Analisis Performa Chatbot Offline pada Platform Mobile Berbasis Android dengan Algoritma Knuth-Morris-Pratt dan Boyer-Moore

Reyhan Naufal Hakim/13517029
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
reyhan_kim@students.itb.ac.id

***Abstrak*—Penggunaan *chatbot* telah mencapai berbagai segmen industri, seperti industri telekomunikasi untuk *automated service*, industri kesehatan untuk konsultasi daring seputar penyakit umum, dan bahkan industri UMKM untuk pertanyaan seputar produk dan pemesanan barang. Akan tetapi, mayoritas dari *chatbot* yang beredar masih sangat bergantung dengan *server* karena performanya yang cepat dan *reliable*. Jarang ada aplikasi *chatbot* yang memanfaatkan performa perangkat *mobile* untuk dapat memproses pembicaraan pengguna secara *offline*. Ini dikarenakan *resource* perangkat *mobile* sangat terbatas sehingga menimbulkan kesan *chatbot* tidak responsif. Untuk itu, penulis meneliti tentang bagaimana performa *chatbot offline* pada platform *mobile* berbasis Android, khususnya dengan algoritma Knuth-Morris-Pratt dan Boyer-Moore untuk mengetahui seberapa baik performa pemrosesan *chatbot* di *platform* Android secara *native*, dari segi waktu maupun ruang.**

Keywords—chatbot; offline; performa; mobile

I. PENDAHULUAN

Chatbot adalah program komputer atau mungkin adalah kecerdasan buatan yang berkemampuan untuk berbincang via media teks ataupun suara. *Chatbot* biasanya didesain untuk mensimulasikan bagaimana manusia saling berinteraksi melalui percakapan sehingga chatbot disebut lolos *Turing Test.* [1] *Chatbot* biasa digunakan untuk sistem yang membutuhkan percakapan seperti implementasi *customer service* atau FAQ (*Frequently Asked Questions*). Beberapa chatbot memiliki kemampuan *Natural Language Processing* (NLP) untuk memproses masukan dari pengguna dan beberapa chatbot yang lebih simpel hanya mencocokkan *string* masukan dengan basis data yang ada di sistem *chatbot*.

Tipe *chatbot* yang penulis hendak teliti adalah *chatbot* dengan pemroses *string matching* dan bukan berbasis *Natural Language Processing* (NLP). Untuk *chatbot* ini, algoritma *string matching* yang digunakan adalah algoritma Knuth-Morris-Pratt (KMP) dan Boyer-Moore (BM). Algoritma Knuth-Morris-Pratt adalah algoritma *string matching* yang mencari kejadian sebuah "kata" (misal, W) pada *string* (misal, S) dengan memajukan karakter awal W untuk iterasi luar dan mencocokkan tiap karakter W dengan karakter dari *string* S untuk iterasi dalam. Hanya saja, berbeda dengan algoritma *naive string matching*, iterasi dapat dilangkahi beberapa kali dengan memanfaatkan informasi yang didapatkan dari *string matching* sebelumnya.[2] Pada algoritma Boyer-Moore, *string matching* dilakukan dari kanan ke kiri dan ketika ada ketidakcocokan (atau kecocokan penuh dari string *W*), algoritma ini akan menggunakan dua fungsi pergeseran yang disebut sebagai *good-suffix shift (mathing shift)* dan *bad-character shift* (*occurence shift*).[3]

Ide penelitian ini bermula dari permasalahan penulis dalam mengimplementasikan *chatbot* dengan *string matching* secara *offline* pada *platform* Android, di mana penulis menilai pemrosesannya sangat lambat dan tidak responsif. Karena itu, penulis berpikir perlu untuk melakukan pengujian terhadap performa aplikasi *chatbot offline* di Android. Untuk melakukan pengujian performa ruang dan waktu secara *realtime*, penulis memanfaatkan *Android Profiler* untuk menganalisis *CPU load*, penggunaan memori, dan waktu eksekusi program.

