Identifikasi Plagiarisme Karya Musik dengan Penerapan Algoritma *String Matching* dan Levenshtein Distance

Sarah Azka Arief - 13520083
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: sarahazkarief@gmail.com

***Abstrak*—Dengan terbatasnya jumlah nada yang dapat digunakan untuk membuat suatu karya musik, plagiarisme telah menjadi suatu permasalahan yang lazim ditemukan pada industri musik. Identifikasi kehadiran plagiarisme pada suatu karya musik dapat dilakukan dengan menerapkan algoritma *string matching* untuk membandingkan kemiripan melodi antarlagu. Pada makalah ini, penentuan plagiarisme musik akan dibahas dengan memanfaatkan konsep *string matching* dengan menggunakan algoritma Knuth-Morris-Pratt dan Boyer-Moore.**

Keywords—Plagiarisme, Musik, String Matching, Levenshtein Distance

I. Pendahuluan

Sebagai bidang yang menghasilkan keuntungan dengan menghasilkan lagu dan komposisi musik, industri musik dapat diibaratkan sebagai pabrik pendorong penghasilan karya musik. Karena terbatasnya variasi nada yang dapat dihasilkan dari jumlah nada yang ada di dunia, tak jarang apabila beberapa karya musik yang dihasilkan di industri musik dapat terdengar mirip antara satu sama lain. Ketika suatu musik terdengar mirip dengan musik lainnya, terdapat kemungkinan bahwa salah satu dari kedua musik tersebut merupakan hasil dari plagiarisme yang dilakukan terhadap karya musik lainnya. Benar atau tidaknya dugaan plagiarisme tersebut dapat ditentukan dengan membandingkan elemen-elemen pembangun kedua musik tersebut.

Musik adalah seni aransemen bunyi dengan memodifikasi unsur seperti melodi, ritme, hingga warna nada. Musik sendiri dapat didefinisikan melalui beragam elemen seperti melodi, ritme, atau warna nada yang mendirikan musik tersebut. Corak dari suatu karya musik yang dapat membedakan suatu karya musik dengan yang lainnya juga dapat didefinisikan melalui elemen-elemen pembangun tersebut. Karena elemen-elemen tersebut dapat dijadikan indikator yang membedakan suatu karya musik dengan karya musik lainnya, pada makalah ini akan diambil salah satu elemen yang akan dijadikan tolak ukur plagiarisme antarkarya musik. Elemen yang akan dikulik dan dibahas pada makalah ini adalah elemen melodi dari suatu musik yang terdiri atas susunan nada.

II. Landasan Teori

A. Plagiarisme Musik

Kegiatan mengakui suatu karya musik yang adalah imitasi atau merupakan karya musik milik orang lain sebagai karya milik sendiri merupakan plagiarisme musik. Berdasarkan konteksnya, kasus plagiarisme musik dapat dibagi menjadi dua konteks yakni *sampling* dan *music idea*. Pada *music idea*, plagiarisme dilakukan terhadap melodi atau motif dari suatu karya musik. Sedangkan pada *sampling*, porsi dari suatu rekaman lagu digunakan sebagai bagian dari lagu lain. Demi mengantisipasi terjadinya tindakan plagiarisme, hak cipta serta kekayaan intelektual dari suatu karya musik telah diatur oleh hukum.

Sebuah karya kreatif tak perlu didaftarkan terlebih dahulu untuk mendapatkan perlindungan hukum karena hak kekayaan intelektual merupakan hak eksklusif yang dimiliki oleh pencipta. Oleh karena itu, karya apapun yang sudah dipublikasikan dan diketahui oleh khalayak umum sudah secara otomatis mendapat perlindungan hukum melalui hak kekayaan intelektual. Karya yang termasuk mencakup karya seni, sastra, dan ranah ilmu pengetahuan sehingga karya musik termasuk karya yang mendapat perlindungan hukum dengan adanya hak kekayaan intelektual.

