Tecnologie del linguaggio naturale
(1^ parte)

Relazione Progetto

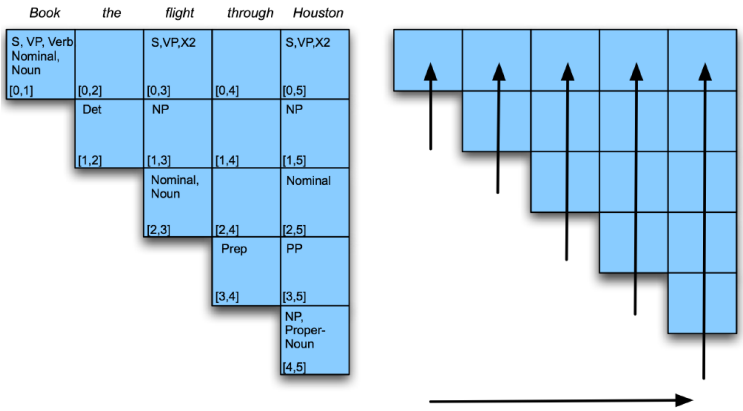
Walter Maltese, Pierandrea Morelli

Introduzione
La consegna da noi scelta per il presente progetto riguarda l'implementazione dell'algoritmo **CKY (Cocke Kasami Younger)**, che assume di lavorare su **grammatiche libere dal contesto**, utilizzando un approccio bottom-up con gestione della memoria tramite **programmazione dinamica**.

Un'assunzione dell'algoritmo CKY è che le regole di produzione siano nella **forma normale di Chomsky**, ovvero che siano del tipo:
A → BC
A → d

Ovvero abbiamo solo delle regole "da completare" e delle regole lessicali.
Ad ogni modo l'assunzione non lede la generalità dell'algoritmo visto che ogni grammatica con l'opportuna aggiunta di regole può essere convertita in forma normale di Chomsky.

L'intuizione che sta dietro all'algoritmo è la seguente: se abbiamo una regola di produzione della forma A → BC allora se A "domina" dalla parola i alla parola j della nostra frase, dovrà esistere un punto k tale per cui B "domina" dal punto i al punto k e tale per cui C "domina" dal punto k al punto j. L'idea è quindi di costruire dei cicli su questi indici e andare a salvare A in posizione [i, j] della tabella, B in posizione [i, k] e C in posizione [k, j].
La complessità dell'algoritmo è O(n3). Se ne mostra il codice nella figura sottostante.
Notiamo come la complessità O(n3) dell'algoritmo derivi dai tre for anni- dati. Intuitivamente la prima riga del primo for si occupa di riempire il livello più basso della tabella.
Nei due for interni andiamo invece a riempire i livelli superiori.




```
function CKY-PARSE(words, grammar) returns table

for j ← from 1 to LENGTH(words) do
    table[j − 1, j] ← {A | A → words[j] ∈ grammar}
    for i ← from j − 2 downto 0 do
        for k ← i + 1 to j − 1 do
            table[i,j] ← table[i,j] ∪
                {A | A → BC ∈ grammar,
                     B ∈ table[i,k],
                     C ∈ table[k,j]}
```

Implementazione grammatiche
Il file di riferimento che contiene le regole per le grammatiche utilizzate è “getGrammar.py”. Esso presenta, inoltre, un metodo `useGrammar(grammar)` che ci permette di utilizzare i dizionari (le grammatiche relative) presenti in questo script nel file principale (“CKY.py”), per l’esecuzione del parsing sulle frasi di esempio.

Partiamo discutendo del modo in cui abbiamo rappresentato le grammatiche utilizzate nel progetto.
Abbiamo scelto di implementarle tramite dei dizionari python, con chiavi gli antecedenti delle regole e come rispettivi valori le relative produzioni (i conseguenti di tali regole).
In particolare, tali valori dei dizionari sono inclusi in una lista di liste, che racchiude tutte le possibili produzioni valide per quello specifico antecedente della regola (la chiave di quel dizionario), e ciascuna di tali produzioni (es: ["NP", "VP"], derivante dalla chiave “S”) è, poi, raccolta a sua volta in una lista.