II. LANDASAN TEORI

A. Algoritma *Naive String Matching* (Baseline)

Misalkan terdapat sebuah *pattern* "W" dan sebuah teks *string* panjang "S", dan kita diminta menentukan apakah W ada pada S. Pada *Naive String Matching*, jika kita memiliki teks char T[n] dan *pattern* char P[m], kita dapat menentukan solusi dari permasalahan di atas dengan algoritma *C code* berikut:

```
for (i=0; T[i] != '\0'; i++) {
   for (j=0; T[i+j] != '\0' && P[j] != '\0' &&
   T[i+j]==P[j]; j++) ;
   if (P[j] == '\0') //found a match
}
```

Pada algoritma di atas, terdapat dua buah iterasi: iterasi dalam yang berkompleksitas O(m) dan iterasi luar yang berkompleksitas O(n). Dengan begitu, algoritma di atas secara keseluruhan berkompleksitas O(mn).

Algoritma ini tidak sepenuhnya buruk. Pada kasus terbaik dan rata-rata, biasanya iterasi dalam dapat menemukan ketidakcocokan *string* dengan cepat dan langsung berlanjut ke

iterasi luar selanjutnya. Akan tetapi, pada kasus terburuk, iterasi dalam dapat melanjutkan iterasi ke karakter terakhir dari *pattern*, di mana karakter terakhir tersebut tidak cocok dengan *string* teks, dan informasi yang didapatkan selama proses iterasi ini diabaikan begitu saja pada *naive string matching*. Padahal, jika kita teliti kembali, informasi pola karakter *string* teks yang didapatkan dari suatu iterasi dapat dimanfaatkan untuk mengoptimalkan iterasi selanjutnya.

```
      0  1  2  3  4  5  6  7  8  9 10 11
   T: w  a  w  a  w  a  w  o  m  a  w  o

i=0: X
i=1:    X
i=2:       w  a  w  X
i=3:          X
i=4:             w  a  w  o
i=5:                X
i=6:                   w  X
i=7:                         X
i=8:                            X
i=9:                               w  X
i=10:                              X
```

Misal, pada kasus di atas, kita ingin mencari *pattern* “wawo” pada *string* teks “wawawawomawo”. Beberapa perbandingan dari kasus di atas dinilai tidak efisien. Misalkan saja, setelah iterasi i = 2, kita mengetahui bahwa T[3] = “a” sehingga tidak perlu membandingkan “w” di iterasi i = 3. Dan juga, kita mengetahui bahwa T[4] = “w” sehingga kita tidak perlu mengulang iterasi yang sama pada i = 4.

B. Algoritma *Knuth-Morris-Pratt* (KMP)

Ide dibalik algoritma KMP adalah memanfaatkan informasi yang didapatkan dari *partial match* sepanjang j karakter saat pencocokan pattern dimulai dari posisi i sehingga kita mengetahui karakter apa saja yang ada di T[i]..T[i + j - 1]. Dari informasi ini, terdapat dua kemungkinan untuk menghemat jumlah iterasi yang dilakukan.

Pertama, kita dapat melangkahi beberapa iterasi yang tidak memiliki solusi yang mungkin dengan cara mencoba mencocokkan *partial match* yang sebelumnya telah ditemukan pada iterasi selanjutnya.

```
i=2: w  a  w
i=3:    w  a  w  o
```

Pada kasus di atas, dua buah *pattern* saling berlawanan. Kita mengetahui bahwa pada i = 2, T[3] dan T[4] adalah “a” dan “w”, sehingga tidak mungkin “w” dan “a” pada i = 3 adalah *pattern* yang kita cari. Kita dapat melangkahi satu atau lebih iterasi sampai kita menemukan sepasang *pattern* yang tidak saling berlawanan.

```
i=2: w  a  w
i=3:       w  a  w  o
```

Pada kasus di atas, terdapat sepasang “w” yang beririsan. Kita definisikan bahwa irisan dari dua buah *string* x dan y adalah urutan karakter terpanjang yang merupakan akhiran dari x dan awalan dari y. Pada kasus ini, irisan dari “waw” dan “wawo” adalah “w”. Secara umum, nilai i yang ingin kita langkahi adalah nilai i yang berkorespondensi terhadap irisan terpanjang dari *partial match* saat ini. Untuk merepresentasikan ide ini dalam bentuk *source code*, digunakan *C code* berikut:

```
int i=0;
while (i<n) {
    for (j=0; T[i+j] != '\0' && P[j] != '\0' &&
    T[i+j]==P[j]; j++);
    if (P[j] == '\0') //found a match;
    i = i + max(1, j-overlap(P[0..j-1],P[0..m]));
}
```

