Analyse de dépendance par programmation dynamique

CYK : rappel

- Etant donné une phrase $S = w_1 \dots w_n$, on construit une table T, telle que
  - $A \in T[i][j] \Leftrightarrow A \stackrel{*}{\Rightarrow} w_i \dots w_j$
- Principe de construction :
  - Si $B \in T[i][k] \wedge C \in T[k+1][j] \wedge A \rightarrow B\,C \in R$
  - Alors $A \in T[i][j]$
- d'où l'algorithme

```
pour i = 1 à n faire { INITIALISATION }
  T[i][i] = {A|A → w_i}

pour j = 1 à n faire
  pour i = j − 1 à 1 faire
    pour k = i à j − 1 faire
      T[i][j] = T[i][j] ∪ {A|A → BC}
      avec B ∈ T[i][k] et C ∈ T[k+1][j]
```

Graph based Parsing : rappels

- **fonction de score** d'un arbre de dépendance $T = (V, A) \in \mathcal{T}(S)$:

$$\lambda(T) = \lambda(V, A) \in \mathbb{R}$$

- Etant donné une phrase S, on cherche l'arbre $\hat{T}$ de score maximal

$$\hat{T} = \arg\max_{T \in \mathcal{T}(S)} \lambda(T)$$

- Le score d'un arbre est égal à la somme du score de ses **facteurs**

$$\lambda(T) = \sum_{\psi \in \Psi(T)} \lambda(\psi)$$

- Modèle d'ordre 1 :

$$1 \text{ facteur } = 1 \text{ dépendance}$$

- Calcul du score des facteurs d'une phrase à l'aide d'un classifieur et stockage dans une table λ_1 :

$$\lambda_1[g][d][l]$$

Graph based Parsing + Algorithme CYK

- Peut on stocker dans une table toutes les analyses partielles ?
- Peut on calculer récursivement le score d'une analyse à partir du score de ses parties ?

Contiguité

- Un arbre de dépendance est **contigu** si, pour tout nœud de l'arbre n, l'ensemble des descendants de n correspondent à un segment de la phrase.
- Les arbres de dépendance projectifs possèdent la propriété de contiguité.
- Un arbre de dépendance contigu peut être construit de proche en proche, en combinant des segments contigus deux à deux pour constituer un segment

Combinaison de deux arbres contigus

- Soit deux segments $S_1 = w_i \dots w_k$ et $S_2 = w_{k+1} \dots w_j$ correspondant aux arbres A_1 et A_2, tel que A_1 a pour racine w_g et A_2 a pour racine w_d.
- Il est possible de combiner A_1 et A_2 pour former l'arbre A correspondant au segment $S = S_1 \cdot S_2$ de deux manières :