In tale maniera abbiamo, pertanto, rappresentato sia la grammatica L1 di Jurafsky, data in consegna, che la grammatica per la lingua Dothraki richiesta, facendo riferimento alla sintassi del Dothraki presente nella relativa pagina di wiki.
Abbiamo riportato, poi, come richiesto, le frasi da parsificare per le due grammatiche L1 e quella Dothraky in delle liste apposite contenenti le parole costituenti.

Grammatica L1

```
L1 = {
    "S": [["NP", "VP"],["X1","VP"],["book"],["include"],["prefer"],["Verb","NP"],["X2","PP"],["Verb","PP"],["VP","PP"]],
    "NP": [["Det","Nominal"],["I"],["she"],["me"],["TWA"],["Houston"]],
    "Nominal": [["Nominal","Noun"],["Nominal","PP"],["book"],["flight"],["meal"],["money"],["morning"]],
    "VP": [["Verb","NP"],["Verb","PP"],["VP","PP"],["book"],["include"],["prefer"],["X2","PP"]],
    "PP": [["Preposition","NP"]],
    "X1": [["Aux","NP"]],
    "X2": [["Verb","NP"]],
    "Det": [["that"],["this"],["a"],["the"]],
    "Noun": [["book"],["flight"],["meal"],["money"],["morning"]],
    "Verb":[["book"],["include"],["prefer"]],
    "Aux": [["does"]],
    "Preposition": [["from"],["to"],["on"],["near"],["through"]]
}
```

La grammatica L1 di Jurafsky fornita per il testing dell’algoritmo implementato è, secondo le assunzioni fatte al fine del corretto utilizzo del CKY, una grammatica context-free (una grammatica formale in cui ogni regola sintattica è della forma: V → w, dove V è un simbolo non terminale e w è una sequenza di simboli terminali e non terminali), le quali regole di produzione sono nella forma normale di Chomsky (sulla destra delle produzioni possono presentarsi esclusivamente o due simboli non terminali oppure un solo simbolo terminale).
Di conseguenza, nei modi esplicitati nel paragrafo precedente, abbiamo riportato nel nostro codice tale grammatica fornitaci, direttamente in Chomsky normal form.

Grammatica Dothraki
La lingua Dothraki ha avuto la sua genesi all’interno della serie televisiva americana Games of Thrones.
Di conseguenza, è stata creata ispirandosi nella sua struttura alla lingua inglese, nonostante ne differisce in alcuni aspetti.
L'ordine di base delle parole è SVO (soggetto-verbo-oggetto). In una frase di base, l'ordine di questi elementi (quando tutti e tre sono presenti) è come in inglese: prima viene il soggetto

(S), seguito dal verbo (V), e poi l'oggetto (O).
Il soggetto nelle frasi basilari è sempre presente, mentre in assenza dell'oggetto precede il verbo "SV".

```
Dothraki = {
    "S": [["NP", "VP"],["X1","VP"],["VP","PP"], ["NP","NP"], ["NP","JJ"]],
    "X1": [["Aux","NP"]],
    "Aux": [["hash"]],
    "VP": [["VP","NP"],["VP","ADV"],["astoe"],["dothrak"],["zhilak"],["dothrae"],["VP","PP"],["ittesh"],["VP","Pron"],["nesak"]],
    "PP": [["Preposition","NP"]],
    "NP":[["anha"],["yera"],["yer"],["Dothraki"],["NP","ADV"],["mahrazh"],["mori"],["lajakis"],["NP","JJ"], ["lajak"]],
    "Preposition": [["ki"]],
    "ADV":[["chek"],["ADV","ADV"],["asshekh"]],
    "JJ":[["ivezhi"],["JJ,JJ"],["mori"],['gavork']],
    "Pron":[["haz"]]
}
```

Si può notare, fra le altre regole, che è risultata necessaria la creazione della regola S -> NP NP per poter riconoscere quelle frasi che vengono definite "0-Copula Sentencies".
Questo costrutto sintattico rappresenta una delle maggiori differenze rispetto alla lingua inglese, poiché in quest'ultima la copula è data dal verbo essere (in inglese scriviamo lettermente "X è Y", "X era Y" o "X sarà Y").
In dothraki tale struttura è differente, in quanto la copula è del tutto assente.
E' sufficiente scrivere soltanto X-NOM Y-NOM, in cui X e Y possono essere un pronome o un nome ed entrambi nel caso nominativo, per indicare che X è Y.
E' il caso della frase "Anha gavork" (I'm hungry).