Selain optimasi di atas (yang merupakan optimasi untuk melangkahi iterasi luar yang tidak perlu), kita juga dapat melakukan optimasi untuk melangkahi iterasi dalam yang tidak perlu.

```
    i=2: w  a  w
    i=4:       w  a  w  o
```

Misal, pada kasus di atas, terdapat pasangan karakter “w” yang beririsan dan diuji oleh iterasi i = 2. Seharusnya, kita tidak perlu lagi melakukan pengujian pada i = 4. Jika kita sudah memiliki *pattern* irisan dengan *partial match* sebelumnya dalam suatu iterasi, kita dapat mengabaikan pengujian sepanjang karakter *pattern* yang beririsan sebelumnya. Dengan mengimplementasikan ide ini, kode di atas dapat kita ubah menjadi:

```
i=0;
o=0;
while (i<n) {
    for (j=o; T[i+j] != '\0' && P[j] != '\0' &&
    T[i+j]==P[j]; j++);
    if (P[j] == '\0') //found a match;
    o = overlap(P[0..j-1],P[0..m]);
    i = i + max(1, j-o);
}
```

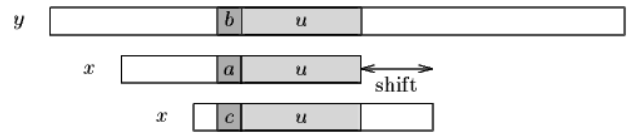
Algoritma KMP ini melakukan paling banyak 2n perbandingan, dengan kompleksitas waktu O(n).

C. Algoritma *Boyer-Moore* (BM)

Algoritma Boyer-Moore dinilai sebagai algoritma yang paling efisien untuk pencocokan *string*. Algoritma Boyer-Moore biasa digunakan pada *text editor*, khususnya untuk menjalankan operasi *search* dan *replace*. Algoritma ini bekerja dengan memindai karakter-karakter dari *pattern* yang dimiliki dari ujung paling kanan teks hingga ujung paling kiri, dimulai dari potongan *string* yang paling kanan dari teks tersebut. Jika *string* tidak cocok atau *string* cocok seluruhnya (bukan *partial match*), algoritma ini akan menggunakan dua buah fungsi untuk menggeser *pattern* pengujian *string matching*: *good-suffix shift* (*matching shift*) dan *bad-character shift* (*occurence shift*).

Misal, kita memiliki *string* x yang merupakan *pattern* yang ingin kita cari dan *string* y yang merupakan *string* teks. Kita asumsikan terjadi ketidakcocokan *string* pada karakter x[i] = a

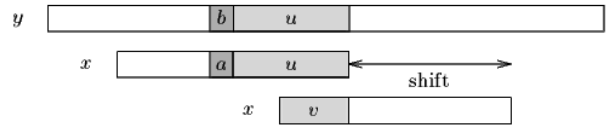
pada *pattern* dan karakter y[i+j] = b pada *string* teks pada sebuah pengujian di posisi j.



Gambar 1. *good-suffix shift*, *pattern* u yang beririsan diawali oleh karakter c, di mana c berbeda dengan a dan c didapatkan di iterasi yang lebih awal.

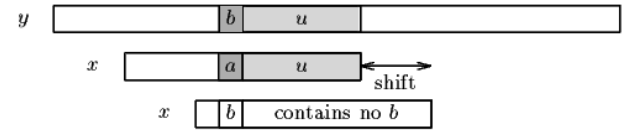
Lalu, x[i + 1 ... m - 1] = y[i + j + 1 ... j + m - 1] = u dan x[i] ≠ y[i + j]. *Good-suffix shift* menyejajarkan y[i + j + 1 ... j + m - 1] = x [i + 1 ... m - 1] dengan *partial match* pada x yang didahului oleh karakter yang berbeda dari x[i] (lihat gambar di atas).

Jika *partial match* seperti contoh di atas tidak ada, pergeseran yang terjadi adalah pergeseran *pattern* agar urutan karakter akhir v terpanjang dari y[i + j + 1 ... j + m - 1] sejajar dengan awalan x yang cocok (lihat gambar 2).