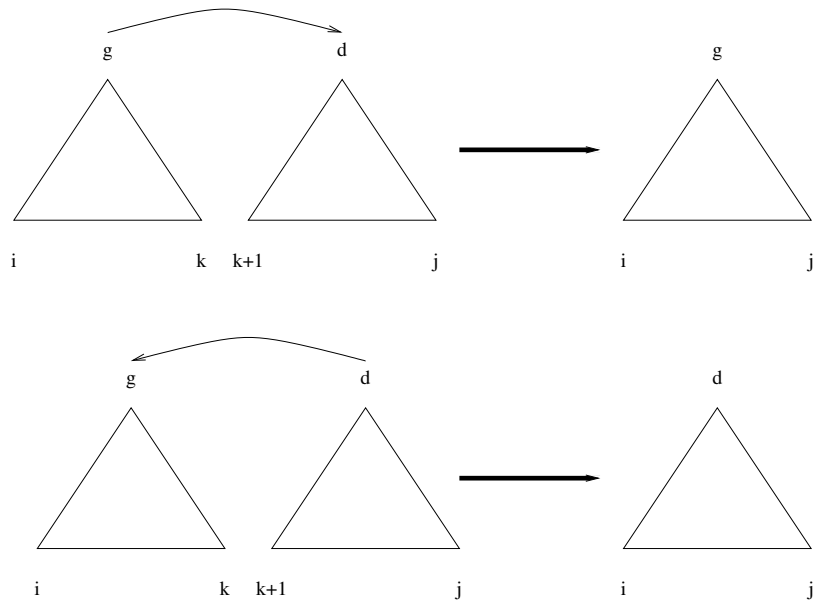


Table d'analyse

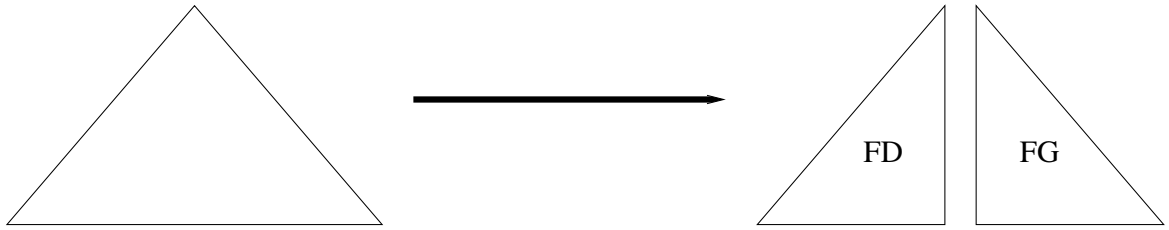
- On note $T[i][j][r]$ la table stockant le score du meilleur arbre correspondant au segment $w_i \dots w_j$ et ayant w_r pour racine.
- On remplit la table T de la manière suivante :

```
pour j = 1 à n faire
  pour i = j − 1 à 1 faire {segment [i, j]}
      pour k = i à j − 1 faire {on le coupe en deux : [i, k] et [k + 1, j]}
        pour r = i à k faire {on choisit une racine r dans [i, k]}
          pour l = k + 1 à j faire {on choisit une racine l dans [k + 1, j]}
            T[i][j][r] = max(T[i][j][r], T[i][k][r] + T[k + 1][j][l] + λ(r, l))
            T[i][j][l] = max(T[i][j][l], T[i][k][r] + T[k + 1][j][l] + λ(l, r))
```

Pas très efficace : $\mathcal{O}(n^5)$

Demi-arbres

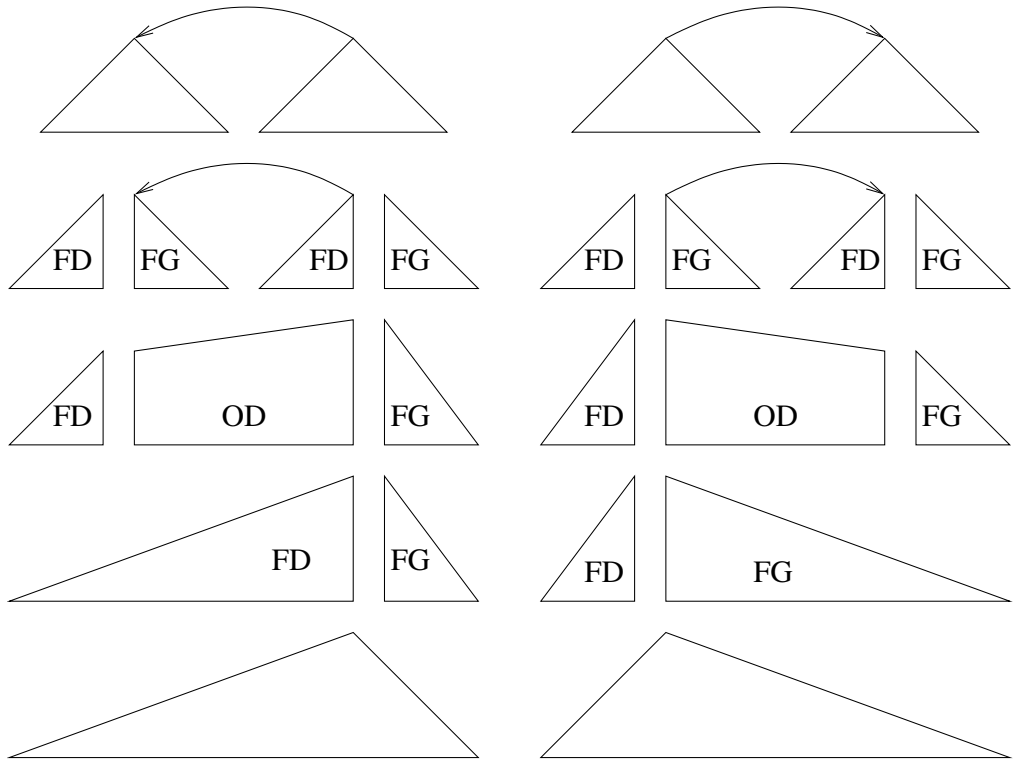
- Il est possible de décomposer un arbre en deux demi-arbre corresponants aux fils gauches et aux fils droits de la racine



- Le demi-arbre gauche est appelé Fermé Droit (FD)
- Le demi-arbre droit est appelé Fermé Gauche (FG)
- Ils peuvent être construits séparément l'un de l'autre.
- Etant donné un segment $[i, j]$ et l'entier k tel que $i \leq k < j$ on peut construire les FD du segment $[i, k]$ et les FG du segment $[k + 1, j]$ séparément.
- Il est ensuite possible de combiner chaque élément de $FD[i][k]$ avec chaque élément de de $FG[k + 1][j]$ pour construire tous les arbres du segment $[i, j]$

