Analisis Kompleksitas Waktu Pada Algoritma Pengubahan Bahasa *Context-Free Grammar* (CFG) ke Bentuk *Chomsky Normal Form* (CNF)

Kenny Benaya Nathan - 13521023[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521023@std.stei.itb.ac.id

***Abstrak*—Untuk membuat suatu program dalam komputer, seorang *user* akan memberikan instruksi kepada komputer apa yang harus dilakukan dan apa yang tidak boleh dilakukan. Instruksi ini diberikan dalam sekumpulan kode bahasa manusia yang masih bisa dimengerti oleh manusia. Kode tersebut kemudian diubah ke dalam bahasa komputer dengan menggunakan *compiler*. Salah satu fase yang dikerjakan oleh *compiler* adalah fase *parsing*. *Parsing* bisa dilakukan dengan beberapa algoritma, salah satunya adalah algoritma Cocke-Younger-Kasami (CYK) yang memerlukan *grammar* Chomsky Normal Form (CNF) sebagai bahasa masukannya. Namun, cara paling cocok untuk membuat aturan suatu *grammar* (dalam hal ini *grammar* bahasa pemrogramman) adalah dengan *Context-Free Grammar* (CFG). Untuk itu, penulis mencoba untuk mengimplementasikan algoritma yang efisien untuk mengkonversi bentuk CFG ke dalam bentuk CNF dengan menghitung kompleksitas waktu dari pengimplementasian tersebut. Hasilnya, program yang diimplementasikan penulis tidak begitu efisien karena membutuhkan waktu kuadratik relatif terhadap jumlah masukan aturan produksi CFG dan linear terhadap jumlah aturan produksi unit.**

***Keywords*—*Context-Free Grammar* (CFG), *Chomsky Normal Form* (CNG), kompleksitas waktu, *Big-O Notation* (O(n))**

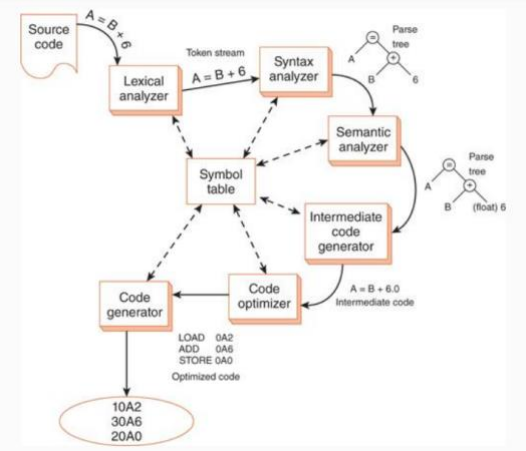
I. Introduction

Pada zaman revolusi industri 4.0 ini, hidup seluruh manusia sudah bergantung pada teknologi. Dunia digital sudah berkembang dengan sangat pesat sehingga memengaruhi kehidupan seluruh manusia, mulai dari pekerjaan, sekolah, pemaparan informasi, hingga kehidupan sehari-hari. Tanpa teknologi digital, tentu kehidupan manusia tidak akan berjalan dengan lebih mudah dan lebih efisien daripada sekarang ini. Salah satu profesi yang menjaga agar teknologi digital tetap berjalan dan bisa berkembang adalah seorang *programmer*.

Seorang programmer bekerja di belakang layar untuk mengembangkan sebuah *software*/perangkat lunak. Untuk membuat suatu program, semua cara kerja, alur, dan instruksi tersebut dibuat dalam sekumpulan kode.

Kode-kode tersebut adalah bahasa manusia yang dibuat sedemikian rupa sehingga masih mudah untuk dipahami oleh manusia di mana masih bisa untuk dikomunikasikan dengan komputer. Namun, kode tersebut masih dalam bentuk bahasa manusia. Jadi, bagaimana caranya komputer mampu menjalankan instruksi dari bahasa manusia? Jawabannya adalah dengan *compiler*.

Inti utamanya, seorang *user* hanya perlu untuk memberi tahu komputer apa yang harus dilakukan dan tidak dilakukan dalam kode bahasa *programming*. *Compiler* akan melakukan sisanya. Beberapa langkah yang dilakukan oleh sebuah *compiler* adalah memproses kode tersebut untuk dilihat apakah kalimat-kalimat kode dalam program tersebut sudah layak pakai, di mana mengikuti aturan bahasa kode (*syntax*) tersebut. Langkah-langkah tersebut disebut dengan *parsing* [1 p. 1179].



Gambar 1.1 Enam fase untuk *compile* sebuah program
(Sumber: *The Essentials of Computer Organization and Architecture p.1178*)

Ada beberapa *grammar* bahasa dan algoritma *parser* yang sudah ada untuk melakukan *compile*. Salah satunya adalah *Context-Free Grammar* (CFG) dan algoritma *Cocke-Younger-Kasami* (CYK) untuk algoritma *parser*-nya. Namun, masukan dari algoritma membutuhkan *grammar* yang berbeda, yaitu *Chomsky Normal Form* (CNF). Maka dari itu, diperlukanlah sebuah program untuk mengkonversi CFG menjadi *grammar* CNF agar bisa dimasukan ke dalam algoritma CYK. Program tersebut akan menjadi dasar dari pokok permasalahan yang akan dianalisis pada makalah ini. Proses Analisis ini akan menentukan seberapa efektif algoritma ini.

