Penyelesaian *Superpermutation* dengan Metode Backtracking

Muhammad Khairul Makirin
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
khairulmakirin@live.com

***Abstract*—Persoalan superpermutasi merupakan persoalan yang saat ini sedang ramai dibicarakan, mulai dari *brute force* sampai teorema graf sudah dipakai sebagai metode pencarian superpermutasi terpendek, tetapi masih banyak hasil superpermutasi yang masih belum tervalidasi dan tidak diketahui. Dalam makalah ini penulis akan membahas bagaimana persoalan superpermutasi dapat diselesaikan dengan algoritma *backtraking* agar mendapat superpermutasi terpendek.**

Keywords—superpermutasi, backtracking, strategi algoritmik

I. Pendahuluan

Dalam bidang matematika ada permasalahan yang sedang populer yang menarik perhatian banyak, yaitu Superpermutasi. Superpermutasi merupakan masalah mencari *string* atau rangkaian karakter yang mengandung seluruh permutasi dari n buah objek sebagai sub-stringnya, tetapi yang paling menarik adalah mencari string terpendek yang mengandung seluruh permutasi dari n objek, masalah ini pertama kali diajukan oleh Ashlock dan Tillotson pada tahun 1993 [1], secara general masalah ini tergolong NP-Hard [2] dan mencari metode untuk menyelesaikan permasalahan ini masih menjadi penelitian yang aktif dalam bidang *computer science* dan matematika, dengan hasil terbaru yaitu pada tanggal 27 Februari 2019 telah ditemukan superpermutasi untuk n=7 dengan panjang 5906.

II. Landasan Teori

A. Algoritma *Backtracking*

Algoritma backtracking adalah sebuah metode pemecahan masalah yang mangkus, terstruktur, dan sistematis, algoritma ini sama seperti *exhaustive search*, hanya saja pada backtracking hanya pilihan yang mengarah ke solusi yang akan dipilih dan diekplorasi selanjutnya, pilihan yang tidak mengrah ke solusi tidak akan dipertimbangkan lagi dan akan dipangkas (*pruning*), hal ini lah yang membuat algoritma backtracking mangkus dan lebih baik dari *exhaustive search* karena dapat mengeliminasi banyak status hanya dengan pengecekan satu kali. Algoritma ini pertama kali diperkenalkan oleh D. H Lehmer pada tahun 1950, secara umum metode runut-balik mempunyai properti sebagai berikut [3] :

1. Solusi Persoalan

Solusi dinyatakan sebagai vektor dengan *n-tuple,*

$$X = (x_1, x_2, \ldots, x_n),\ x_i \in S_i$$

mungkin saja $S_1 = S_2 = \ldots = S_n$, sebagai contoh dalam permasalahan integer knapsack problem, solusi dapat digambarkan sebagai vektor yang terdiri atas 0 dan 1, $S_i = \{0, 1\}$

2. Fungsi Pembangkit

Fungsi pembangkit akan membangkitkan nilai x_k yang merupakan komponen dari vektor solusi,

3. Fungsi Pembatas

Fungsi pembatas atau fungsi kriteria dinyatakan sebagai

$$B(x_i, x_2, \ldots, x_k),$$

fungsi ini akan digunakan untuk menentukan apakah solusi saat ini $(x_i, x_2, \ldots, x_n)$ mengarah ke solusi, jika ya, maka pembangkitan nilai untuk x_{k+1} akan dilanjutkan, tetapi jika tidak maka $(x_1, x_2, \ldots, x_k)$ akan dibuang dan tidak akan pernah dipertimbangkan lagi dalam pencarian solusi. Fungsi pembatas tidak selalu dinyatakan dalam bentuk matematis, ia dapat dinyatakan sebagai predikat yang bernilai *true* atau *false*, atau dalam bentuk lain yang ekivalen.

