ANNO ACCADEMICO 2024/2025

Tecnologie del Linguaggio Naturale

CKY Elfico

Luca Barra

UNIVERSITÀ
DI TORINO

DIPARTIMENTO DI INFORMATICA

1 Overview Generale

1.1 Il Progetto

La consegna del progetto prevedeva di implementare l'algoritmo CKY (Cocke-Kasami-Younger), un algoritmo per il parsing di grammatiche context-free in CNF (Chomsky Normal Form).

1.1.1 CKY in Breve

Concetti chiave:

- La grammatica fornita deve essere in CNF:
  - $A \rightarrow BC$ (con B e C simboli non-terminali).
  - $A \rightarrow a$ (con a simbolo terminale).
- Programmazione dinamica bottom-up.

Domanda 1.1

Come funziona l'algoritmo?

Prendiamo in considerazione una stringa $s = w_1, \ldots, w_n$:

1. **Inizializzazione:**
   - Viene creata una matrice T di dimensioni nxn dove $T[i][j]$ contiene l'insieme dei simboli non terminali che possono essere generati dalla sottostringa $w[i \ldots j]$.
   - Per ogni parola w_i in input:
     - Per ogni regola $A \rightarrow w_i$, viene aggiunta A a $T[i][i]$
2. **Filling the table:**
   - Per ogni lunghezza di sottostringa l da 2 a n:

     Per ogni indice iniziale i da 0 a $n - l$:

     Sia $j = i + l - 1$

     Per ogni punto di divisione k da i a $j - 1$:

     Per ogni regola binaria $A \rightarrow B\ C$:

Se $B \in T[i][k]$ e $C \in T[k+1][j]$, allora:
Aggiungi A a $T[i][j]$

3. **Finale:**
   - Se il simbolo iniziale della grammatica (S) è presente in $T[0][n-1]$, allora la stringa appartiene al linguaggio generato dalla grammatica.
   - In caso contrario, la stringa non è derivabile secondo le regole della grammatica.

1.2 Semplice Grammatica Quenya CF

Grammatica in CNF

$$
\begin{aligned}
S &\rightarrow NP\ VP \\
S &\rightarrow \text{Pronverb NP} \\
\\
NP &\rightarrow Det\ NP \\
NP &\rightarrow NP\ \text{Noun} \\
NP &\rightarrow \text{hesto} \mid \text{macil} \mid \text{aran} \mid \text{aiwi} \mid \text{eldar} \mid \text{atan} \mid \text{eldan} \mid \text{lumba} \mid \text{tecil} \\
Noun &\rightarrow \text{hesto} \mid \text{macil} \mid \text{aran} \mid \text{aiwi} \mid \text{eldar} \mid \text{atan} \mid \text{eldan} \mid \text{lumba} \mid \text{tecil} \\
VP &\rightarrow Verb\ NP \\
\\
\text{Pronverb} &\rightarrow \text{Nanye} \mid \text{Nalme} \\
\text{Verb} &\rightarrow \text{same} \mid \text{tira} \mid \text{antane} \\
\text{Det} &\rightarrow \text{I}
\end{aligned}
$$

Per costruire questa semplice grammatica si è partiti da un'analisi di 5 frasi:

1. I hesto samë macil.
   - $S \rightarrow NP\ VP$
   - $NP \rightarrow Det\ NP$
   - $Noun \rightarrow hesto$
   - $VP \rightarrow Verb\ NP$
   - $Verb \rightarrow same$
   - $Noun \rightarrow macil$
2. I aran tíra aiwi.
   - $S \rightarrow NP\ VP$
   - $NP \rightarrow Det\ NP$
   - $Det \rightarrow i$
   - $NP \rightarrow aran$
   - $VP \rightarrow Verb\ NP$
   - $Verb \rightarrow tira$
   - $NP \rightarrow aiwi$
3. Nanye lumba.
   - $S \rightarrow Pronverb\ NP$

- $Pronverb \rightarrow Nanye$
- $Noun \rightarrow lumba$

4. Nálmë eldar.
    - $S \rightarrow Pronverb\ NP$
    - $Pronverb \rightarrow Nalme$
    - $NP \rightarrow eldar$
5. I atan antanë i eldan tecil.
    - $S \rightarrow NP\ VP$
    - $NP \rightarrow Det\ NP$
    - $Det \rightarrow i$
    - $NP \rightarrow atan$
    - $VP \rightarrow Verb\ NP$
    - $Verb \rightarrow antane$
    - $NP \rightarrow Det\ NP$
    - $Det \rightarrow i$
    - $NP \rightarrow NP\ Noun$
    - $NP \rightarrow eldan$
    - $Noun \rightarrow tecil$

2 Implementazione

2.1 L'Algoritmo CKY

Listing 2.1: Algoritmo CKY

```
for length in 2..=n {
    for i in 0..=n - length {
        let j = i + length - 1;
        for k in i..j {
            let left_set = table[i][k].clone();
            let right_set = table[k + 1][j].clone();
            for left in &left_set {
                for right in &right_set {
                    let possible_lhs =
                        grammar.get_non_terminals(&[left.symbol.clone(),
                        right.symbol.clone()]);
                    for lhs in possible_lhs {
                        let parent_node = ParseTreeNode {
                            symbol: lhs,
                            children: vec![left.clone(), right.clone()],
                         };
                        table[i][j].insert(parent_node);
                    }
                }
            }
        }
    }
}
```

2.1.1 Struttura

- Il primo loop (`for length in 2..=n`) va a iterare sulla lunghezza delle sottostringhe da 2 fino alla lunghezza della frase n.
- Successivamente (`for i in 0..=n - length { let j = i + length - 1; }`), per una data lunghezza di sottostringa, si considerano tutte le possibili posizioni iniziali i, calcolando la posizione finale j.