II. LANDASAN TEORI

A. Context-Free Grammar

Dalam teori bahasa formal, *Context-Free Grammar* (CFG) adalah sebuah bentuk matematika untuk mendeskripsikan aturan-aturan dalam sebuah bahasa. CFG memungkinkan untuk mengekspresikan bahasa yang kompleks menjadi sekumpulan aturan yang lebih sedarhana [2 p. 173]. Secara formal, CFG didefinisikan sebagai $G = \{V, T, S, P\}$ di mana

- V = Kumpulan simbol non-*terminal* (variabel)
- T = Kumpulan simbol *terminal*
- S = Simbol *Start*
- P = Aturan produksi

CFG menentukan apa yang boleh dan tidak boleh ada dalam sintaks suatu bahasa dengan menggunakan kumpulan aturan produksi. Aturan-aturan tersebut memberikan bagaimana simbol tersebut dapat dibentuk dari simbol-simbol lainnya.

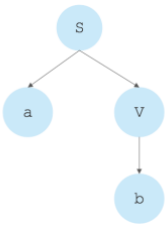
$$A \rightarrow X \tag{1}$$

Aturan produksi dalam CFG memiliki bentuk dalam persamaan (1). Persamaan tersebut kemudian didefinisikan lagi dalam persamaan (2).

$$X = \{V \cup T\}^* \text{ and } A \in V \tag{2}$$

Aturan produksi terdiri dari sebuah simbol non-*terminal* (variabel) yang akan didefinisikan di bagian kiri. Sementara itu, pendefinisian dari variabel di bagian kiri merupakan kumpulan variabel dan simbol *terminal.* Variabel adalah simbol yang masih dapat didefinisikan oleh aturan produksi lainnya, sedangkan *terminal* adalah simbol yang tidak bisa didefinisikan lagi oleh aturan produksi lainnya. Artinya, *terminal* merupakan bagian dari bahasa itu sendiri. Aturan-aturan produksi itu dapat dilakukan berulang untuk menggantikan simbol variabel dengan aturan-aturan lainnya.

CFG memiliki sebuah simbol *Start* untuk mulai menjelaskan bahasa yang diwakili melalui kumpulan-kumpulan aturan produksi tersebut.



Gambar 2.1 Salah satu contoh representasi aturan produksi dalam sebuah tree (Sumber: dokumen pribadi)

Aturan produksi dalam CFG dapat direpresentasikan dalam sebuah tree, seperti pada gambar 2.1. CFG dimulai dari simbol *Start* (S) yang memiliki aturan produksi $S \rightarrow aV$ dan $V \rightarrow b$, di mana V merupakan sebuah variabel yang masih memiliki aturan produksi lainnya. Berbeda dengan simbol a dan b yang tidak memiliki aturan lagi sehingga bisa disebut sebagai simbol terminal.

CFG disebut sebagai *Context-Free* karena aturan produksi yang digunakan dapat dibuat tanpa memerlukan informasi tentang konteksnya untuk dapat diterapkan. Artinya, aturan produksi dalam CFG bisa diterapkan pada setiap simbol variabel dalam sebuah kalimatnya, tanpa memedulikan di mana simbol itu berada dalam kalimat tersebut.

B. Chomsky Normal Form

Chomsky Normal Form (CNF) merupakan salah satu bentuk spesial dari CFG yang lebih sederhana. Karena bentuknya yang sederhana, CNF sangat bermanfaat karena mampu untuk mengekspresikan sebuah bahasa yang kompleks dengan aturan produksi yang lebih sederhana dan mudah untuk melakukan proses penganalisisan p.261]. Sebuah aturan CFG dapat dikatakan dalam bentuk CNF jika dan hanya jika:

- Hanya memiliki 1 terminal di bagian kanan. Contoh pada persamaan (3)
$$A \rightarrow a \tag{3}$$
- Hanya memiliki 2 variabel di bagian kanan. Contoh pada persamaan (4)
$$A \rightarrow BC \tag{4}$$
- Memiliki simbol *null* hanya pada pendefinisian *Start.* Contoh pada persamaan (5)
$$S \rightarrow \varepsilon \tag{5}$$

Sesuai dengan namanya, Avram Noam Chomsky adalah orang yang pertama kali mengusulkan bahwa CFG merupakan sebuah cara yang bisa dilakukan untuk mendeskripsikan bahasa natural dan berhasil membuktikan bahwa CFG dapat disederhanakan ke dalam bentuk yang kemudian kita sebut sebagai CNF. Langkah-langkah untuk mengonversi CFG ke dalam CNF adalah sebagai berikut:

1. Jika ada sebuah simbol *start* dalam aturan manapun di bagian kanan, maka kita akan membuat simbol *start* baru beserta aturan baru yang menghasilkan simbol *start*. Contohnya $S0 \rightarrow S$.
2. Eliminasi seluruh *Null productions,* aturan produksi yang menghasilkan simbol *null,* dengan menggunakan *Null production removal algorithm*
3. Eliminasi seluruh *Unit productions,* aturan produksi yang hanya menghasilkan 1 variabel, dengan menggunakan *Unit production removal algorithm*. Contohnya $A \rightarrow B$.
4. Ubah seluruh aturan produksi $A \rightarrow B_1 \dots B_n$ di mana $n > 2$, dengan variabel baru seperti $A \rightarrow B_1 C$ di mana $C \rightarrow B_2 \dots B_n$. Ulangi hingga seluruh aturan produksinya hanya menghasilkan 2 simbol.
5. Ubah semua aturan produksi yang menghasilkan 1 terminal dan 1 variabel seperti $A \rightarrow aB$, dengan variabel baru seperti $A \rightarrow YB$ di mana $Y \rightarrow a$.

