ui, create a new one, else, insert all its content into a new ui son, and then create a new one**
    - while currentNode.type not in ['[]', 'layer', '()', 'ui']:
      - currentNode←currentNode.father
    - if currentNode.type == 'ui'
      - currentNode←currentNode.father.createSon(type='ui')
    - else:
      - insert node A of type currentNode.type between currentNode and currentNode.father
      - set currentNode.type to 'ui'
      - currentNode←A.createSon(type='ui')
  - case] or) or }: **go to corresponding parent node. For ')', mark it as integrated if node is integrated. For ']' and ')' and if node contains node of type different than 'ui', then change its type. Finally, if ']', node must be included in a *kernel*. If it is not the case, create that kernel and move currentNode in it.**
    - until currentNode.type == '[]', '()' or '{}' respectively

            - currentNode ←currentNode.father
        * if ')' AND currentNode.ismarked('integrated')
            - set currentNode.type to 'integratedinkernel'
            - currentNode.unmark('integrated')
        * elseif ']' or ')': AND if there are sons of currentNode that are not of type 'ui' or if content of currentNode is not empty
            - set currentNode.type to 'inkernel' for ')' or to 'embedded' for ']'
        * if ']' and currentNode.father.type not in [*'kernel*']:
            - A←currentNode.father
            - currentNode.detach()
            - kernels← A.getSonsOfType['kernel']
            - if kernels is empty, father←A.createSon('kernel') and move A.content into father.content
            - else father←kernels[0]
            - currentNode.attach(father)
        * currentNode ←currentNode.father
    - case |: **find parent pile, and create a new layer**
        * until currentNode.type == '{}'
            - currentNode ←currentNode.father
        * currentNode←currentNode.createSon(type='layer')
    - case |}: **find parent pile, mark it as interrupted and go to its parent node**
        * until currentNode.type == '{}'
            - currentNode ←currentNode.father
        * currentNode.mark('interrupted')
        * currentNode ←currentNode.father
    - case {|: **find the last interrupted pile son, unmark it, and create a new layer**
        * A= last (currentNode.getSonsOfType('{}')) where ismarked('interrupted'))
        * A.unmark('interrupted')
        * B=A.createSon(type='layer')
        * currentNode←A

- currentPos←currentPos+1

Termination

- delete all nodes such that neither them nor their progeny have any content
- for all nodes A in the tree:
    - if (A.type is not in '*kernel*', discursivemarker, intro) and A.content is not empty and A has no son of type kernel:
        * B=A.createSon(type='kernel')
        * move content of A into content of B