Relazione progetto di laboratorio TLN

Bushaj Aldo - Bushaj Antonino

Introduzione al problema

L'algoritmo di CKY (Cocke–Younger–Kasami algorithm) nasce per cercare di risolvere i problemi visti a lezione, generati dai parser top-down e left-to-right.

Panoramica generale sul CKY

Il CKY è un algoritmo di parsing che utilizza una strategia di ricerca bottom-up e left-right. Il parser può essere valutato sulla base di tre aspetti:

- **Una grammatica** context free in forma normale di Chomsky e dobbiamo assicurarci che la stringa che stiamo trattando sia un linguaggio della grammatica.
- **L'algoritmo** nello specifico si suddivide in due parti:
    - una strategia di ricerca bottom-up dove partiamo dalle parole e risaliamo fino alla radice e left-to-right ossia leggiamo le parole da sinistra verso destra, che è il modo di operare che hanno gli esseri umani. Consideriamo che vi sono varie alternative alla soluzione implementata in quanto possiamo avere strategie di ricerca right-to-left o anche bidirezionali che sono quelli che danno le prestazioni migliori.
    - organizza la memoria in modo dinamico, in particolare sfrutta la programmazione dinamica in modo da evitare problemi di esplosione combinatoria. Anche qui vi sono alternative quali organizzazione della memoria con back-tracking o il beam search
- **L'oracolo** infine che è di tipo rule-based