C. Kompleksitas Waktu

Sebuah algoritma yang baik tidak hanya melihat apakah program tersebut berjalan atau tidak, melainkan juga melihat tingkat efisiensinya, mulai dari waktu yang diperlukan hingga ruang memori yang dibutuhkan relatif terhadap ukuran masukan (n) yang diproses untuk mengeksekusi program tersebut.

Semakin kompleks algoritma, semakin besar pula waktu dan ruang yang dibutuhkan. kompleksitas algoritma yang ditinjau dari seberapa besar waktu yang digunakan, disebut kompleksitas waktu yang biasanya dilambangkan sebagai T(n).

Tingkat kompleksitas waktu dari sebuah algoritma dapat diukur dari jumlah tahapan operasi primitif yang dilakukan dalam sebuah fungsi/algoritmanya. Beberapa jenis operasi tersebut meliputi operasi baca (input), operasi tulis (output), operasi aritmetika (+, -, *, /), operasi pengisian nilai (asignment), operasi perbandingan, operasi pengaksesan elemen, pemanggilan fungsi, alokasi memori, dan lain-lain.

Akan tetapi, akan menjadi sangat rumit jika kita menghitung banyaknya operasi yang dilakukan, sehingga lebih baik kita mengukur laju perubahan kebutuhan waktu sebuah algoritma jika masukan yang diberikan meningkat. Selain itu, untuk menyederhanakan hitungan, hal yang bisa dilakukan adalah dengan menghitung jumlah operasi khas yang mendasari algoritma tersebut, tidak perlu seluruh operasi yang ada. Contoh dari operasi khas tersebut seperti algoritma pencarian (*searching*), pengurutan (*sorting*), perkalian dua matriks, perhitungan polinom, dan sebagainya.

Untuk jumlah masukan n yang besar, bisa menggunakan notasi kompleksitas waktu asimptotik. Notasi ini dapat melihat secara lebih jelas bagaimana kebutuhan waktu sebuah algoritma tumbuh seiring meningkatnya ukuran masukan n. Salah satu notasi kompleksitas waktu asimptotik adalah notasi O-besar (*Big-O Notation*).

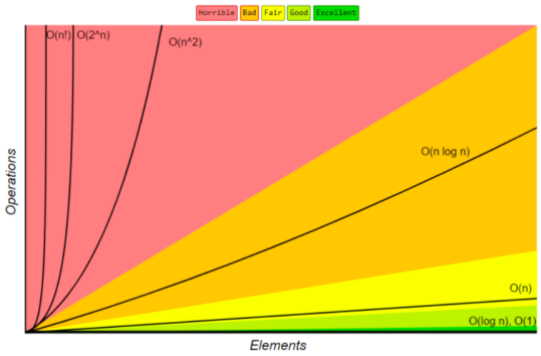
Big-O adalah sebuah sebuah konsep matematis yang bisa dipakai untuk menghitung tingkat efetivitas dari sebuah data struktur dan algoritma. Notasi *Big-O*, yaitu O(n), menyatakan berapa langkah yang dilakukan oleh sebuah algoritma jika terdapat masukan sebanyak n buah elemen. Lebih tepatnya, seberapa besar dan bagaimanakah keefektifan sebuah algoritma berubah seiring bertambahnya jumlah elemen data. Notasi *Big-O* dapat membantu kita untuk menemukan ataupun membandingkan antar 2 algoritma, algoritma mana yang paling cepat untuk digunakan. [3 p. 36]. Secara formal, *Big-O Notation* dapat didefinisikan dalam persamaan (6) sebagai

$$T(n) = O(f(n)) \tag{6}$$

Jika terdapat sebuah konstanta C dan n_0 sedemikian rupa sehingga $T(n) \leq Cf(n)$ untuk $n \geq n_0$, maka f(n) dapat dianggap sebagai batas atas/maksimal dari T(n) karena berorde paling besar f(n) untuk nilai n yang besar.

TABEL I.
PENGELOMPOKAN ALGORITMA BERDASARKAN *BIG-O NOTATION*

Big-O Notation	Nama
O(1)	Konstan
O(log n)	Logaritmik
O(n)	Linier
O(n log n)	Linier logaritmik
$O(n^2)$	Kuadratik
$O(n^3)$	Kubik
$O(2^n)$	Eksponensial
O(n!)	Faktorial



Gambar 2.2 Grafik efisiensi beberapa big-o notation
(Sumber: *bigocheatsheet.com*)

Dapat dilihat dari gambar 2.2, algoritma dengan kompleksitas waktu asimptotik dengan orde yang lebih kecil adalah algoritma yang lebih efisien.