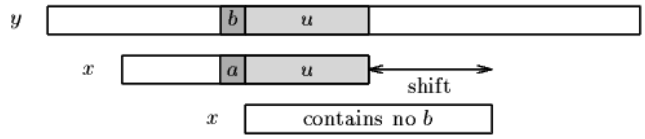
Gambar 2. *good-suffix shift*, hanya akhiran dari u yang beririsan dengan x.

Bad-character shift terdiri dari proses menyejajarkan *string* teks y[i + j] dengan urutan karakter paling kanan yang cocok dari x[0 ... m-2], seperti yang dapat dilihat di gambar di bawah ini:



Gambar 3. *the bad-character shift*, a ada di x.

Jika y[i + j] tidak ada pada *pattern* x, tidak ada urutan karakter pada x yang ada di y dan dapat memenuhi y[i + j], dan ujung kiri dari posisi pengujian disejajarkan dengan y[i + j + 1].



Gambar 4. *the bad-character shift*, b tidak ada pada x.

Melalui ide di atas, kita dapat membentuk *pseudocode* daru algoritma Boyer-Moore:

```
Input: string x (pattern) and string y (text)
Output: index of the first substring of pattern
i <-- m – 1
j <-- m – 1
repeat
       if x[j] = y[i] then
              if j = 0 then
                     return i {match found}
              else {check the next char}
                     i <-- i – 1
                     j <-- j – 1
       else {y[j] <> x[i] move the pattern}
       i <-- i + m – j – 1
       i <-- i + max(j – last(y[i]), match(j))
       j <-- m – 1
until i > n – 1
return “no match”
```

Algoritma BM ini melakukan paling banyak 3n perbandingan, dengan kompleksitas waktu O(n/m).

D. Android Native Application Development

Android adalah *platform mobile* yang paling banyak digunakan di dunia saat ini. Pada Desember 2018, menurut majalah CeoWorld, Android memegang *market share* sebesar 75,16% untuk sistem operasi *mobile* di dunia. Android dikembangkan oleh Google sebagai sistem operasi yang *Unix-like*, berjalan di atas kernel Linux yang telah dimodifikasi. Android ditulis dengan bahasa Java (untuk UI), C (kernel dan OS), C++, Kotlin, dan bahasa lainnya. Untuk pengembangan *Native Android Application*, dibutuhkan bahasa Java atau Kotlin karena *virtual machine*/*runtime* yang ada pada Android (Dalvik dan ART) hanya dapat mengenali *bytecode* dari dua bahasa itu secara native.

Karena perangkat *mobile* memiliki *resource* yang sangat terbatas, terutama dari segi pemroses (CPU) dan memori, optimasi pada *software* level sangat penting dilakukan untuk tetap menjamin *responsiveness* dan *reliability* dari sebuah *Android App*.

III. Implementasi Aplikasi Mobile

A. Gambaran Implementasi

Implementasi aplikasi Android dilakukan dengan menggunakan Android Studio IDE dengan Android SDK untuk Android versi 6.0 hingga 9.0. Bahasa pemrograman yang digunakan adalah Java, dengan *layouting* memanfaatkan XML. Aplikasi yang dibuat pada dasarnya sudah pernah penulis buat dan publikasikan melalui *git repository* penulis.[4]

Secara umum, aplikasi akan menjalankan sebuah *MessageListActivity* yang merupakan layar utama dari aplikasi. *MessageListActivity* merupakan layar seperti *messaging app* pada umumnya. Hanya saja, pengguna berbicara langsung dengan *chatbot*.

Saat pengguna mengirimkan sebuah pesan, *string* pesan akan dikirimkan ke *constructor* Message untuk selanjutnya ‘dilempar’ menjadi pesan kepada pemroses *string matching*. Setelah mengubah string menjadi *lowercase*, menghilangkan

tanda baca, dan menghilangkan *stopwords*, pemroses akan mengirim *object* Message kepada *MessageListActivity* sebagai pesan respons. *Adapter* pada *MessageListActivity* kemudian akan memperbaharui view pada *RecyclerView* yang ada di *MessageListActivity* sehingga pesan respons dapat dilihat pengguna sebagai jawaban dari *chatbot*.