- Il terzo loop (`for k in i..j`) esplora tutte le possibili divisioni (punti di split) della sottostringa corrente, dividendola in due parti: $[i..k]$ e $[k+1..j]$.
- Per ogni possibile divisione, si estraggono i sottoinsiemi di simboli non terminali presenti nelle celle $table[i][k]$ e $table[k+1][j]$, che rappresentano rispettivamente il lato sinistro e destro della regola binaria.
- Per ogni combinazione di simboli $left$ e $right$ nei due insiemi, si consultano le regole della grammatica alla ricerca di una regola $A \rightarrow left\ right$ valida.
- Se esiste una regola del genere, si crea un nuovo nodo dell'albero sintattico con simbolo A e figli $left$ e $right$, e si inserisce questo nodo nella cella $table[i][j]$.
- Alla fine dell'algoritmo, la cella $table[0][n-1]$ conterrà tutti i possibili alberi sintattici completi che derivano l'intera frase. Se tra questi alberi è presente un nodo con simbolo iniziale della grammatica (S), allora la frase è grammaticalmente valida.

Note:-

La complessità è $O(n^3)$ perché ci sono n^2 celle nella matrice del CKY e per ogni cella si provano n possibili "split point".

2.1.2 Output

Il CKY originale è un recognizer che restituisce solo se una frase appartiene o meno alla grammatica. In questo caso è stata implementata una semplice estensione per cui viene restituito anche l'albero di parsing generato dalla frase come JSON.

2.2 La Gestione delle Grammatiche

2.2.1 Struttura

Listing 2.2: Struttura rappresentante una grammatica CFG

```
pub struct Cfg {
    /// Grammar rules stored as a map from non-terminal symbols to possible RHS sequences.
    rules: HashMap<String, Vec<Vec<String>>>,
}
```

Questa struttura (`struct`) rappresenta una grammatica libera dal contesto (CFG) utilizzando una `HashMap`. La chiave è un simbolo non terminale (`String`), mentre il valore associato è un vettore di vettori di stringhe (`Vec<Vec<String»`), dove ciascun vettore interno rappresenta una possibile produzione per quel simbolo.

Listing 2.3: Inizializzazione di una grammatica

```
pub fn new(grammar: &str) -> Self {
        let file_path = format!("rsrc/grammar/{}.json", grammar);

        if !Path::new(&file_path).exists() {
            panic!("Grammar␣file␣'{}'␣does␣not␣exist", file_path);
        }

        let json_data = fs::read_to_string(&file_path)
            .unwrap_or_else(|_| panic!("Failed␣to␣read␣JSON␣file:␣{}", file_path));

        let rules: HashMap<String, Vec<Vec<String>>> =
            serde_json::from_str(&json_data).expect("Failed␣to␣parse␣JSON");
```

```
        Cfg { rules }
}
```

Il metodo `new` consente l'inizializzazione della grammatica a partire da un file JSON. Viene costruito dinamicamente il percorso del file, che viene poi letto e deserializzato tramite `serde_json`. Grazie al sistema di tipi statici di Rust, il parser JSON verifica che la struttura dei dati nel file corrisponda esattamente al tipo previsto: `HashMap<String, Vec<Vec<String»>`. In questo modo, la deserializzazione fallisce immediatamente se il formato del file non è corretto, garantendo maggiore sicurezza e robustezza.

Listing 2.4: Ricerca delle produzioni inverse nella grammatica

```
pub fn get_non_terminals(&self, sequence: &[String]) -> HashSet<String> {
    let mut result = HashSet::new();
    for (lhs, rhs_list) in &self.rules {
        for rhs in rhs_list {
            if *rhs == sequence {
                result.insert(lhs.clone());
            }
        }
    }
    result
}
```

Questa funzione ha un ruolo fondamentale durante il parsing CKY. Riceve in input una sequenza di simboli (esempio `["NP", "VP"]`) e restituisce tutti i simboli non terminali che, secondo la grammatica caricata, possono generare direttamente quella sequenza. L'algoritmo itera attraverso tutte le regole della grammatica (`rules`), e confronta ogni produzione con la sequenza passata in input. Se trova una corrispondenza esatta, inserisce il simbolo di sinistra (LHS) all'interno di un `HashSet`, che rappresenta l'insieme dei simboli non terminali validi per quella specifica produzione. Questa operazione è essenziale nella fase centrale dell'algoritmo CKY, quando si costruiscono nuovi nodi dell'albero sintattico combinando due sottostrutture: si verifica infatti se esiste una regola binaria $A \rightarrow B\ C$ che possa giustificare la fusione dei due figli.

2.2.2 Scrivere Grammatiche

Le grammatiche devono essere fornite come JSON. Per rendere più facile e veloce la scrittura è stato implementato un semplice converter che prende una grammatica in CNF scritta in modo naturale e la converte in JSON.

Listing 2.5: Converter

```
  for line in reader.lines() {
      let line = line?;
      if line.trim().is_empty() || line.trim().starts_with("//") {
          continue;
      }

      if let Some(caps) = re.captures(&line) {
          let lhs = caps[1].to_string();
          let rhs = caps[2]
              .split('|')
              .map(|s| s.trim())
              .filter(|s| !s.is_empty())
```

```
                .map(|rhs_str| {
                    rhs_str
                        .split_whitespace()
                        .map(|s| {
                            if terminal_re.is_match(s) {
                                terminal_re.replace(s, "$1").to_string()
                            } else {
                                s.to_string()
                            }
                        })
                        .collect::<Vec<String>>()
                })
                .collect::<Vec<Vec<String>>>();

            rules_map.entry(lhs).or_default().extend(rhs);
        }
    }
```