III. HASIL PENELITIAN

A. Implementasi Algoritma Konversi CFG ke CNF

Penulis sudah mengimplementasikan algoritma dari pengubahan bentuk CFG ke dalam bentuk CNF dengan beberapa modifikasi. Program yang dibuat menggunakan bahasa Python. Hasil program dari pengimplementasian algoritma tersebut dapat dilihat pada gambar 3.1.

(a)

```python
import os

rules = {}
# READ AND WRITE ----------------------
# READ CFG.TXT
def readCFG(file):
    path = os.getcwd()
    with open(path + '/' + file) as cfg:
        row = cfg.readlines()
        rowConverted = []
        for i in range(len(row)):
            rowConverted.append(row[i].replace("->", "").split())
    return rowConverted

# WRITE CNF.TXT
def writeGrammar(grammars):
    path = os.getcwd()
    cnf = open(path + '/' + 'cnf.txt', 'w')
    for rule in grammars:
        cnf.write(rule[0])
        cnf.write(" -> ")
        for i in rule[1:]:
            cnf.write(str(i))
            cnf.write(" ")
        cnf.write("\n")
    cnf.close()

# MAIN ----------------------
# List of grammar
def addGrammar(grammar):
    global rules
    if grammar[0] not in rules:
        rules[grammar[0]] = []
    rules[grammar[0]].append(grammar[1:])

```

(b)

```
# CFG to CNF
def convertToCNF(grammars):
    global rules
    foundS = False
    unit = []                   # Unit Production Rule
    result = []                 # CNF Result
    idx = 0                     # New variable
    for grammar in grammars:
        newGrammars = []        # Temporary list of new grammars (with 2 elements)

        # Start symbol in the right hand side
        if ('S' in grammar[1:]) and not foundS:
            grammars.append(["S'", "S"])
            foundS = True

        # 1 variable
        if len(grammar) == 2 and not grammar[1].islower():
            unit.append(grammar)
            addGrammar(grammar)
            continue

        # more than 2 variables
        while len(grammar) > 3:
            newGrammars.append([grammar[0], grammar[1]] + [f"{grammar[0]}{str(idx)}"])
            grammar = [f"{grammar[0]}{str(idx)}"] + grammar[2:]
            idx += 1

        # terminal inside grammar (2 elements in the right hand side)
        if len(grammar) == 3:
            if grammar[1].islower():
                newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[1]])
                grammar = [grammar[0], f"{grammar[0]}{str(idx)}", grammar[2]]
                idx += 1
                if grammar[2].islower():
                    newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[2]])
                    grammar = [grammar[0], f"{grammar[0]}{str(idx-1)}", f"{grammar[0]}{str(idx)}"]
                    idx += 1

            if grammar[2].islower():
                newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[2]])
                grammar = [grammar[0]] +  grammar[1] + [f"{grammar[0]}{str(idx)}"]
                idx += 1

        if grammar:
            addGrammar(grammar)
            result.append(grammar)

        if newGrammars:
            for i in range(len(newGrammars)):
                addGrammar(newGrammars[i])
                result.append(newGrammars[i])

    # Processing unit production rule
    while unit:
        grammar = unit.pop()
        if grammar[1] in rules:
            for value in rules[grammar[1]]:
                # Replace variable with it's own rule
                newGrammar = [grammar[0]] + value

                # 1 terminal or 2 variables
                if len(newGrammar) > 2 or newGrammar[1].islower():
                    result.append(newGrammar)

                # if still 1 variable, then loop it ; final state = 1 terminal or 2 variables
                else:
                    unit.append(newGrammar)
                addGrammar(newGrammar)
    return result
```

Gambar 3.1 pengimplementasian algoritma konversi CFG ke CNF: (a) fungsi pembantu (b) fungsi utama
(Sumber: dokumen pribadi)

Secara garis besarnya, tahapan dari program di atas adalah

1) Program membaca file CFG yang dimasukan oleh *user* pada *current working directory*, nama file tergantung nama yang dimasukan oleh *user*.
2) Program menjalankan algoritma utama konversi CFG ke CNF.
3) Program menulis hasil dari konversi tersebut ke dalam file "cnf.txt" pada *current working directory* sekarang.

Program ini memuat empat fungsi, yaitu `readCFG(file)`, `writeGrammar(grammars)`, `addGrammar(grammar)`, dan `convertToCNF(grammars)`. Contoh cara untuk menjalankan program ini adalah dengan menjalankan perintah `writeGrammars(readCFG("cfg.txt")))` pada terminal. Program ini juga meng-*import library* os untuk mengambil *path current working directory* dari user untuk membaca dan menulis file dari user. Penulis akan menganalisis kompleksitas waktu dari masing-masing fungsi di atas.