Hanya saja, jika kita tidak memproses pesan pengguna secara *multithread*, aplikasi tidak akan bisa merespons masukan apapun dan menimbulkan kesan tidak responsif. Karena itu, penulis mengimplementasikan *multithreading* pada pemroses pesan agar aplikasi tetap dapat menerima masukan walaupun pemroses *string matching* sedang bekerja.

B. Source Code Khusus String Matching

Berikut adalah potongan kode program yang dibutuhkan untuk menghilangkan tanda baca, mengubah kata masukan menjadi *lowercase*, dan menghilangkan *stopwords*.

```
public static String FormattingString(ArrayList<String>
stopWords, ArrayList<String> synonym, String kata){
        //Menghapus tanda ? pada format pencarian
        kata = kata.replace("\\W?\\?","");
        kata = kata.toLowerCase();

        String[] subkata = kata.split(" ");

        for(int i=0;i<stopWords.size();i++){
            if(cekRegex(kata, stopWords.get(i))){
                //Menghapus stopwords pada kata
                for(int j=0;j<subkata.length;j++){
                    if(subkata[j]!=stopWords.get(i)){
                        if(j==0){
                           kata = subkata[j] + " ";
                        }else if(j==subkata.length-1){
                           kata += subkata[j];
                        }else{
                           kata = kata + subkata[j] + " ";
                        }
                    }
                }
            }
        }

        //Menghapus synonym pada input pertanyaan dengan
pencarian kata synonym menggunakan regex
        for(int i=0;i<synonym.size();i++){
            String[] temp = synonym.get(i).split("\\=");
            if(cekRegex(kata, temp[0])){
                kata = kata.replaceAll(temp[0], temp[1]);
            }
        }

        return kata;
    }

public static boolean cekRegex(String kataInput, String
pattern){

        boolean cek;

        String regex = ".*" + pattern + ".*";

        cek = Pattern.matches(regex, kataInput);

        return cek;

    }
```

Untuk memproses data *string* menjadi respons *chatbot*, digunakan potongan kode berikut:

```
public static String StringMatching(String message){
        String question = message;
        ArrayList<String> dataPertanyaan =
getDataPertanyaan();
        ArrayList<String> dataSynonym = getDataSynonym();
        ArrayList<String> dataStopWords =
getDataStopWords();

        ArrayList<String> alternate = new
ArrayList<String>();

        question = FormattingString(dataStopWords,
dataSynonym, question);
        final float matchPercentage = (float) 0.9;
        float persentasekecocokan;
        String[] temp;
        boolean found = false;
        int i;
        int result;
        String out = null;

        //Pengecekan langsung satu String dengan algoritma
KMP
        i=0;
        while(i<dataPertanyaan.size() && !found){
            temp = dataPertanyaan.get(i).split("\\?\\W");
            result = KMP(question,
FormattingString(dataStopWords, dataSynonym, temp[0]));
            if(result==0){
                out = temp[1];
                found = true;
            }else{
                i++;
            }
        }

        if(!found){
            //Pengecekan per-substring dengan algoritma BM
            i=0;
            float pembilang;    //counter jumlah huruf
yang match
            float pembagi;      //counter jumlah huruf
dalam kalimat
            while(i<dataPertanyaan.size() && !found){
                String[] subkata = question.split(" ");

                temp =
dataPertanyaan.get(i).split("\\?\\W");
                pembilang=0;
                pembagi = temp[0].length()-
subkata.length+1;

                for(int j=0; j<subkata.length;j++){
                    result =
BM(FormattingString(dataStopWords, dataSynonym,
temp[0]),subkata[j]);

                    if(result!=-1){
                        pembilang += subkata[j].length();
                    }
                }

                persentasekecocokan = pembilang/pembagi;


if((persentasekecocokan)>=matchPercentage){
                    out = temp[1];
                    found = true;
                }else{
                    if((persentasekecocokan)>=0.5){
                        alternate.add(temp[0]);
                    }
                    i++;
```

```
                }
            }
        }

        if(!found){ //Pertanyaan tidak ditemukan baik
dicari dari kalimat maupun per-kata
            if(alternate.size()!=0){
                out = "Apakah yang Anda maksud : \n";

                for(int k=0;k<alternate.size();k++){
                    if(k==alternate.size()-1){
                        out = out + "- " +
alternate.get(k);
                    }else{
                        out = out + "- " +
alternate.get(k) + "\n";
                    }

                }
            }else{
                out = "I don't understand, senpai~ Please
try a different sentence. >w<";
            }
        }
        return out;
}
```

