UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona
FIB

Implementació de l'Algorisme de Cocke–Younger–Kasami (CKY - CYK)

Pràctica de PAA (Programació i Algorismia Avançada)

Nils Duran i Ramon Andreu

1.- Arxius de la Pràctica:

L'arxiu amb la gramàtica: *gramatica_fnc.py*

Arxius amb gramàtiques de prova:

- *g1.txt*
- *g1_prob.txt*
- *g2.txt*
- *g2_prob.txt*

  …
- *g7.txt*

Arxius que executen proves:

- *proves.py*
- *proves2.py*

2.- Format de les Gramàtiques:

Hem decidit que el millor format per la nostra implementació del CKY és la proposada a l'enunciat de la pràctica, ja que ens sembla molt més natural d'aquesta manera. La gramàtica s'ha de posar en un arxiu *txt*. Ara explicarem exactament com ha de ser la gramàtica amb uns exemple:

Exemple en FNC:

S -> AB | CD | CB | SS
A -> BC | a
B -> SC | b
C -> DD | b
D -> BA

Cal recalcar que els espais entre "BC | a" són necessaris per que l'algorisme pugui processar la gramàtica correctament (ho diem per experiència). Un exemple **NO correcte** seria:

S -> a|XA|AX|b
…

Exemple en CFG General:

S -> aSa | bSb | a | b

Com abans, els espais també són necessaris. A més hem de dir que, com ens han ensenyat a l'assignatura de PLH (Processament del Llenguatge Humà), *tota CFG que no generi la seqüència buida, pot convertir-se a FNC*. Nosaltres hem seguit aquesta indicació, però igualment és possible que l'algorisme funcioni correctament amb gramàtiques que continguin la seqüència buida, ja que està preparada per això, però no ho assegurem.

Exemple en CNF/CFG Probabilístic:

S -> a | XA | AX | b	[0.1 0.4 0.4 0.1]
A -> RB	[1]
B -> AX | b | a	[0.5 0.25 0.25]
X -> a	[1]
R -> XB	[1]

Les probabilitats de les regles es posen entre '[]', separades per espais, i sense comes. L'algorisme contempla la possibilitat de que la gramàtica no estigui en CNF i ser probabilística.

3.- L'Algorisme CKY / PCKY:

Per aquesta implementació hem decidit que pot ser una bona idea convertir aquesta gramàtica en una classe, per després poder processar internament la gramàtica si cal, a més de poder implementar els mètodes que siguin necessaris per que així quedi més organitzat i clar.

Encara que la classe està ben comentada i explicada, la comentarem per sobre per que quedi clar que fa cada una de les parts.
El codi és molt llarg així que a l'hora d'explicar una part només posaré la part inicial del codi:

```
class Gramatica_FNC():
    def __init__(self, file, to_fnc = False, pcky = False):
        self.grammar = {}
        self.probabilities = {}
        self.N = {}
        self.Σ = {}

        if pcky == False:
           with open(file) as f:
               for line in f:
                     line     =     re.split(r"\s*->\s*|\s*\|\s*",
line.strip())
                   self.grammar[line[0]] = line[1:]

                   …
```

El *init* de la classe té una funció molt important per l'automatització del processament de la gramàtica, ja que és l'encarregada de llegir els arxius i processar-les segons aquests paràmetres:

- *file (string)*: La direcció del del arxiu, en forma d'*string*.

- *to_fnc (boolean)*: Com l'algorisme CKY només sap operar amb gramàtiques en CNF, cal posar el paràmetre en *True* si la gramàtica NO està en CNF. En cas contrari no cal posar res.

- *pcky (boolean)*: En cas de que la gramàtica dins l'arxiu *file* tingui probabilitats, cal posar *pcky* com a *True*. En cas contrari no cal posar res.

Una vegada s'ha processat la gramatica, què tenim?

Doncs la gramàtica ha estat codificada de la següent manera:

A la variable *N* que s'ha creat al *init*, tindrem un diccionari tal que així:

```
N: {'XA': ['S'], 'AX': ['S', 'B'], 'RB': ['A'], 'XB': ['R']}
```

Aquest diccionari representa les parts NO terminals de la gramàtica, sent les claus els resultats de les regles i els valors les regles que les generen.