Di seguito riportiamo lo pseudocodice dell'algoritmo

```
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 algoritmo CKY

Analogamente allo pseudo codice sopra riportato,nella nostra versione abbiamo implementato il CKY seguendola struttura illustrata di seguito. Abbiamo utilizzato gli stessi indici in quanto questa è la versione ottimizzata dell'algoritmo, infatti questi indici sono stati già testati e danno garanzia di efficienza.

Funzioni utilizzate

Il CKY utilizza fondamentalmente tre funzioni ausiliarie:

- **get_rules:** la quale prendendo in input la grammatica utilizzata (quindi all'occorrenza Dothraki e quella sulla lingua Inglese), ci permette di ottenere tutte le possibili regole che producono ciò che sto cercando. Posso trovarmi nella situazione in cui cerco tutte le regole che producono dei terminali, o regole che producono non terminali, vediamole nel dettaglio:
    - **terminali:** nel caso in cui sto cercando le regole che producono i terminali, per capire meglio cosa fa questa funzione vediamo un esempio pratico.

      Se stiamo cercando le regole che producono la parola *book*, la funzione get_rules mi restituirà l'array *ret_rules* contenente tutte le regole che producono book, ES:

        - Verb -> book, include, ….
        - Nominal -> book, flight, meal, ….
        - Noun -> book, morning, ….
    - **non terminali:** nel caso in cui invece sto cercando le regole che producono i non terminali, vediamo cosa restituisce la funzione con un esempio pratico.

      In questo caso noi passiamo due non terminali VERB NP, e cerchiamo tutte le regole che producono questi due non terminali. Quindi in questo caso la funzione get_rules mi restituirà l'array *ret_rules* contenente tutte le regole che producono VERB ed NP, ES:

        - VP -> Verb NP, VNP PP, ….
        - VNP -> Verb NP
        - S -> Verb NP, ANP VP, NP VP, ….
- **save_rules:** questa funzione invece semplicemente salva le regole che mi producono la parola/terminale trovate al passo precedente, salvandole sia nel dizionario grammar_rules che nella matrice parsing_matrix. Questo fa uso del principio di gestione della memoria dinamica, quindi andiamo a salvare i nostri risultati intermedi che fungono da soluzione per piccole parti di problemi, in quanto queste non interferiscono tra loro perciò nel momento

in cui abbiamo una parte di soluzione la salviamo senza dover rieffettuare il lavoro da zero ogni volta

- **find_matches:** infine l'ultima funzione utilizzata, trova tutte le possibili combinazioni di regole a partire dalle due liste date in input. Quindi quello che restituisce è una matrice di tutte le terne dei match trovati, tra le varie combinazioni:
    - Es: [[VP, 4, 5], [PP, 6, 5]]), ecc….

  Funzione utilizzata per creare la nuova radice A che produce i due sottoalberi B e C adiacenti tra loro.

Quindi quello che facciamo nell'algoritmo CKY implementato, è chiamare opportunamente le funzioni appena citate, all'interno delle varie iterazioni dei for.

Infine abbiamo implementato alcune funzioni che ci permettono di visualizzare i risultati prodotti, e quindi avere in output sia la matrice creata che gli alberi con le possibili interpretazioni.

Quindi quello che abbiamo fatto è stato dichiarare una funzione **print_result** la quale prima di tutto stampa la matrice risultante della frase data in input utilizzando la funzione ausiliaria print_matrix.

Subito dopo andrà a verificare che la frase passata in input, dopo che il CKY ha fatto le opportune operazioni, appartiene o meno alla grammatica. Per fare ciò verifichiamo di ottenere la radice S, il che significa che siamo riusciti a produrre l'albero, quindi chiamiamo la funzione ausiliaria **print_tree** che mi produrrà in output tutti i possibili alberi e quindi tutte le possibili interpretazioni della frase data in input, più una frase è ambigua quindi con possibili diverse interpretazioni e più sono gli alberi prodotti .

Strutture dati

In questa sezione vediamo le due grammatiche analizzate con le relative frasi su cui le abbiamo testate.

CKY su grammatica L1

Grammatica in forma normale di Chomsky

$S \rightarrow NP\ VP$	$S \rightarrow NP\ VP$
$S \rightarrow Aux\ NP\ VP$	$S \rightarrow X1\ VP$
	$X1 \rightarrow Aux\ NP$
$S \rightarrow VP$	$S \rightarrow book \mid include \mid prefer$
	$S \rightarrow Verb\ NP$
	$S \rightarrow X2\ PP$
	$S \rightarrow Verb\ PP$
	$S \rightarrow VP\ PP$
$NP \rightarrow Pronoun$	$NP \rightarrow I \mid she \mid me$
$NP \rightarrow Proper\text{-}Noun$	$NP \rightarrow TWA \mid Houston$
$NP \rightarrow Det\ Nominal$	$NP \rightarrow Det\ Nominal$
$Nominal \rightarrow Noun$	$Nominal \rightarrow book \mid flight \mid meal \mid money$
$Nominal \rightarrow Nominal\ Noun$	$Nominal \rightarrow Nominal\ Noun$
$Nominal \rightarrow Nominal\ PP$	$Nominal \rightarrow Nominal\ PP$
$VP \rightarrow Verb$	$VP \rightarrow book \mid include \mid prefer$
$VP \rightarrow Verb\ NP$	$VP \rightarrow Verb\ NP$
$VP \rightarrow Verb\ NP\ PP$	$VP \rightarrow X2\ PP$
	$X2 \rightarrow Verb\ NP$
$VP \rightarrow Verb\ PP$	$VP \rightarrow Verb\ PP$
$VP \rightarrow VP\ PP$	$VP \rightarrow VP\ PP$
$PP \rightarrow Preposition\ NP$	$PP \rightarrow Preposition\ NP$

Utilizzo

L’algoritmo è stato testato sulle seguenti frasi

- Book the flight through Houston
- Does she prefer a morning flight

Abbiamo visto a lezione che le frasi sono ambigue, con diversi alberi in quanto prendiamo il caso di “through Houston" questa può assumere diversi significati through Houston può modificare o il singolo VP o il singolo S , quindi pensare “through Houston" dove Houston può essere un motore di ricerca quindi che va a modificare il volo, oppure una città dalla quale passa il nostro volo

Risultati

Come possiamo vedere dall’immagine sotto, la prima frase “Book the flight through Houston” ha tre possibili alberi di derivazione, infatti identifica correttamente le tre possibili interpretazioni che possiamo dare alla frase.

```
S ->
        Verb = Book
        NP ->
                Det = the
                Nominal ->
                        Nominal = flight
                        PP ->
                                Prep = through
                                NP = Houston
-----------------------------------------------------------------------
S ->
        VP ->
                Verb = Book
                NP ->
                        Det = the
                        Nominal = flight
        PP ->
                Prep = through
                NP = Houston
-----------------------------------------------------------------------
S ->
        VNP ->
                Verb = Book
                NP ->
                        Det = the
                        Nominal = flight
        PP ->
                Prep = through
                NP = Houston
```

Per la seconda frase analizzata invece vediamo il relativo albero di derivazione ottenuto, con l'unica possibile interpretazione possibile.