Pada UU No. 28 Tahun 2014 yang mengatur hak cipta, tidak dijabarkan secara spesifik mengenai definisi plagiarisme itu sendiri. Namun, secara umum plagiarisme dalam undang-undang hak cipta dijabarkan dalam beberapa istilah seperti mengumumkan, memublikasikan, atau menjual hasil karya orang lain tanpa seizin pemilik karya tersebut. Apabila terbukti bahwa suatu karya merupakan hasil plagiarisme, secara hukum pencipta karya plagiat tersebut dapat dituntut karena tergolong telah melakukan tindakan pidana.

Menurut undang-undang hak cipta, apabila tidak ada pengakuan, musisi yang menuduh orang lain atas tuduhan pencurian karya harus membuktikan ‘akses’ dan ‘kemiripan’ lagu tersebut dengan lagu yang tersangka dijiplak. Dalam hal ini, ‘akses’ berarti orang yang diduga sebagai penjiplak terbukti

telah mendengar lagu yang dijiplak sementara ‘kesamaan’ adalah lagu yang diduga plagiat harus memiliki komponen musik yang unik dari lagu yang dijiplak. Walaupun begitu, sulit untuk mencapai definisi pasti dari aspek ‘kemiripan’ suatu lagu dengan yang lainnya.

Batasan suatu karya dapat disebut sekedar mengambil inspirasi dari karya lain atau menjiplak hasil karya orang lain relatif subjektif. Contoh, seorang musisi bisa saja mengatakan bahwa karyanya dijiplak oleh musisi lain, tetapi batasan sejauh mana sang penduga dapat menggugat pencipta yang terduga menjiplak tersebut tergolong abu-abu. Oleh karena itu, indikator plagiarisme dalam musik biasanya akan menitikberatkan elemen-elemen pembangun suatu musik seperti contoh melodi dari musik tersebut. Apabila ditemukan bahwa melodi antara dua lagu menyerupai satu sama lain, bukti tersebut dapat mendukung dugaan plagiarisme pada salah satu lagu tersebut.

B. String Matching

Permasalahan yang berusaha diselesaikan pada konsep pencocokkan *string* atau *string matching* adalah mencari kemunculan suatu pola berupa *string* dalam teks atau *string* lainnya. Jika diberikan teks T berupa *string* yang panjangnya n karakter dan pola P berupa *string* yang panjangnya m karakter, dapat dicari lokasi pertama di T yang bersesuaian dengan P dengan asumsi m < n.

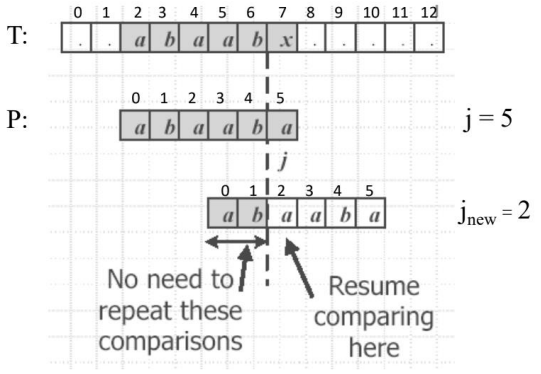
Pada *string matching*, terdapat istilah *prefix* dan juga *suffix*. Pada *string* S dengan ukuran m dengan S = x0 x1 … xm-1, maka *prefix* dari S merupakan *substring* S[0…k] dan *suffix* dari S merupakan *substring* S[k…m-1] dengan k merupakan indeks antara 0 hingga m-1. Oleh karena itu, jika terdapat *string* S berupa ‘stima’, kemungkinan *prefix* dari S adalah ‘s’, ‘st’, ‘sti’, ‘stim’, dan ‘stima’ sementara kemungkinan *suffix* dari S adalah ‘a’, ‘ma’, ‘ima’, ‘tima’, dan ‘stima’.