B. Batas Bawah dari Superpermutasi

Pada tahun 2011 seorang *anonymous poster* di forum web 4chan.net telah membuktikan bahwa batas bawah (panjang minimal) dari suatu permutasi dari *n* objek adalah [4]

$$n! + (n-1)! + (n-2)! + n - 3,$$

karena pembuktian di atas sudah di luar cakupan dari makalah ini, penulis tidak akan menuliskannya pada makalah ini.

III. Analisis Masalah

Rekor untuk superpermutasi terpendek untuk tiap *n* objek sampai n = 7 adalah sebagai berikut [5]:

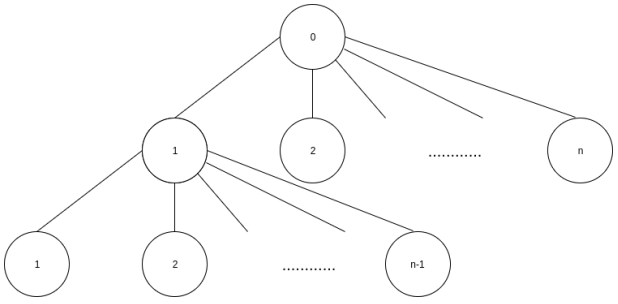
1. n = 1, dengan panjang = 1
2. n = 2, dengan panjang = 3
3. n = 3, dengan panjang = 9
4. n = 4, dengan panjang =33
5. n = 5, dengan panjang = 153
6. n = 6, dengan panjang = 873
7. n = 7, dengan panjang = 5906

Dapat dilihat untuk *n* yang semakin besar, panjang dari superpermutasi terpendeknya dapat meningkat secara eksponensial. Banyaknya pohon status yang akan diperiksa jika menggunakan teknik *brute force* untuk *n* objek dapat dicari dengan persamaan rekursif ini,

$$\sum_{i=0}^{n} S(i),$$

dengan *S(i)* adalah,

$$S(i) = \begin{cases} n, & i=0 \\ S(i-1)\ x\ (n-1), & i>0 \end{cases}$$



Gambar 1. contoh pohon pencarian memakai brute force

Sumber : Koleksi Pribadi Penulis

Untuk mendapatkan superpermutasi terpendek dari *n* objek akan diperlukan suatu fungsi untuk mendapatkan panjang subset terbesar antara prefix/suffix hasil solusi saat ini dengan prefix/suffix hasil permutasi/kandidat bagian solusi lainnya, hal ini dikarenakan semakin banyak *overlapping*/irisan antara hasil permutasi satu dengan yang lainnya akan menghasilkan string superpermutasi yang lebih pendek, selain itu untuk memvalidasi hasil yang telah didapatkan akan diperlukan fungsi seperti *string matching* untuk mencari seluruh permutasi dari *n* objek pada string yang telah dihasilkan.

IV. Penerapan Algoritma Backtracking

Parameter yang akan digunakan untuk backtracking adalah sebagai berikut,

1. Solusi Persoalan merupakan suatu vektor dengan *k-tuple*

$$X = (x_1, x_2, \ldots, x_k),$$

dari himpunan $S = \{1, \ldots, n\}$ yang menyatakan string dari solusi sekarang

2. Fungsi Pembangkit adalah aksi pengambilan salah satu hasil permutasi dari *n* buah objek

3. Fungsi Pembatas akan mengecek apakah panjang string setelah diambil suatu hasil pembatas merupakan string terpendek yang telah ditemukan

Implementasi lengkap dengan Bahasa Python 3 terdapat pada lampiran makalah ini.