Un'altra differenza con l'inglese consiste nel diverso posizionamento di aggettivi e avverbi all'interno della frase: infatti se in inglese gli aggettivi sono posti antecedentemente al nome a cui si riferiscono, in Dothraki, invece, vengono inseriti a seguire ("NP -> NP ADJ").

Per quanto concerne gli avverbi, in inglese vengono posizionati vicino (spesso a seguire del verbo) alle parole che modificano mentre in dothraki sono usualmente posizionate a terminazione di frase.

In modo analogo alla lingua inglese l'ordine di base delle parole è "SVO" : prima viene il Soggetto (S), poi il Verbo (V), poi l'Oggetto (O).
Il soggetto nelle frasi basilari è sempre presente, mentre in assenza dell'oggetto precede il verbo "SV".

Asserito questo fatto è comprensibile basarsi sulle regole della grammatica "L1" per parsificare frasi dichiarative come: *anha zhilak yera (I love you).*

Di conseguenza sono state generate le seguenti regole:
S -> NP VP
VP -> Verb NP

In conclusione, per poter parsificare anche le frasi interrogative (es. "hash yer astoe ki Dothraki") abbiamo utilizzato le seguenti regole di produzione
"S -> X1 VP"
"X1 -> Part NP"

Infatti, in dothraki le domande di base iniziano con una "question word", ad es. "hash" o con un pronome interrogativo come "fin" per poi seguire il normale ordine delle parole SVO.

Strutture dati utilizzate

`cky_matrix = [[[] for j in range(n + 1)] for i in range(n + 1)]`
Matrice utilizzata per contenere i risultati dell'algoritmo CKY (restituito dal metodo `CKY`)

`tree_nodes = [[[] for i in range(n + 1)] for j in range(n + 1)]`
Matrice "di copia" per tener traccia di tutti i nodi inseriti, al fine di ricostruire gli alberi di derivazione (il risultato della stampa in S-espressioni del metodo `print_Tree_Nodes`)

`class Node`
Classe che permette la costruzione di alberi binari, rappresentanti le produzioni sulle frasi da parsificare col CKY.
Ogni oggetto **Node** è composto dai seguenti campi:

- "**root**": rappresenta la radice del nodo, contenente i non terminali delle regole di produzione
- "**left**" e "**right**": rappresentanti i sottoalberi sinistri e destri del nodo
- "**terminal**": permette di memorizzare il simbolo terminale (delle regole in forma A->"aaa")
- "**status**": consente di verificare lo stato di un nodo (se esso è terminale o meno)

Metodi implementati

`def parser_gs(grammar,sentence)`

È il metodo richiamato all'interno del main, permette di eseguire l'algoritmo CKY, dati in input una determinata grammatica (L1 o Dothraky nel nostro caso) e una relativa frase da parsificare (e1, e2, …), e conseguentemente effettuare una stampa del risultato del CKY sia tramite la matrice risultante che in forma di S-espressioni dell'albero generato.

Vedremo in seguito i metodi utilizzati al suo interno. ("`CKY`", "`print_CKY_Matrix`" e `print_Tree_Nodes`).

`def CKY(R,s):`

Dopo aver inizializzato le due matrici sopra citate,
giungiamo (alle riga 28) al primo ciclo for corrispondente all'iterazione più esterna dello pseudo codice del CKY.
Qui andiamo ad iterare su tutte le regole della grammatica passata come parametro (R) e si vanno a ricercare tutte le possibili produzioni di regole terminali, del tipo A (non terminale)-> alpha (terminale).