String matching sendiri dapat digunakan dalam pencarian baik seperti di dalam *editor text*, *web search engine*, hingga *bioinformatics*. Terdapat beberapa algoritma yang dapat digunakan untuk *string matching* seperti *brute force algorithm*, *rabin-karp algorithm*, dan *knuth-morris-pratt algorithm*. Pada makalah ini, dua algoritma yang akan digunakan untuk mengidentifikasi plagiarisme antarkarya musik adalah algoritma *knuth-morris-pratt* dan *boyer-moore*.

1) Algoritma Knuth-Morris-Pratt

Algoritma Knuth-Morris-Pratt (KMP) merupakan algoritma *string matching* yang mencari pola pada suatu teks dari kiri ke kanan. Pola dari algoritma KMP menyerupai algoritma *brute force* dengan perbedaan pada algoritma ini pemindahan *pattern* yang dilakukan relatif lebih baik apabila dibandingkan dengan algoritma *brute force*. Hal ini dikarenakan pergeseran yang dilakukan pada algoritma KMP relatif lebih heuristik dan cerdik ketimbang algoritma *brute force*.

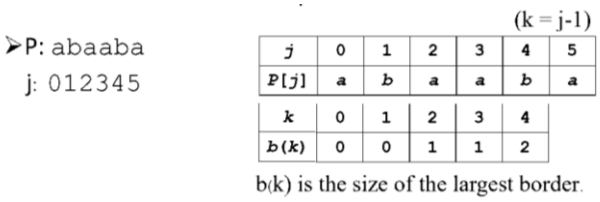
Pada algoritma Knuth-Morris-Pratt, urutan penyelesaian dimulai dengan mencari cara menghindari perbandingan yang sia-sia ketika sedang melakukan perpindahan dan perbandingan pola. Apabila pada persoalan dengan teks T serta pola P ditemukan perbedaan antara P[j] dan T[i], pergeseran palng efektif terhadap pola yang menghindari perbandingan tidak efektif adalah menggeser prefix terbesar dari P[0…j-1] yang merupakan *suffix* dari P[1…j-1].



Gambar 2.1 Contoh algoritma Knuth-Morris-Pratt

Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Seperti pada gambar 2.1, pada pola P berupa ‘abaab’ didapat *prefix* terbesar yang juga merupakan *suffix* adalah ‘ab’. Jumlah pergesaran akan sama dengan panjang pola dikurangi panjang *prefix* terbesar sehingga dengan panjang pola sebesar 5 dan panjang *prefix* terbesar berupa 2 didapat jumlah pergeseran sebesar 3. Pada KMP, terjadi *preprocessing* pada *pattern* untuk menemukan semua *prefix* terpanjang dalam pola untuk setiap posisi dengan menggunakan *border function* b(k). Apabila j adalah posisi terjadinya ketidasamaan dan k adalah pola sebelum ketidaksamaan (k = j – 1), b(k) adalah panjang *prefix* terpanjang dari P[0…k] yang merupakan *suffix* dari P[1…k]. Contohnya dapat dilihat sebagai berikut:



j	0	1	2	3	4	5
P[j]	a	b	a	a	b	a

k	0	1	2	3	4
b(k)	0	0	1	1	2

Gambar 2.2 Contoh Border Function

Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Pada gambar 2.2, b(4) bernilai 2 karena pada posisi itu *prefix* terpanjang yang juga merupakan *suffix* adalah pola ‘abaab’ yaitu ‘ab’ dengan panjang 2 dan seterusnya seperti demikian.

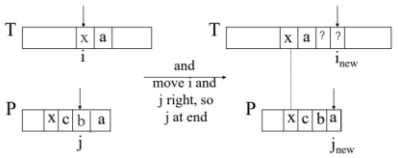
Secara umum, dapat disimpulkan bahwa KMP mirip dengan algoritma *brute force* dengan perbedaan adanya *preprocessing* dan teknik pergeseran yang membuat KMP lebih efisien dan lebih cepat. Kompleksitas waktu dari algoritma KMP adalah O(m+n). Kelebihan lain dari KMP adalah algoritma tidak memerlukan *backtracking* pada *text input* sehingga algoritma ini cocok untuk memproses *file* yang sangat besar. Adapula kekurangan dari algoritma

KMP adalah ketika ukuran dari alfabet bertambah karena kemungkinan terjadinya *error* semakin besar.