1) readCFG(file)

Fungsi `readCFG` berfungsi untuk membaca file yang dimasukan oleh *user*. Parameter yang diambil oleh fungsi ini adalah nama file yang kemudian akan dimasukkan ke dalam variabel `file`. Contoh untuk menjalankannya adalah dengan menjalankan perintah readCFG("cfg.txt").

```
def readCFG(file):
    path = os.getcwd()
    with open(path + '/' + file) as cfg:
        row = cfg.readlines()
        rowConverted = []
        for i in range(len(row)):

rowConverted.append(row[i].replace("->",
"").split())
```

Figur 1 implementasi fungsi `readCFG`
(Sumber: dokumen pribadi)

Analisis kompleksitas waktu yang sesuai secara bertahap pada figur (1):

- Pertama, program tersebut membuka file seperti pada figur (2) yang diberikan dan membaca semua barisnya dan dimasukan ke dalam variabel row. Kemudian, setiap elemen pada *list* row akan dimasukkan ke dalam *list* `rowConverted`, di mana sudah memisahkan komponen lain sehingga hanya akan menghasilkan satu karakter per elemen dan tidak ada elemen lain selain karakter huruf. Ini akan mengambil waktu linear atau O(n) di mana n adalah jumlah baris dalam file yang dibaca.

```
S -> A S A
S -> a B
A -> B
A -> S
B -> b
```

Figur 2 Contoh file yang akan dibaca pada file "cfg.txt"
(Sumber: dokumen pribadi)

- Selanjutnya, program tersebut mengulang sebanyak jumlah baris dalam file yang dibaca. Setiap iterasi dari perulangan tersebut akan memproses baris tersebut dengan menghilangkan karakter "->" dan memisahkan elemen-elemen dari baris tersebut menjadi sebuah list. Ini juga akan mengambil waktu konstan atau O(1) untuk setiap baris yang diproses.
- Pada akhirnya, program ini akan mengembalikan sebuah variabel `rowConverted` dengan tipe *list of list* yang berisi hasil dari setiap baris yang sudah dipisahkan per elemen untuk membentuk sebuah aturan produksi dalam sebuah *list*.

Secara keseluruhan, kompleksitas waktu dari fungsi readCFG adalah linear atau O(n), di mana n merupakan jumlah baris yang dimasukan dalam *file user*.

2) writeGrammar(grammars)

Fungsi `writeGrammar` berfungsi untuk menuliskan hasil dari konversi CFG ke CNF dalam file "cnf.txt". yang dimasukan oleh *user*. Parameter yang diambil oleh fungsi ini adalah sebuah variabel bertipe *list of list* yang berisi aturan produksi akhir dalam bentuk CNF. Variabel tersebut kemudian dimasukkan ke dalam variabel lokal `grammars` dalam fungsi ini.

```
def writeGrammar(grammars):
    path = os.getcwd()
    cnf = open(path + '/' + 'cnf.txt', 'w')
    for rule in grammars:
        cnf.write(rule[0])
        cnf.write(" -> ")
        for i in rule[1:]:
            cnf.write(str(i))
            cnf.write(" ")
        cnf.write("\n")
    cnf.close()
```

Figur 3 implementasi fungsi `writeGrammar`
(Sumber: dokumen pribadi)

Analisis kompleksitas waktu yang sesuai secara bertahap pada figur (3):

- Program akan mengiterasi setiap *list* atau setiap aturan produksi, kemudian menuliskan setiap aturan tersebut ke dalam *file* "cnf.txt" dengan format yang mirip dengan *file* "cfg.txt" sebelumnya. Bagian ini akan memakan waktu linear atau O(n) di mana n merupakan jumlah aturan produksi.
- Kemudian, program akan menulis setiap token aturan ke dalam *file* "cnf.txt". Bagian ini akan memakan waktu konstan atau O(1) per token.

Secara keseluruhan, kompleksitas waktu dari fungsi `writeGrammar` adalah linear atau O(n) sesuai dengan jumlah aturan produksi hasil dalam bentuk CNF.

3) addGrammar(grammar)

Fungsi `addGrammar` berfungsi untuk memasukkan sebuah aturan produksi ke dalam variabel *dictionary* `rules`, yaitu, daftar isi dari seluruh aturan produksi yang ada. Parameter yang diambil oleh fungsi ini adalah sebuah *list* yang isinya adalah sebuah aturan produksi. Variabel tersebut kemudian dimasukkan ke dalam variabel lokal dalam fungsi ini yang bernama `grammar`.

```
def addGrammar(grammar):
    global rules
    if grammar[0] not in rules:
        rules[grammar[0]] = []
    rules[grammar[0]].append(grammar[1:])
```

Figur 4 implementasi fungsi `addGrammar`
(Sumber: dokumen pribadi)

Analisis kompleksitas waktu yang sesuai secara bertahap pada figur (3):

- Program akan memeriksa apakah simbol variabel pada bagian kiri sudah ada dalam *dictionary* `rules`. Jika tidak ada, maka program akan membuat sebuah *key* baru, yaitu simbol variabel itu sendiri. Langkah ini akan memakan waktu konstan atau O(1).
- Terakhir, program akan memasukkan sisa dari isi list tersebut (elemen pada bagian kanan) ke dalam sebuah *value* bertipe *list* dari *key* simbol variabel bagian kiri. Langkah ini juga memakan waktu konstan atau O(1).

Karena tidak adanya iterasi atau pengulangan seperti pada fungsi-fungsi sebelumnya, maka operasi yang dilakukan hanya pada sebuah elemen dalam masukan *user*, sehingga fungsi ini hanya memakan waktu konstan atau O(1) terhadap jumlah aturan produksi.