A la variable *Σ* que s'ha creat al *init*, tindrem un altre diccionari tal que així:

```
Σ: {'a': ['S', 'B', 'X'], 'b': ['S', 'B']}
```

On les claus del diccionari seran els símbols terminal de la gramàtica, i els seus valors les regles que les generen.

Després, tenim la variable *grammar*, que conté tota la gramàtica en un altre diccionari, aquesta vegada les claus són els símbols que generen altres símbols terminals o no terminals de la gramàtica i els valors aquests últims:

```
{'S': ['AB', 'CD', 'CB', 'SS'], 'A': ['BC', 'a'], 'B': ['SC', 'b'], 'C': ['DD', 'b'], 'D': ['BA']}
```

I, en cas de que la gramàtica sigui probabilista, la variable *probabilities*, no estarà buida i contindrà un diccionari on les claus son els símbols que un altre cop els símbols que generen altres símbols terminals o no terminals de la gramàtica i els valors les probabilitats de generar aquests últims.

```
{'S': [0.3, 0.3, 0.2, 0.2], 'A': [0.7, 0.3], 'B': [0.7, 0.3], 'C': [0.7, 0.3], 'D': [1.0]}
```

Hem decidit treballar amb diccionaris per no haver de recórrer totes les regles de la gramàtica, fent servir les claus per accedir a les produccions directament. Després veurem que això ens permetrà millorar la complexitat temporal de l'algorisme ($O(n^3 \cdot P)$ en lloc de $O(n^3 \cdot G)$).
Amb això que fem? Doncs ara que tenim la gramàtica codificada, ja es pot aplicar algorisme de CKY (o CYK)

```
def CKY_det(self, cadena: str):
        n = len(cadena)
        if n == 0:

            return 'S' in self.grammar and '' in self.grammar['S']

        # Creem la taula triangular superior per el CKY
        taula = [[set() for _ in range(i + 1)] for i in range(n)]

        # Omplim el cas base (línia de sota)
        for i in range(n):
            taula[-1][i].update(self.Σ[cadena[i]])

        # Apliquem l'algorisme CKY
        for length in range(2, n + 1):
            for i in range(n - length + 1):
                for k in range(1, length):
                    for nt in self.N:
                        B, C = nt
                        if B in taula[-k][i] and C in taula[-(length - k)][i + k]:
                            taula[-length][i].update(self.N[nt])

        # print(cadena)
        # self.print_taula(taula)
        return 'S' in taula[-n][0]
```

Aquesta és la nostra versió de l'algorisme, que es basa en programació dinàmica, com es demana a la pràctica. No hi ha res nou ja que aquest algorisme és bastant antic, per tant segueix els passos de manera estàndard, adaptat a les nostres necessitats.
Per explicar una mica més sobre la implementació, només podem dir que la seva complexitat algorítmica és de $O(n^3 \cdot P)$, on P és el nombre de produccions de la gramàtica en CNF de la regla amb més produccions de la gramàtica gràcies al fet que fem servir un diccionari, que per tant, és $O(n^3)$. Pel fet d'usar programació dinàmica, la seva complexitat espacial és de $O(n^2)$.

Ara passem al segon algorisme, també mètode de la classe, el *PCKY*:

```
def CKY_prob(self, cadena: str):
     n = len(cadena)
     if n == 0:
         return float('S' in self.grammar and '' in self.grammar['S'])

     # Creem la taula triangular superior per el CKY
     taula = [[defaultdict(float) for _ in range(i + 1)] for i in range(n)]

     # Omplim el cas base
     for i in range(n):
         for nt in self.Σ[cadena[i]]:
             idx = self.grammar[nt].index(cadena[i])
             taula[-1][i][nt] = self.probabilities[nt][idx]

     # Apliquem l'algorisme CKY
     for length in range(2, n + 1):
         for i in range(n - length + 1):
             for k in range(1, length):
                 for nt in self.N:
                     B, C = nt
                     prob_B = taula[-k][i].get(B, 0)
                     prob_C = taula[-(length - k)][i + k].get(C, 0)
                     if prob_B > 0 and prob_C > 0:
                         for rule in self.N[nt]:
                             idx = self.grammar[rule].index(nt)
                             taula[-length][i][rule] +=
self.probabilities[rule][idx] * prob_B * prob_C

        return taula[-n][0].get('S', 0.0)
```