In particolare, andiamo ad iterare su ogni termine sinistro e destro delle regole della grammatica passata, e andiamo a cercare all'interno della lista di tali "rule" (i termini destri / valori delle regole) dei conseguenti ("right") con un solo termine (`len(right) == 1`).
Inoltre, ci accertiamo che il simbolo terminale che abbiamo trovato sia corrispondente al terminale target che stiamo attualmente considerando `right[0] == s[j - 1]`.
Possiamo notare come questo sia l'unico caso possibile di regole con conseguenti di lunghezza 1, dato che le grammatiche da noi utilizzate in input sono tutte in Chomsky Normal Form, e pertanto non possono esistere regole del tipo: S -> VP, ma soltanto A -> BC (non terminali) oppure A -> a (terminale).
Trovate le regole che soddisfano tali condizioni, possiamo pertanto aggiungere nella matrice cky_matrix in posizione [j-1] [j] (sulla diagonale della tabella) l'antecedente della regola trovata ("left").
Allo stesso modo inseriamo nella matrice tree_nodes un nuovo nodo (rappresentante un sottoalbero) con radice "left" (tale termine antecedente della regola trovata) e come sottoalberi figli sinistri e destri "None", in quanto si tratta di regole del tipo A -> a.
L'ultimo parametro (s[j-1])contiene l'informazione da memorizzare per la futura stampa (il terminale vero e proprio).

Giungiamo così, alle righe 44-45, ai successivi due cicli for dell'algoritmo, che si interessano della ricerca di regole non più "terminali" ma del tipo A -> B C, al fine di inserire nuovi elementi nella matrice del CKY.
In particolare, andiamo come fatto in precedenza a scorrere su tutti i conseguenti delle regole associate alla grammatica considerata (i termini "right") e verifichiamo se ricadiamo nel caso delle regole citate.
Infatti, quando vale la condizione `len(right) == 2`, e ci siamo accertati del fatto che i simboli non terminali presenti proprio in tale conseguente (il right considerato) sono già presenti nella grammatica del CKY nelle posizioni [i][k] e [k][j] della matrice, possiamo inserire all'interno di quest'ultima il termine "left" (l'antecedente) della regola in questione, in posizione [i][j] della matrice.

Per concludere, come fatto in precedenza, alle righe 63-66, andiamo a riempire di pari passo anche la matrice (rappresentante la struttura sintattica dell'albero) dei nodi dell'albero che si sta generando.
In essa, infatti, ogni casella contiene i sottoalberi (nodi) della matrice CKY risultante.
Qui controlliamo, pertanto, l'esistenza dei 2 sottoalberi B e C (i nodi con radici i due termini NT trovati nella regola appena considerata valida).
Se tali nodi esistono, allora aggiungiamo, analogamente a quando fatto per la cky_matrix (in cui aggiungevamo nella matrice il termine "left" relativo alla regola interessante trovata), un nuovo Node (con root il simbolo non terminale A (il "left" appena trovato) e come sottoalberi figli left e right i due simboli non terminali B e C).

Scegliamo di restituire come valori di ritorno del metodo:
`tree_nodes[0][n]` , rappresentante l'albero completo della matrice tree_nodes costruita, in quanto in tale maniera restituiamo le radici di tutti i vari alberi risultanti (dalla relativa radice ritrovo poi i collegamenti a tutti i sottoalberi);
`cky_matrix ,` rappresentante l'intera matrice del CKY generata.

`def print_CKY_Matrix(result)`
Il presente metodo permette la corretta stampa della matrice del CKY risultante (passata come parametro), in seguito all'applicazione del relativo metodo.
Si è scelto di usufruire della struttura dati "DataFrame" (importata dal modulo "pandas").
Pandas DataFrame è una struttura dati tabulare bidimensionale, potenzialmente eterogenea, con assi etichettati (righe e colonne). Un DataFrame è una struttura dati bidimensionale, cioè i dati sono allineati in modo tabulare in righe e colonne.
Pandas DataFrame consiste di tre componenti principali, i dati, le righe e le colonne.
Il funzionamento di questo metodo è molto semplice, in quanto si tratta di un semplice controllo sull'elemento [0][n] della matrice (ovvero la radice dell'albero).
Se quest'ultimo corrisponde al simbolo "S", rappresentante la radice (lo "Start-symbol") della grammatica considerata, si attesta che la frase che è stata parsificata è di senso compiuto, data la grammatica utilizzata in input.
Pertanto, se ricadiamo in tale caso considerato si effettua la stampa del DataFrame risultante, o in caso contrario si indica l'invalidità della frase in input.