4) convertToCNF(grammars)

Fungsi `convertToCNF` merupakan fungsi utama yang memuat seluruh algoritma konversi bentuk CFG ke CNF. Fungsi ini menerima variabel bertipe *list of list* yang berisi aturan-aturan produksi yang ada. Variabel ini kemudian dimasukkan ke dalam variabel lokal `grammars`.

Implementasi dari fungsi ini bisa dilihat pada figur (5) Ada beberapa langkah yang dimodifikasi dari Langkah yang teoretis agar mampu menjalankan program seefisien mungkin.

a) Program menginisiasi variabel Boolean `foundS` bernilai "False", *list of list* `unit`, *list of list* `result`, dan integer `idx` bernilai 0.
b) Program mengiterasi setiap aturan produksi (`grammar`) dalam `grammars` dan menginisiasi *temporary list of list* `newGrammars`.
c) Jika menemukan simbol S pada bagian kanan aturan, maka program akan menambahkan aturan baru di mana $S' \rightarrow S$ dan mengubah foundS menjadi bernilai "True", sehingga kondisi ini tidak akan terakses lagi.
d) Jika pada aturan hanya ada 1 variabel, maka aturan tersebut akan dimasukkan ke dalam `unit` dan menjalankan fungsi `addGrammar`.
e) Selama aturan memiliki lebih dari 2 elemen
    a. Program akan melakukan subtitusi terhadap elemen kedua dan seterusnya. Variabel `grammar` akan diganti dengan aturan subtitusi tersebut, dengan bagian kanannya adalah elemen kedua dan seterusnya
f) Jika ada elemen terminal di aturan bagian kanan dalam `grammar` yang panjangnya sudah menjadi 2 elemen di bagian kanan, maka program akan melakukan subtitusi kembali, sehingga hasilnya menjadi 2 variabel di bagian kanan.
g) Program akan menjalankan fungsi `addGrammar` dan menambahkan *list* `result` dengan masukan `grammar` yang sekarang sudah memenuhi.
h) Jika masih ada elemen di dalam *temporary list* `newGrammars`, program akan menjalankan fungsi `addGrammar` dan menambahkan *list* `result` dengan masukan setiap elemen `newGrammars`.
i) Keluar dari iterasi
j) Selama masih ada elemen dalam `list`, program akan melakukan subtitusi elemen variabel bagian kanan. Pada akhirnya, setiap elemen akan memenuhi syarat

dari aturan CNF. Jika subtitusi gagal, maka program akan kembali memasukkan aturan ini ke dalam list `unit` sehingga nanti akan diulangi lagi proses ini.

```
def convertToCNF(grammars):
    global rules
    foundS = False
    unit = []                   # Unit Production Rule
    result = []                 # CNF Result
    idx = 0                     # New variable
    for grammar in grammars:
        newGrammars = []        # Temporary list of new grammars (with 2 elements)

        # Start symbol in the right hand side
        if ('S' in grammar[1:]) and not foundS:
            grammars.append(["S'", "S"])
            foundS = True

        # 1 variable
        if len(grammar) == 2 and not grammar[1].islower():
            unit.append(grammar)
            addGrammar(grammar)
            continue

        # more than 2 variables
        while len(grammar) > 3:
            newGrammars.append([grammar[0], grammar[1]] + [f"{grammar[0]}{str(idx)}"])
            grammar = [f"{grammar[0]}{str(idx)}"] + grammar[2:]
            idx += 1

        # terminal inside grammar (2 elements in the right hand side)
        if len(grammar) == 3:
            if grammar[1].islower():
                newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[1]])
                grammar = [grammar[0], f"{grammar[0]}{str(idx)}", grammar[2]]
                idx += 1
                if grammar[2].islower():
                    newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[2]])
                    grammar = [grammar[0], f"{grammar[0]}{str(idx-1)}", f"{grammar[0]}{str(idx)}"]
                    idx += 1

            if grammar[2].islower():
                newGrammars.append([f"{grammar[0]}{str(idx)}", grammar[2]])
                grammar = [grammar[0]] +  grammar[1] + [f"{grammar[0]}{str(idx)}"]
                idx += 1

        if grammar:
            addGrammar(grammar)
            result.append(grammar)

        if newGrammars:
            for i in range(len(newGrammars)):
                addGrammar(newGrammars[i])
                result.append(newGrammars[i])

    # Processing unit production rule
    while unit:
        grammar = unit.pop()
        if grammar[1] in rules:
            for value in rules[grammar[1]]:
                # Replace variable with its own rule
                newGrammar = [grammar[0]] + value

                # 1 terminal or 2 variables
                if len(newGrammar) > 2 or newGrammar[1].islower():
                    result.append(newGrammar)

                # if still 1 variable, then loop it ; final state = 1 terminal or 2 variables
                else:
                    unit.append(newGrammar)
                addGrammar(newGrammar)
    return result
```

Figur 5 implementasi fungsi `convertToCNF`
(Sumber: dokumen pribadi)

Untuk perhitungan kompleksitas waktu dari fungsi `convertToCNF` akan menjadi sedikit berbeda karena fungsi ini bergantung pada beberapa data, seperti jumlah aturan produksi yang ada (`grammar`), jumlah elemen pada `newGrammars`, jumlah elemen pada `unit`, beserta dengan panjang dari aturan produksi tersebut.