2) Algoritma Boyer-Moore

Boyer-Moore merupakan algoritma *string matching* yang proses pencariannya didasarkan pada dua teknik yakni *looking-glass technique* dan *character-jump* technique. Teknik *looking-glass* digunakan untuk mencari pola P pada teks T dengan bergerak mundur menelusuri P dimulai dari ujungnya. Untuk teknik *character-jump*, ambil contoh kasus dimana terjadi ketidaksamaan pada T[i] dngan T[i] adalah x dan P[j] adalah b. Terdapat 3 kasus yang mungkin untuk scenario tersebut yakni sebagai berikut:

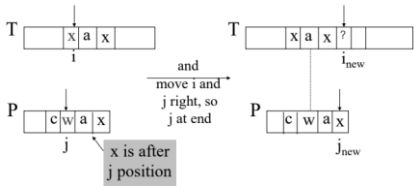
1. Kasus 1 adalah saat P memiliki huruf x di suatu tempat. Pada kasus ini, P akan dipindahkan ke kanan agar meluruskan kemunculan terakhir x pada P di T[i].



Gambar 2.3 Contoh kasus 1

Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

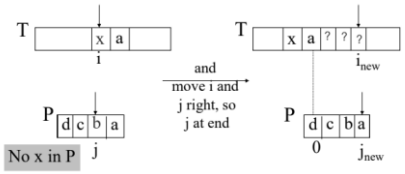
2. Kasus 2 adalah P memiliki huruf x, tetapi tidak bisa dipindahkan ke kanan (misal karena x berada pada posisi pattern sekarang). Pada kasus ini, P hanya digeser ke kanan sebesar 1.



Gambar 2.4 Contoh kasus 2

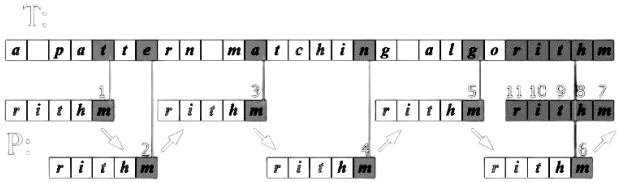
Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

3. Kasus 3 adalah saat kasus 1 dan 2 tidak bisa diaplikasikan sehingga P digser untuk meluruskan P[0] dengan T[i+1].



Gambar 2.5 Contoh kasus 3

Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf



Gambar 2.6 Contoh algoritma Boyer-Moore

Sumber:https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf

Pada algoritma Boyer-Moore, dilakukan *preprocess* untuk mengetahui kemunculan terakhir suatu alfabet pada pola P. Kemunculan ini dicari dengan suatu fungsi yang melakukan *preprocessing* pada suatu pola dan mengembalikan suatu *array* yang berisi semua indeks dari kemunculan terakhir semua karakter ASCII. Contoh, untuk kata 'stima', a akan bernilai 4, i akan bernilai 2, dan seterusnya. Karakter yang tidak terdapat pada pola akan diisi dengan nilai -1.

Algoritma Boyer-Moore mmiliki kompleksitas kasus terburuk berupa O(nm+A). Namun, algoritma ini sangat cepat apabila digunakan untuk alfabet yang besar sehingga algoritma Boyer-Moore cocok untuk teks yang berbahasa Inggris dan tidak cocok untuk binary.

C. Levenshtein Distance

Dalam ilmu komputer, Levenshtein Distance merupakan tolak ukur yang digunakan untuk mengukur perbedaan antara dua *sequence of string*. Levenshtein Distance yang didapat antara dua kata adalah jumlah langkah minimum yang dibutuhkan untuk mengubah salah satu kata menjadi kata yang lainnya, dengan langkah tersebut berupa perubahan huruf yang mencakup penambahan, penghapusan, serta penggantian salah satu huruf dari kata yang diubah.