Pada implementasi tersebut penulis menggunakan beberapa fungsi yaitu :

1. findLargestSubset

   Fungsi ini akan mengembalikan panjang subset terbesar dari irisan antara suffix atau prefix string solusi yang didapatkan sekarang dan string candidata selanjutnya yang merupakan hasil permutasi dari *n* objek

2. generateNewElement

   Fungsi ini akan mengembalikan seluruh elemen baru yang akan dimasukan ke stack berdasarkan simpul sekarang dan simpul-simpul anaknya

3. validateSolution

   validateSolution akan mengembalikan *true* atau *false* berdasarkan apakah semua anggota hasil permutasi *n* objek terdapat dalam sebuah *string.*

Proses kerja dari algoritma tersebut menggunakan DFS (*Depth First Search*) sebagai penelusuran pohon status, sehingga akan dipakai istilah yang biasa digunakan pada teori graf untuk menjelaskan algoritma tersebut, secara singkat algoritma tersebut dapat dituliskan sebagai berikut :

1. Enumerasi semua permutasi dari *n* objek
2. Bangkitkan salah satu hasil permutasi dengan fungsi pembangkit(x_k)
3. Hitung subset terbesar dari solusi sekrang dengan x_k
4. Gabungkan *string* solusi dengan x_k dengan banyaknya *overlapping* karakter adalah panjang subset terbesar dari solusi dan x_k
5. Jika panjang hasil langkah empat merupakan hasil terpendek sejauh ini, ekspan lagi simpul-simpul lainnya yang melalui simpul tadi, dengan kata lain, masukkan "simpul-simpul" anak ke dalam *stack*, dengan begini simpul-simpul yang sudah menghasilkan solusi yang lebih panjang dari rekor sekrang tidak akan di-ekspan
6. Jika simpul sekarang tidak mempunyai simpul anak, maka solusi yang didapatkan dari simpul paling awal ke simpul sekarang merupakan suatu superpermutasi, jika superpermutasi tersebut merupakan superpermutasi terpendek sejauh ini, *update* solusi global dan fungsi pembatas dengan panjang yang baru
7. Ulangi dari langkah dua sampai stack kosong.

V. Kesimpulan

Kesimpulan yang didapatkan adalah metode *backtracking* dapat digunakan sebagai metode alternatif untuk mencari superpermutasi terpendek dan tentunya lebih mangkus dan efisien daripada *exhaustive search*, adapun perbandingan dengan metode lain belum diperiksa dalam makalah ini.

Acknowledgment

Puji syukur penulis panjatkan kehadirat tuhan YME, atas karunia dan nikmat-Nya penulis dapat menyelesaikan makalah ini. Penulis juga ingin menyampai terima kasih yang sebesar-besarnya kepada orang tua yang telah mendukung penulis, dan tidak lupa kepada teman-teman serta dosen mata kuliah IF2211 Strategi Algoritma yang tanpa bantuan serta dukungannya tidak akan terwujud makalah ini.

References

[1] Houston, R. (22 Agustus 2014). *Tackling the Minimal Superpermutation Problem*

[2] Johnston, N. (18 Maret 2013). *Non-Uniqueness of Minimal Superpermutations*

[3] Munir, Rinaldi. (2008). *Diktat Kuliah IF2211: Strategi Algoritmik*

[4] Anonymous 4chan poster, Houston R., Pantone J.,Vatter V., (25 October 2018). *A Lower Bound on the Length of the Shortest Superpattern*

[5] http://www.njohnston.ca/2013/04/the-minimal-superpermutation-problem/ diakses pada 25 April 16.30 WIB