Operasi khas pertama yang kita temukan adalah `for grammar in grammars`. Operasi ini akan mengakses setiap elemen *list* dari *list of list* `grammars` kemudian dimasukkan ke dalam variabel lokal `grammar`. Langkah ini membutuhkan waktu linear atau O(n) bergantung pada jumlah elemen dari *list* `grammars`.

Kemudian, di dalam iterasi luar tersebut, kita akan menemukan banyak operasi kondisional. Seluruh operasi kondisional tersebut hanya membutuhkan waktu konstan atau O(1), kecuali 1 kondisional, yaitu `if newGrammars`.

Operasi kondisional tersebut memiliki iterasi lagi di dalamnya yang mengakses setiap elemen di dalam *list* `newGrammars`. Bisa disimpulkan bahwa operasi ini membutuhkan waktu linear atau O(m). Di sini, variabel yang digunakan bukanlah n seperti biasanya, karena operasi ini bergantung pada elemen di dalam `newGrammars` yang jelas bisa berbeda dengan jumlah elemen di dalam `grammars`. Namun, ada kemugkinan bahwa jumlah elemen pada `newGrammars` bisa saja sama dengan jumlah elemen pada `grammars`. Oleh karena itu, bisa didapatkan bahwa operasi kondisional ini membutuhkan waktu linear atau O(n) relatif dengan jumlah elemen dari `grammars`.

Pada iterasi luar, bisa ditemukan adanya operasi 'while'. Operasi ini melakukan iterasi sebanyak panjang dari aturan produksi yang diakses saat ini, sesuai dengan kondisi 'while' yang diinstruksikan. Maka dari itu, operasi 'while' ini memakan waktu linear atau O(p) di mana p merupakan panjang elemen pada aturan produksi yang saat ini diakses.

Sebenarnya, panjang elemen pada aturan produksi memiliki nilai yang sangat kecil. Sangat kecil sekali kemungkinannya panjang elemen pada aturan produksi bisa menyamai jumlah aturan produksi CFG yang dilampirkan (jika jumlah aturan produksi yang diberikan dalam jumlah yang sangat besar). Sebanyak-banyaknya, panjang elemen pada aturan produksi tidak akan memengaruhi kompleksitas waktu dari algoritma ini. Oleh karena itu, kita dapat menghiraukan nilai p sehingga menjadi nilai konstan dan akhirnya mendapatkan nilai O(1) untuk operasi 'while'.

Setelah keluar dari iterasi `grammars`, ada operasi 'while' yang mengiterasi seluruh elemen pada list `unit`. Karena itulah, dapat disimpulkan bahwa iterasi `unit` ini membutuhkan waktu linear atau O(q) relatif terhadap jumlah elemen pada `unit`. Di dalam iterasi `unit`, masih ada iterasi lagi yang mengakses *value* dari *key* yang diinginkan pada *dictionary* `rules`. *Value* pada `rules` merupakan *list of list* lagi, di mana setiap *key* (variabel bagian kanan) bisa memiliki beberapa aturan produksi yang direpresentasikan oleh *value.* Oleh karena itu, bisa didapatkan bahwa operasi iterasi `unit` membutuhkan waktu kuadratik atau $O(r^2)$ relatif terhadap panjang dari *dictionary* `rules`. *Big-O Notation* dari iterasi `rules` bisa juga direpresentasikan sebagai $O(n^2)$ karena panjang `rules` bisa disamakan dengan jumlah aturan produksi pada `grammars`.

Fungsi `addGrammar` memiliki kompleksitas waktu sebesar O(1). Fungsi ini tidak akan terlalu memengaruhi performa algoritma ini.

Untuk menghitung hasil akhir *Big-O notation* dari fungsi `convertToCNG` ini, dimulai dengan menjumlahkan seluruh *Big-O* pada iterasi dalam, kemudian mengalikannya dengan iterasi bagiannya, dan akhirnya jumlahkan seluruh *Big-O* pada indentasi pertamanya.

$$T(n) = O(n * \big(O(n) + O(n)\big) + O(n^2) \quad (7)$$

$$T(n) = O\big(n * O(n)\big) + O(n^2) \quad (8)$$

$$T(n) = O(n^2) + O(n^2) = O(n^2) \quad (9)$$

Dari perhitungan pada persamaan (7), (8), dan (9), didapatkan bahwa kompleksitas waktu dari fungsi `convertToCNF` adalah kuadratik atau $O(n^2)$ relatif terhadap jumlah aturan produksi yang ada.