Aquesta implementació és germana de la que acabem d'explicar, doncs és la mateixa, però també gestiona probabilitats.
La seva complexitat algorítmica és de $O(n^3 \cdot P)$, on P és el nombre de produccions de la gramàtica en CNF, que per tant, és $O(n^3)$. La seva complexitat espacial també és de $O(n^2)$.

Ara passem a la implementació de del mètode que converteix les CFG a CNF, les quals tenim dues versions, una per a gramàtiques no probabilistes i un altre per les que ho són.

Degut a les fases de conversió, el mètode és molt llarg, i per tant el veurem part per part:

```
def CFG_a_CNF_prob(self):
        """
        Transforma la gramàtica probabilista de CFG a CNF.
        """
        símbols_usats = set()
        for regla in self.grammar:
            for elem in self.grammar[regla]:
                símbols_usats.add(regla)
                for literal in elem:
                    símbols_usats.add(literal)
        nt_disponibles = [x for x in
'ωψφχτπξμλκθηζδβΩΨΦΣΠΞΛΘΔΓZYXWVUTSRQPONMLKJIHGFEDCBA' if x not in símbols_usats]

        substitucions = {}  # Clau: símbols antics, Valor: símbols nous
        self.print_grammar()

        # Pas 1: Regles híbrides
        for regla in list(self.grammar):
            for idx in range(len(self.grammar[regla])):
                # Si la regla té més de 2 símbols i algun és terminal
                if len(self.grammar[regla][idx]) >= 2 and any(map(str.islower,
self.grammar[regla][idx])):
                    for símbol in self.grammar[regla][idx]:
                        if símbol.islower():
                            if símbol not in substitucions:
                                # Guardem la substitució per a futur ús en altres
regles (consistència)
                                substitucions[símbol] = nt_disponibles.pop()
                            self.grammar[regla][idx] =
self.grammar[regla][idx].replace(símbol, substitucions[símbol])
                            self.grammar[substitucions[símbol]] = [símbol]
```

Aquesta part com bé diuen els comentaris, comperteixen les regles hibrides (regles amb resultat terminal i no terminal ex: S -> Ab), en regles no hibrides.

```
# Pas 2: Regles unitàries
        for _ in range(len(list(self.grammar))**2):
            for regla in list(self.grammar):
                if regla not in self.grammar:
                    continue
                for idx in range(len(self.grammar[regla])):
                    if self.grammar[regla][idx] in self.grammar and
len(self.grammar[regla][idx]) == 1:
                        clau_tmp = self.grammar[regla][idx]
                        self.grammar[regla].extend(self.grammar[clau_tmp])
```

```
                        self.grammar[regla].remove(clau_tmp)
                        # Eliminar regla unitària (clau regla)
                        del self.grammar[clau_tmp]
                        break
```

Ara el que fa és eliminar les regles no unitàries, un exemple de regla no unitària seria S -> N, N -> a, que el mètode convertirà a S -> a.

En tercer lloc, tenim aquesta part,

```
# Pas 3: Regles de més de 2 símbols no terminals
        for _ in range(len(list(self.grammar))**2):
            for regla in list(self.grammar):
                for idx in range(len(self.grammar[regla])):
                    if len(self.grammar[regla][idx]) > 2:
                        while len(self.grammar[regla][idx]) > 2:
                            if self.grammar[regla][idx][:2] not in substitucions:
                                substitucions[self.grammar[regla][idx][:2]] =
nt_disponibles.pop()

self.grammar[regla].append(substitucions[self.grammar[regla][idx][:2]])

self.grammar[substitucions[self.grammar[regla][idx][:2]]] =
[self.grammar[regla][idx][:2]]
                            self.grammar[regla][idx] =
substitucions[self.grammar[regla][idx][:2]] + self.grammar[regla][idx][2:]
```

Que s'encarrega de gestionar les regles no binàries per convertirles en binàries, un exemple seria, S -> ABC, que el mètode la convertiria en S -> AD i D -> BC.