Berikut adalah algoritma BM yang digunakan:

```
public static int BM(String kataInput, String pattern){
        int last[] = buildLast(pattern);
        int n = kataInput.length();
        int m = pattern.length();
        int i = m-1;

        if(i>n-1){
            return -1;
        }else{
            int j = m-1;
            do{

if(pattern.charAt(j)==kataInput.charAt(i)){
                    if(j==0){
                        return 0;
                    }else{
                        i--;
                        j--;
                    }
                }else{
                    int lo = last[kataInput.charAt(i)];
                    i = i+m-Math.min(j, 1+lo);
                    j = m-1;
                }
            }while(i<=n-1);

            return -1;
        }
}
public static int[] buildLast(String pattern){

        int last[] = new int[128];

        for(int i=0;i<128;i++){
            last[i] = -1;
        }

        for(int i=0;i<pattern.length();i++){
            last[pattern.charAt(i)] = i;
        }

        return last;
}
```

Dan, berikut adalah algoritma KMP yang digunakan:

```
public static int KMP (String kataInput, String pattern){
        int n = kataInput.length();
        int m = pattern.length();

        int fail[] = computeFail(pattern);

        int i = 0;
        int j = 0;

        while(i<n){
            if(pattern.charAt(j) == kataInput.charAt(i)){
                if(j==m-1){
                    return 0;
                }
                i++;
                j++;
            }else if(j>0){
                j = fail[j-1];
            }else{
                i++;
            }
        }

        return -1;
}

public static int[] computeFail(String pattern){

        int fail[] = new int[pattern.length()];
        fail[0] = 0;

        int m = pattern.length();
        int i = 1;
        int j = 0;

        while(i<m){
            if(pattern.charAt(j)==pattern.charAt(i)){
                fail[i] = j+1;
                i++;
                j++;
            }else if(j>0){
                j = fail[j-1];
            }else{
                fail[i] = 0;
                i++;
            }
        }
        return fail;
}
```

Berikut adalah struktur Message.java:

```
public class Message {
        private String message;
        private User sender;
        private Date createdAt;

        public Message(String message, User sender) {
            setMessage(message);
            setSender(sender);
            setCreatedAt();
        }

        String getMessage() {
            return this.message;
        }

        User getSender() {
            return this.sender;
        }

        Date getCreatedAt() {
```

```
            return this.createdAt;
        }


        void setMessage(String message) {
            this.message = message;
        }

        void setSender(User sender) {
            this.sender = sender;
        }

        void setCreatedAt() {
            this.createdAt =
Calendar.getInstance().getTime();
        }
}
```

C. *Database*

Database yang penulis gunakan bersifat raw textfile (berformat .txt) dan tidak menggunakan NoSQL. Data yang disimpan dalam database tersebut adalah sebagai berikut

1. Data pertanyaan (datapertanyaan.txt), dengan 100 entri data dan berisi 5188 karakter.
2. Data *stopwords* (datastopwords.txt), dengan 758 entri data dan berisi 6442 karakter.
3. Data sinonim (datasynonym.txt), dengan 7 entri data dan berisi 110 karakter.

Seluruh isi database disimpan dalam sebuah ArrayList pada saat aplikasi dijalankan.

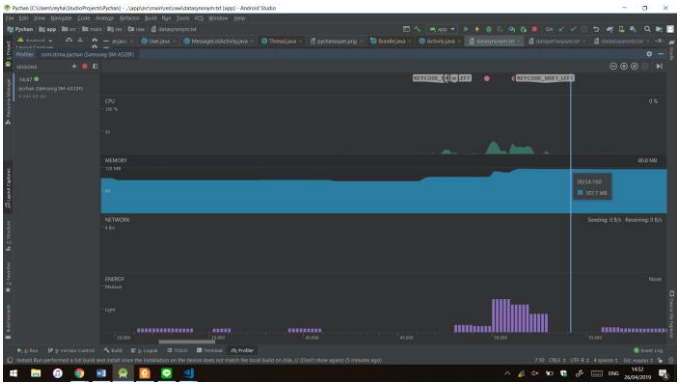
IV. Analisis Performa Algoritma

Analisis performa akan dibagi menjadi beberapa bagian: *idle*, respon 100% cocok, memunculkan pilihan respon, dan respon tidak cocok. Analisis dilakukan pada perangkat Samsung Galaxy A5 (2017) dengan spesifikasi perangkat keras sebagai berikut:

- Prosesor: Octa-core 1.9 GHz Cortex-A53
- Memori: 3 GB RAM
- Sistem Operasi: Android 8.0.1 Oreo

A. Idle

Saat idle, dengan menggunakan Android Profiler, CPU Load rata-rata berada di angka 0% dan penggunaan memori berada di angka 107.6MB hanya untuk menjalankan aplikasi chatbot. Dari pengamatan, tampak bahwa penggunaan CPU tinggi saat membuka aplikasi saja (terdapat spike) hingga penggunaan CPU mencapai 25%.

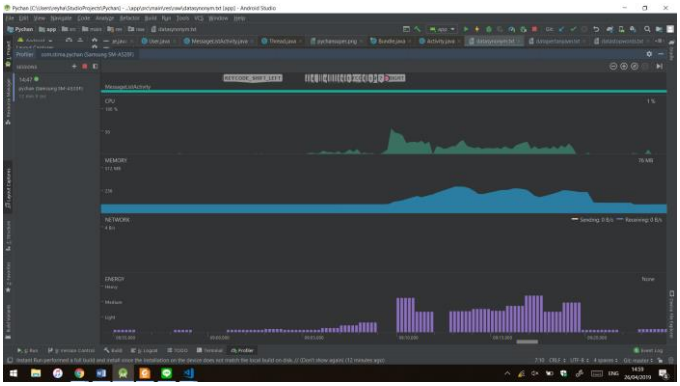


Gambar 5. Tampilan *Android Profiler* saat aplikasi idle.

B. Respon 100% Cocok

Saat menerima pesan yang responnya 100% cocok (artinya, pesan yang diterima akan *match* 100% dengan database), dengan menggunakan Android Profiler, CPU Load berada di rentang 12.6% hingga 53 % selama 10 detik dan penggunaan memori meningkat dari 107.6MB menjadi hingga 304MB (fluktuatif antara 304MB dan 187.3MB).

Dari pengamatan, tampak bahwa penggunaan CPU melonjak tinggi hingga ke angka 53% tepat setelah menerima masukan dari pengguna. Terlihat pula bahwa *garbage collector* dari Java bekerja selama program melakukan *string matching*.



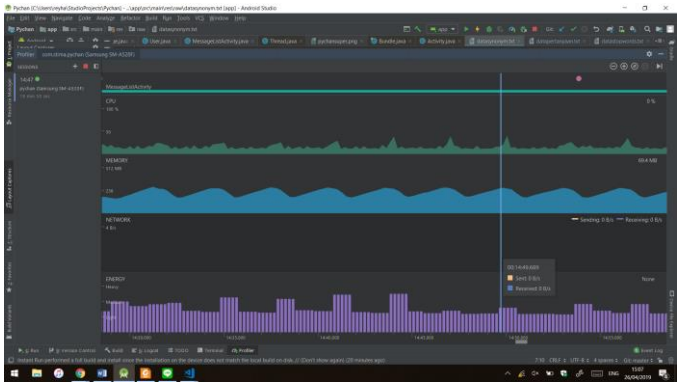
Gambar 6. Tampilan *Android Profiler* saat aplikasi menerima pesan yang ada di database.

C. Respon 50% sampai 90% Cocok dan Menampilkan Pilihan Respon

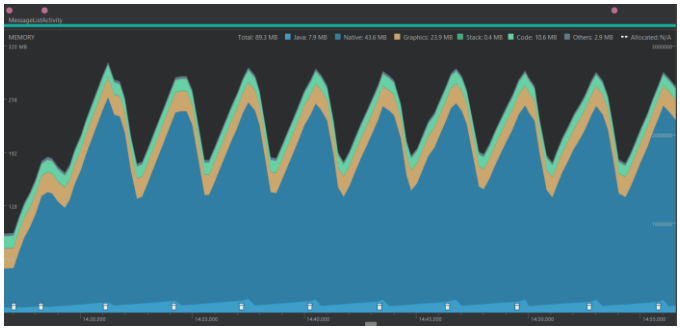
Saat menerima pesan yang responnya 50% sampai 90% cocok dan menampilkan pilihan respon, dengan menggunakan Android Profiler, CPU Load berada di rentang 13.1% hingga 45.6% selama 57 detik dan penggunaan memori meningkat dari 95.5MB menjadi hingga 284.4MB (fluktuatif antara 284.4MB dan 183.8MB).