Sebelumnya, dari operasi 'while' yang mengiterasi seluruh elemen `unit`, didapatkan bahwa jumlah elemen unit memengaruhi algoritma konversi CFG ke CNF ini. Jumlah elemen unit ini tidak bisa disamakan dengan jumlah aturan produksi yang dimasukan. Apabila jumlahnya sama, maka akan terjadi *infinite loop* karena program tidak bisa mendapatkan aturan produksi yang jika disubtitusi hasilnya akan memenuhi syarat CNF. Hal ini dikarenakan seluruh aturan produksi dalam `grammars` hanya memiliki satu variabel dan tidak memiliki 2 variabel. Oleh karena itu hasil akhir dari kompleksitas waktu dari fungsi `convertToCNF` adalah kuadratik atau $O(n^2)$ relatif terhadap jumlah aturan produksi dan linear atau O(q) relatif terhadap jumlah aturan produksi unit.

$$T(n, q) = O(n^2) + O(q), q \neq n \quad (10)$$

B. Hasil Kompleksitas Waktu Algoritma

Program utama dari algoritma ini meliputi fungsi `writeGrammar`, `convertToCNF`, dan `readCFG`. Untuk menjalankan program ini bisa dilihat dari figur (6)

```
writeGrammar(converToCNF(readCFG("filename.txt")))
```

Figur 6 Menjalankan fungsi-fungsi utama dari algoritma konversi CFG ke CNF (Sumber: dokumen pribadi)

Dengan menjumlahkan seluruh *Big-O* dari fungsi-fungsi tersebut sesuai dengan persamaan (12), didapatkanlah hasil akhir kompleksitas waktu dari konversi CFG ke CNF, yaitu kuadratik relatif teradap jumlah seluruh aturan produksi yang dimasukan dan linear terhadap jumlah aturan produksi unit dan tidak sama dengan jumlah aturan produksi.

$$T(n, q) = O(n) + O(n) + O(n^2) + O(q), q \neq n \quad (11)$$

$$T(n, q) = O(n^2) + O(q), q \neq n \quad (12)$$

IV. Simpulan dan Saran

Dari hasil pengimplentasian hingga pengujian yang telah dibuat diperoleh simpulan sebagai berikut,

i. besar kompleksitas waktu (*Big*-O) pada pengimplementasian algoritma konversi CFG ke CNF dihasilkan pada persamaan (12), yaitu kuadratik terhadap jumlah seluruh aturan produksi CFG yang dimasukan dan linear terhadap jumlah aturan produksi unit yang jumlahnya tidak sama dengan seluruh aturan produksi.
ii. dengan merujuk pada gambar 2.2, pengimplementasian algoritma ini yang dilakukan oleh penulis termasuk dalam tingkat program yang kurang efisien karena memakan waktu yang cukup banyak seiring dengan bertambahnya jumlah aturan produksi yang dimasukan.

Dari penelitian ini juga, dapat diberikan beberapa saran sebagai berikut,

i. proses *parsing* dapat dilakukan dengan algoritma lain yang lebih efisien, karena algoritma CYK memerlukan tahapan sebelum mencapai algoritma intinya yang kurang efisien, yaitu tahap mengkonversi bentuk CFG ke dalam bentuk CNF
ii. untuk penelitian selanjutnya yang berkaitan dengan konversi bentuk CFG ke dalam bentuk CNF, buatlah implementasi sesederhana mungkin agar tidak memakan waktu yang banyak untuk masukan aturan produksi yang banyak

Ucapan Terima Kasih

Segala Puji dan syukur diucapkan kepada Tuhan Yang Maha Esa atas berkat dan rahmat-Nya sehingga makalah ini dapat diselesaikan dengan baik. Tidak lupa kepada Orang tua penulis yang selalu memberikan semangat dukungannya selama proses pendidikan penulis. Kepada Yang terhormat Dr. Ir. Rinaldi Munir, M.T. sebagai dosen pengampu mata kuliah matematika diskrit (IF2120) untuk kelas K3 yang telah menjadi pembimbing serta menurunkan segala ilmu dan pengetahuannya akan materi terkait hingga memberikan kesempatan kepada penulis untuk membuat makalah ini dan melakukan eksplorasi. Penulis juga ingin memberikan terima kasih juga kepada seluruh kolega dan pihak lain yang tidak dapat disebut satu-persatu yang telah mendukung penulis baik dari segi moral maupun material.

Referensi

[1] L. Null and J. Lobur, *Essentials of Computer Organization and Architecture*, Sudbury, MA: Jones & Bartlett Learning, 2012, pp.1173-1181.

[2] J. E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languagees, and Computation,* 3rd ed., Delhi: Pearson Education, 2022, pp. 171-275.

[3] J. Wengrow and B. MacDonald, "O Yes! Big O Notation," in *A Common-Sense Guide to Data Structures and Algorithms: Level Up Your Core Programming Skills,* 2nd ed., Raleigh, NC: The Pragmatic Bookshelf, 2020, pp. 35-76.

[4] Laaksonen, Antti, "Time Complexity" in *Competitive Programmer's Handbook,* https://cses.fi/book/book.pdf, 2018, diakses: 9-12-2022.

[5] Munir, Rinaldi, (2022), *Bahan Kuliah IF2120 Matematika Disktit: Kompleksitas Algoritma (Bagian 1),* https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian1.pdf, diakses: 3-12-2022

[6] Munir, Rinaldi, (2022), *Bahan Kuliah IF2120 Matematika Disktit: Kompleksitas Algoritma (Bagian 2),* https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Kompleksitas-Algoritma-2020-Bagian2.pdf, diakses: 3-12-2022

Pernyataan

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 10 Desember 2022

Krnny Benaya Nathan
NIM: 13521023