Secara matematis, Levenshtein Distance antara dua *string* a dan b dengan panjang |a| dan |b| dapat dijabarkan sebagai berikut

$$\operatorname{lev}(a,b) = \begin{cases} |a| & \text{if } |b| = 0, \\ |b| & \text{if } |a| = 0, \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) & \text{if } a[0] = b[0] \\ 1 + \min \begin{cases} \operatorname{lev}\left(\operatorname{tail}(a), b\right) \\ \operatorname{lev}\left(a, \operatorname{tail}(b)\right) \\ \operatorname{lev}\left(\operatorname{tail}(a), \operatorname{tail}(b)\right) \end{cases} & \text{otherwise,} \end{cases}$$

Gambar 2.4 Rumus Levenshtein Distance

Sumber:https://en.wikipedia.org/wiki/Levenshtein_distance

dengan *tail* dari suatu *string* x adalah *string* berisi semua kecuali karakter pertama dari x dan x[n] adalah karakter ke-n dari *string* x dengan index dimulai dari 0.

Contoh dari Levenshtein Distance dapat diambil dengan menggunakan kata *'kitten'* dan *'sitting*', dimana keduanya memiliki Levenshtein Distance sebesar 3. Hal ini dikarenakan langkah minimum yang dibutuhkan untuk mengubah salah satu dari kedua kata tersebut menjadi kata yang lainnya adalah

sebanyak 3 langkah. Penjabaran langkah minimum tersebut adalah sebagai berikut:

1. penggantian huruf ‘k’ dengan ‘s’

   kitten → **s**itten

2. penggantian huruf ‘i' dengan ‘e’

   sitt**e**n → sitt**i**n

3. penambahan huruf ‘g’ pada akhir kata

   sittin → sittin**g**

Pada Levenshtein Distance, terdapat beberapa *upper bound* dan *lower bound*. Adapun detail dari *upper bound* dan *lower bound* dari Levenshtein Distance adalah sebagai berikut:

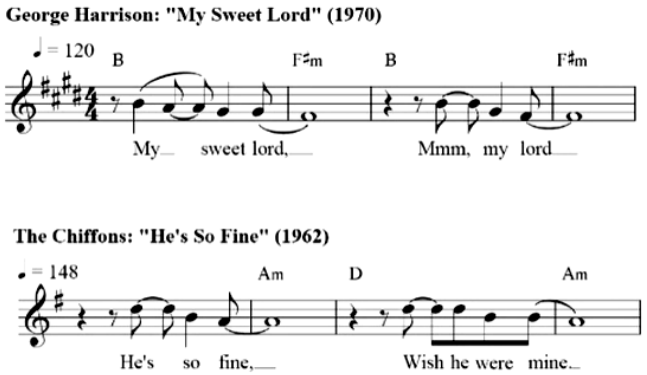
1. Nilai minimum dari Levenshtein Distance adalah sebesar perbedaan dari panjang kedua *string*
2. Nilai maksimum dari Levenshtein Distance adalah sebesar panjang dari *string* yang lebih panjang di antara kedua *string*
3. Nilai dari Levenshtein Distance adalah 0 jika dan hanya jika kedua *string* adalah sama
4. Jika kedua *string* berukuran sama, *upper bound*-nya merupakan Hamming Distance dari kedua *string*, dimana Hamming Distance merupakan jumlah posisi dimana simbol yang bersesuaian diantara kedua *string* berbeda
5. Levenshtein Distance antara dua *string* tidak lebih besar dari jumlah Levenshtein Distance kedua *string* tersebut dari *string* lain (*triangle inequality*)