`def print_Tree_Nodes(nodes_back)`
Questo metodo si occupa, in modo simile al precedente, di effettuare una stampa tramite S-espressioni dell'albero / degli alberi risultanti, costruiti in seguito al riempimento della matrice tree_nodes.
Ricevendo, infatti, in input come parametro la radice dell'albero risultante (tree_nodes[0][n]), posso risalire, tramite i collegamenti intrinsechi della struttura della classe Node, a tutti i sottoalberi (sinistri e destri) derivati.
Per ogni nodo radice presente in tale posizione della matrice tree_nodes, pertanto, controllo se esso corrisponda al termine "S" (rappresentate una corretta esecuzione del CKY sulla frase data in input), ed in tal caso effettuo la stampa del nodo (e relativo sottoalbero associato), tramite una chiamata al metodo (build_String_Tree).

`def build_String_Tree(root, indent)`

Completa il lavoro del precedente metodo, ricevendo in input il nodo radice dell'albero da stampare (sotto forma di S-espressioni), ed un parametro di indentazione, al fine della corretta visualizzazione della stampa in output.
Distingue al suo interno, tramite l'utilizzo del campo "status" del nodo root in input, i nodi terminali (casi base della ricorsione) da quelli non terminali (su cui fare ricorsione).
Nel primo caso, infatti, si procederà alla semplice stampa del nodo terminale, altrimenti si effettuerà ricorsione sui figli sinistri e destri del nodo root (non terminale) e si stamperà di conseguenza una S-espressione rappresentante l'albero con radice "root" e con figli "left" e "right" (indentati secondo il parametro "indent" in input, incrementato ogni volta per indicare il livello di profondità in cui si è giunti).

Test e risultati

Vediamo adesso gli output raggiunti utilizzando la grammatica L1 e la frase "Book the flight through Houston" in input (una delle frasi richieste per l'utilizzo con la grammatica L1 fornita).
Possiamo notare come la matrice del CKY risultante, che viene stampata come primo risultato in output dal programma, mostra nella casella in alto a destra ([0][n]), rappresentante la radice dell'albero generato tramite la frase parsificata, che essa mostra al suo interno 3 simboli "S", attestanti la corretta parsificazione di una frase in senso compiuto, tramite la grammatica L1 adoperata.
In particolare, essendovi 3 diverse "S" in tale casella, ciò mostra come la frase possa giungere ad una corretta parsificazione tramite la grammatica in 3 modi differenti (ossia siamo di fronte ad una frase "ambigua"), a seconda delle produzioni utilizzate proprio all'interno di tale grammatica.

```
| Valid sentence |
```

	1	2	3	4	5
0	[S, Nominal, VP, Noun, Verb]	[]	[S, VP, X2]	[]	[S, VP, X2, S, S, VP, VP]
1		[]	[Det]	[NP]	[]	[NP]
2		[]	[]	[Nominal, Noun]	[]	[Nominal]
3		[]	[]	[]	[Preposition]	[PP]
4		[]	[]	[]	[]	[NP]

Notiamo, pertanto, come i 3 differenti alberi (in forma di S-expressions) vengano stampati come secondo output dal programma, indicando le differenti alternative di produzioni da utilizzare al fine di giungere alla corretta parsificazione.

```
(S (Verb book)
   (NP (Det the)
       (Nominal (Nominal flight)
                (PP (Preposition through)
                    (NP Houston)))))
(S (X2 (Verb book)
       (NP (Det the)
           (Nominal flight)))
   (PP (Preposition through)
       (NP Houston)))
(S (VP (Verb book)
       (NP (Det the)
           (Nominal flight)))
   (PP (Preposition through)
       (NP Houston)))
```

A titolo di esempio, vediamo il funzionamento del primo di tali alberi di derivazione:
la radice dell'albero (S), tramite la produzione della grammatica L1 utilizzata (S -> Verb NP), produce (come figli di tale nodo radice) Verb e NP (che possiamo vedere essere posti alla stessa altezza come indentazione); essi, a loro volta, avranno come conseguenti i risultati di produzioni ulteriori della grammatica.
Infatti, Verb termina nel simbolo terminale book, mentre NP si divide a sua volta in Det e Nominal, (ecc.).

Per concludere, abbiamo inserito all'interno del main la possibilità di visualizzare il medesimo output per la grammatica Dothraki ed una relativa frase da parsificare ("anha zhilak yera" / "I love you").
È sufficiente togliere la dicitura di commento alla riga 115 per eseguire nuovamente il metodo "parser_gs" sugli input aggiornati).