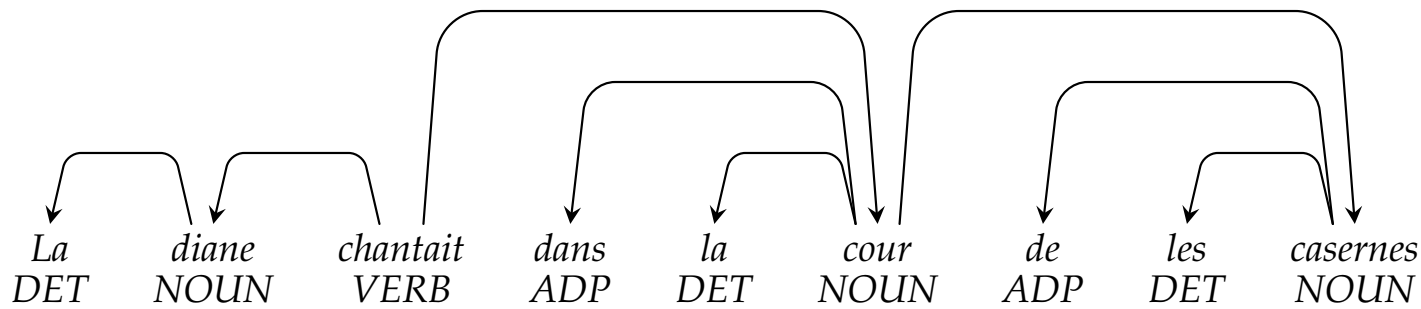
Nouvelles règles de combinaison d'arbres

La combinaison de deux arbres est maintenant réalisée en trois étapes :



Exercice

- Décomposer l'arbre suivant à l'aide des opérations de combinaison des OG, OD, FG, FD



- Au départ chaque mot est associé à un FG et un FD vides.

Remarque clef

Les structures FD, FG, OG et OD peuvent être stockés dans des tables à **deux dimensions** car la racine correspond au mot se trouvant soit à gauche, soit à droite du segment correspondant.

- $OD[i][j]$ Ouvert droit correspondant au segment $[i, j]$, la racine se trouve à droite (indice j)
- $OG[i][j]$ Ouvert gauche correspondant au segment $[i, j]$, la racine se trouve à gauche (indice i)
- $FD[i][j]$ Fermé droit correspondant au segment $[i, j]$, la racine se trouve à droite (indice j)
- $FG[i][j]$ Fermé gauche correspondant au segment $[i, j]$, la racine se trouve à gauche (indice i)

Variante de l'algorithme CYK pour structures de dépendances projectives

pour $i = 1$ **à** n **faire** { INITIALISATION }
  $FG[i][i] = FD[i][i] = 0$

pour $j = 1$ **à** n **faire**
  pour $i = j - 1$ **à** 1 **faire**
    pour $k = i$ **à** $j - 1$ **faire**
      $OG[i][j] = \max(OG[i][j], FG[i][k] + FD[k+1][j] + \lambda(i,j))$
      $OD[i][j] = \max(OD[i][j], FG[i][k] + FD[k+1][j] + \lambda(j,i))$
      $FD[i][j] = \max(FD[i][j], FD[i][k] + OD[k+1][j])$
      $FG[i][j] = \max(FG[i][j], FG[i][k] + OG[k+1][j])$

Plus efficace : $\mathcal{O}(n^3)$

Grammaire hors-contexte sous jacente

- On peut représenter les règles de combinaisons des objets OG, OD, FG, FD sous la forme de règles de réécriture.
- On crée ainsi une grammaire hors-contexte G qui permet de générer des arbres dont les nœuds internes correspondent aux symboles OG, OD, FG, FD et les feuilles aux symboles m et #.

  $$\begin{array}{lcl} \text{S} & \rightarrow & \text{FD}\ \ \text{FG} \\ \text{FD} & \rightarrow & \text{FD}\ \ \text{OD} \\ \text{FG} & \rightarrow & \text{OG}\ \ \text{FG} \\ \text{OG} & \rightarrow & \text{FG}\ \ \text{FD} \\ \text{OD} & \rightarrow & \text{FG}\ \ \text{FD} \\ \text{FG} & \rightarrow & m \\ \text{FD} & \rightarrow & \# \end{array}$$

- $L(G) = \{\#m, \#m\#m, \#m\#m\#m \ldots\}$
- G associe au mot $(\#m)^n$ tous les arbres projectifs pour une phrase de longueur n.

Exercices

- Dessiner tous les arbres de dérivation que la grammaire G associe aux phrases $\#m\#m$ et $\#m\#m\#m$
- Simulez l'algorithme CYK sur la phrase $\#m\#m\#m\#m$