Dari pengamatan, tampak bahwa *garbage collector* terus-menerus menghapus data penelurusan ArrayList yang dianggap tidak akan diakses lagi – padahal besar memori yang digunakan kemudian identik sehingga dapat disimpulkan bahwa runtime Android menghapus dan memuat data yang sama berulang-ulang. Terdapat potensi optimasi di sini, di mana seharusnya

garbage collector tidak menghapus data yang nantinya perlu dipakai kembali.



Gambar 7. Tampilan *Android Profiler* saat aplikasi menerima pesan yang kecocokannya antara 50% sampai 90% dan diminta menampilkan pilihan pertanyaan.

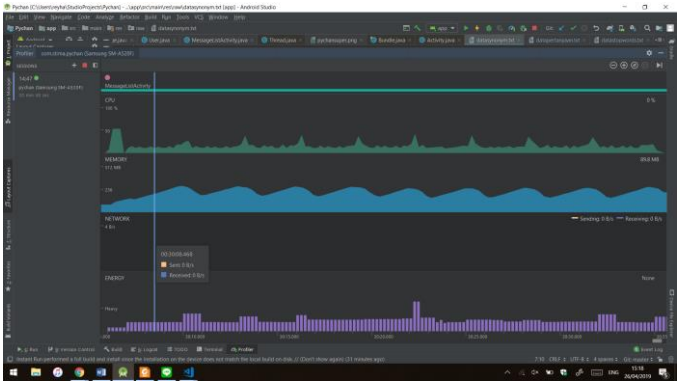


Gambar 8. Tampak bahwa *garbage collector* menghancurkan data dari memori (icon putih di bagian bawah – gambar *trashcan*) yang nantinya akan dialokasikan kembali. Proses ini memakan CPU time dan tentu mengurangi performa aplikasi secara kompleksitas waktu.

D. Respon Tidak Cocok

Saat menerima pesan yang kecocokannya di bawah 50%, dengan menggunakan Android Profiler, CPU Load berada di rentang 12% hingga 54% selama 31 detik dan penggunaan memori meningkat dari 90.6MB menjadi hingga 297.7MB (fluktuatif antara 297.7MB dan 196.4MB).

Dari pengamatan, tampak bahwa *garbage collector* terus-menerus menghapus data penelurusan ArrayList yang dianggap tidak akan diakses lagi – padahal besar memori yang digunakan kemudian identik sehingga dapat disimpulkan bahwa runtime Android menghapus dan memuat data yang sama berulang-ulang. Terdapat potensi optimasi di sini, di mana seharusnya *garbage collector* tidak menghapus data yang nantinya perlu dipakai kembali.



Gambar 9. Tampilan *Android Profiler* saat aplikasi menerima pesan yang kecocokannya <50%

V. KESIMPULAN

Melalui penelitian ini, dapat disimpulkan bahwa selain algoritma yang digunakan, sifat dari suatu *runtime* dapat mempengaruhi efisiensi dari suatu program. Dalam kasus ini, sifat agresif dari *garbage collector* pada *ART runtime* milik Android membatasi performa aplikasi *chatbot* yang diimplementasikan secara *offline*.

REFERENCES

[1] "What is a chatbot?". techtarget.com. Diakses pada 26 April 2019.

[2] Eppstein, *Knuth-Morris-Pratt String Matching*, Irvine, CA: University of California, 1996.

[3] C. Christian, L. Thierry, *Boyer-Moore Algorithm*. Paris, FR: Université Paris-Est Marne-la-Vallée, 1997.

[4] "pychan-android". https://github.com/hikarukei/pychan-android. Diakses pada 26 April 2019.

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

Reyhan Naufal Hakim
13517029