```
   S ->
         ANP ->
                 Aux = Does
                 NP = she
         VP ->
                 Verb = prefer
                 NP ->
                         Det = a
                         Nominal ->
                                 Nominal = morning
                                 Noun = flight
```

Algoritmo Cky sulla grammatica della lingua immaginaria

In questa sezione analizziamo i risultati ottenuti con la seconda grammatica, quella della lingua immaginaria Dothraki.

Grammatica Dothraki

```
Dothraki_Grammar = {
    # Grammatica che produce le frasi seguenti
    "S": ["AV PN", "NP VP", "NP Noun", "NP Verb"],
    "AV": ["ANP Verb" ],
    "ANP":["Aux NP2"],
    "Aux":["hash"],
    "NP": [ "yera", "anha"], # nominativo
    "NP2": ["yer"], # accusativo
    "PN": ["Prep Noun"],
    "VP": ["Verb NP"],
    "Prep": ["ki"],
    "Noun": ["Dothraki", "gavork"],
    "Verb": ["astoe", "zhilak"]
}
```

Utilizzo

L'algoritmo è stato utilizzato sulle seguenti frasi

- Hash yer astoe ki Dothraki? (Do you speak Dothraki?)
- Anha zhilak yera (I love you)
- Anha gavork (I'm hungry)

La grammatica Dothraki utilizza 5 casi, simili a quelli del latino, "nominative, accusative, genitive, ablative and allative". Una parte della frase può assumere diversi part-of-speech, il soggetto di un verbo solitamente prende il nominativo, l'oggetto l'accusativo, il genitivo come nel latino è utilizzato nelle costruzioni possessive(di lui, di lei).
Come possiamo vedere nelle frasi trattate nell'esercizio nella frase:

- ➔ "Hash yer astoe ki Dothraki?" yer in questo caso assume la funzione di soggetto in quanto è l'elemento a cui si rivolge la domanda e viene scritto con *yer*
- ➔ "Anha zhilak yera" in questo caso invece tu/te nella frase i love you il pronome ha una forma diversa rispetto alla frase precedente nonostante si riferisca sempre alla seconda persona singolare, questo perché come accennato la grammatica si basa sui casi che possono assumere forme diverse quindi la differenza è che nella prima frase tu prende il nominativo, nella seconda frase invece lo stesso pronome prende l'accusativo quindi abbiamo yer/yera. Queste differenze sono state sottolineate anche nella stesura della grammatica dove abbiamo
  - "NP": ["yera", "anha"]
  - "NP2": ["yer"]

  Nel primo caso sono entrambi soggetto, NP2 invece abbiamo il caso accusativo appunto.

Risultati

Vediamo qui di seguito gli alberi ottenuti dalla parificazione delle 3 frasi, notando che in tutte le soluzioni abbiamo una sola interpretazione possibile.

Hash yer astoe ki Dothraki? (Do you speak Dothraki?)

```
S ->
        AV ->
                ANP ->
                        Aux = Hash
                        NP2 = yer
                Verb = astoe
        PN ->
                Prep = ki
                Noun = Dothraki
```

Anha zhilak yera (I love you)

```
S ->
        NP = Anha
        VP ->
                Verb = zhilak
                NP = yera
```

Anha gavork (I'm hungry)

```
S ->
        NP = Anha
        Noun = gavork
```

Conclusioni

Per la lingua immaginaria abbiamo costruito una grammatica semplice, che ci permettesse di parsificare correttamente le tre frasi su cui bisognava testarle, quindi non copre tutti i possibili casi e tutte le possibili frasi che si possono formulare in Dothraki.

Concludiamo dicendo che come abbiamo visto anche nelle sezioni precedenti, l'algoritmo funziona correttamente in quanto riesce a generare correttamente gli alberi per le frasi appartenenti alla grammatica, sia considerando la grammatica L1 che quella della lingua immaginaria Dothraki.