Ara sí, per acabar, tenim una part que posa totes les transformacions al seu lloc,

```
# Ajustaments finals al diccionaris N i Σ
        self.N = {}
        self.Σ = {}
        for esq, dre in self.grammar.items():
            terminals = [t for t in dre if len(t) == 1 and t.islower()]
            no_terminals = [nt for nt in dre if len(nt) == 2]
            for t in terminals:
                if t not in self.Σ:
                    self.Σ[t] = [esq]
                else:
                    self.Σ[t].append(esq)
            for nt in no_terminals:
                if nt not in self.N:
                    self.N[nt] = [esq]
                else:
                    self.N[nt].append(esq)
```

Ara toca la versió que gestiona les CFG probabilista, que és exactament igual excepte per una part adicional.

```
        new_probabilities = {}
        for lhs, rhs_list in self.grammar.items():
            if lhs in self.probabilities:
                probabilities = self.probabilities[lhs]
            else:
                probabilities = [1] * len(rhs_list)  # Si no hi han probabilitats
definides asignem 1 a les produccions
            new_prob_list = []
            for idx, rhs in enumerate(rhs_list):
                # Calculem la probabilitat per a la nova producció, si n'hi ha una
existent
                if lhs in self.probabilities and idx <
len(self.probabilities[lhs]):
                    new_prob = self.probabilities[lhs][idx] / len(rhs_list)
                else:
                    new_prob = 1 / len(rhs_list)  # Si no hi ha probabilitat
definida, distribuïm igualment
                new_prob_list.append(new_prob)
            new_probabilities[lhs] = new_prob_list

        # Assignem les noves probabilitats al diccionari de probabilitats
        self.probabilities = new_probabilities
```

Bàsicament, assigna les probabilitats de les noves regles per que siguin les que toquen, distribuint-la equitativament.

Per acabar, passem amb l'experimentació amb els arxius que hem mencionat al principi del informe.

Tenim varies gramàtiques que usarem als arxius de proves. Per saber si les respostes són correctes hem usat el RACSO, que donada una gramàtica et diu si una paraula es generada per la mateixa.

Dins del arxiu de *proves.py* hi ha algo semblant a:

```
cnf_grammar = Gramatica_FNC('g1.txt')

    proves_g1 = ['a', 'aa', 'aaa', 'aaaa', 'aaaaa', 'aaaaaaa', 'b', 'bb', 'bbb',
'bbbb', 'bbbbb', 'ab', 'aab', 'aaab', 'aaaab', 'aaaaaab', 'abab', 'aba', 'abaa',
'abaaa', 'abaab', 'bbbaaa', 'aabaaaa']

    labels_g1 = [True, False, False, True, False, True, True, False, False, False,
False, False, False, True, False, True, False, False, True, False, False, False,
True]

    predicted_g1 = []
    for elem in proves_g1:
        predicted_g1.append(cnf_grammar.CKY_det(elem))

    if predicted_g1 == labels_g1:
        print("La gramàtica s'ha identificat corectament!")
    else:
```

```
        print("La gramàtica NO s'ha identificat corectament")
```

Sí que és veritat que no és un mètode molt eficient però com ho hem provat amb moltes gramàtiques, és molt possible que la implementació sigui correcta. Més que mesurar si ho ha bé amb una gramàtica tenint molts exemples per cada una, creiem que si amb poques proves però amb moltes gramàtiques també es posible. Com és d'esperar, ho fa tot bé.

Per acabar, amb aquest projecte hem après a millorar la eficiencia del nostre codi, a més de ser un bon repàs de l'assignatura de PLH.