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, 26 April 2019

Muhammad Khairul Makirin
13517088

Lampiran 1.

```
""""
M. Khairul Makirin/13517088
Program Studi Teknik Informatika Institut Teknologi Bandung
""""

import itertools
import sys

def findLargestSubset(solution, candidate, fromPrefix = True):
   m = len(solution)
   n = len(candidate)
```

```
    maxSubset = 0

    if m > 0:
        if fromPrefix:
            i = 1
            j = n-1

            #Find the biggest subset from the prefix of solution and suffix of candidate
            while (j > 0):
                suffix = candidate[j:n]
                prefix = solution[0:i]

                if suffix == prefix:
                    maxSubset = i

                j -= 1
                i += 1

        else:
            i = m-1
            j = 1

            #Find the biggest subset from the prefix of candidate and suffix of solution
            while (j < n):
                suffix = solution[i:m]
                prefix = candidate[0:j]

                if suffix == prefix:
                    maxSubset = j

                j += 1
                i -= 1

        return maxSubset
    else:
        return 0

def generateNewElements(top, newSolution):
    #Initializing variables
    print(top)
    newEl = []
    lenSolution = len(newSolution)

    #Firstly delete x in top[1] and top[3]
    if top[3] != []:
```

```
        try:
            top[3].remove(top[1])
        except:
            print(top[3], top[1])
            exit()

    #If top[3] becomes empty
    if top[3] == []:
        newEl.append([lenSolution, None, newSolution, []])
    #If top[3] is not yet empty
    else:
        for element in top[3]:
            newEl.append([lenSolution, element, newSolution, top[3]])

    return newEl


def validateSolution(solution, n):
    permutations = tuple(["".join(i) for i in list(itertools.permutations(str(i) for i in
range(1, int(n)+1)))])

    for permute in permutations:
        if permute not in solution:
            return False, permute

    return True, n

def main():
    #initialize the variables
    solution = ''
    n = sys.argv[1]
    stack = []

    #Add every permutation of N objects
    permutations = tuple(["".join(i) for i in list(itertools.permutations(str(i) for i in
range(1, int(n)+1)))])
    print(permutations)
    #Update minLengthSolution
    minLengthSolution = len(permutations)*int(n)

    #Create the stack for DFS, each element represents the length of the current
solution, the next candidate, the current solution,
    #and the remaining permutations that have not been visited yet
    stack += [[len(solution), i, solution, list(permutations)] for i in permutations]
```

```
    #Start DFS, while the stack is not empty
    while stack != []:
        newEls = []
        for element in stack:
            print(element)
        print("\n")
        #Pop the top of the stack
        top = stack.pop(-1)

        #If there's no candidate left to check
        if top[1] == None:
            if top[0] <= minLengthSolution:
                solution = top[2]
                minLengthSolution = top[0]
        else:
            if top[1] not in top[2]:
                #Get the lenght of current candidate
                lenCandidate = len(top[1])

                #Finding the biggest subset from candidate and the current solution
                nbSubsetPrefix = findLargestSubset(top[2], top[1], True)
                nbSubsetSuffix = findLargestSubset(top[2], top[1], False)

                #Create new solution
                joinedPrefix = top[1][0:lenCandidate-nbSubsetPrefix] + top[2]
                joinedSuffix = top[2] + top[1][nbSubsetSuffix:lenCandidate]

                #Push newSolution to the stack
                if len(joinedPrefix) <= minLengthSolution:
                    newEls += generateNewElements(top, joinedPrefix)

                    for element in newEls:
                        inserted = False

                        for i in range(len(stack)):
                            if stack[i][0] < element[0]:
                                stack = stack[0:i] + [element] + stack[i:len(stack)]
                                inserted = True
                                break

                        if not inserted:
                            stack.append(element)
                if joinedPrefix != joinedSuffix and len(joinedSuffix) <=
minLengthSolution:
                    top[3].append(top[1])
```

```
                    newEls += generateNewElements(top, joinedSuffix)

                    for element in newEls:
                        inserted = False

                        for i in range(len(stack)):
                            if stack[i][0] < element[0]:
                                stack = stack[0:i] + [element] + stack[i:len(stack)]
                                inserted = True
                                break

                        if not inserted:
                            stack.append(element)
            else:
                newEls += generateNewElements(top, top[2])

                for element in newEls:
                    inserted = False

                    for i in range(len(stack)):
                        if stack[i][0] < element[0]:
                            stack = stack[0:i] + [element] + stack[i:len(stack)]
                            inserted = True
                            break

                    if not inserted:
                        stack.append(element)

    print(solution, minLengthSolution)

main()
```