III. Implementasi dan Pengujian

A. Gambaran Permasalahan

Pada makalah ini, implementasi dari algoritma *string matching* untuk mengidentifikasi plagiarisme pada karya musik akan diterapkan pada dua lagu yang pernah terlibat dalam kasus plagiarisme di dunia nyata. Kasus plagiarisme yang diungkit pada makalah ini melibatkan lagu “My Sweet Lord” milik George Harrison serta “He’s So Fine” milik The Chiffons. Pada kasus ini, hakim memutuskan untuk menjatuhkan hukuman berupa denda kepada George Harrison karena dianggap telah melakukan plagiarisme yang dilakukan secara tidak sadar, dimana ketidaksadaran ini berarti tidak ada bukti yang dapat mengaitkan George Harrison dengan The Chiffons sehingga tidak ada bukti bahwa George Harrison sebelumnya pernah mendengar lagu “He’s So Fine” milik The Chiffons.

Apabila lagu “My Sweet Lord” dan “He’s So Fine” dimainkan, akan terdengar bahwa keduanya mirip, tetapi tidak identik. Hal ini dikarenakan melodi dan *chord* yang digunakan hampir sama, tetapi tonalitas, tempo, struktur, instrumen, dan karakter musik dari kedua lagu tersebut berbeda. Karena makalah ini menitikberatkan pada identifikasi kemiripan melodi antarlagu, maka seterusnya makalah ini hanya akan membahas aspek melodi dari kedua lagu tersebut, terutama pada bagian *chorus*.



Gambar 3.1 Bagian chorus dari kedua lagu dalam notasi balok
Sumber: https://johnryle.com/?article=look-what-theyve-done-to-my-song-ma

Gambar di atas merupakan notasi balok dari 4 bar *chorus* lagu “My Sweet Lord” dan “He’s So Fine”. Selain dalam bentuk notasi balok, melodi dari suatu musik dapat dituliskan dalam bentuk notasi huruf dengan huruf C untuk do, D untuk re, E untuk mi, F untuk fa, G untuk sol, A untuk la, dan B untuk si. Notasi *sharp* dapat diindikasikan dengan simbol #, sedangkan *flat* dengan simbol ♭ untuk penulisannya. Dengan begitu, suatu musik dapat diidentifikasi dalam bentuk *string* yang merupakan perwujudan melodinya. Sehingga, melodi dari kedua lagu yang akan dicoba pada algoritma *string matching* pada makalah ini dapat dimodelkan menjadi *string* berupa hasil konversi menjadi notasi huruf sebagai berikut:

Judul Lagu	Notasi Huruf	Kunci
My Sweet Lord	B A A G# G# F# B B G# F# F#	E Maj
He’s So Fine	D D B A A D D D B B A	G Maj

Tabel 3.1 Pemodelan chorus kedua lagu dalam bentuk notasi huruf

Dari gambar 3.1 dan tabel 3.1, dapat dilihat bahwa kedua lagu tersebut memiliki pola melodi serta *chord progression* yang mirip. Akan tetapi, jika dibandingkan not per not, sekilas keduanya berbeda. Hal ini dikarenakan lagu “My Sweet Lord” dan “He’s So Fine” dimainkan pada kunci yang berbeda. “My Sweet Lord” dimainkan pada kunci E mayor, sementara “He’s So Fine” dimainkan pada kunci G mayor.

Perbedaan kunci atau nada dasar berperan besar dalam menentukan hasil identifikasi plagiarisme musik. Suatu lagu dapat dijiplak dan diubah nada dasarnya dengan cara ditransposisi sehingga ketika dicocokkan keduanya tidak akan terlihat mirip. Hal ini dikarenakan transposisi merupakan tindakan mengubah nada dari suatu karya musik dengan tetap menjaga hubungan antarnada tersebut sehingga intervalnya tidak berubah antara satu sama lain. Oleh karena itu, pada identifikasi plagiarisme ini perlu dilakukan transposisi terlebih dahulu agar kedua lagu berada pada kunci yang sama. Transposisi akan dilakukan terhadap lagu “My Sweet Lord” dengan menaikkan 1 ½ nada sehingga dimainkan pada kunci G Mayor.

Judul Lagu	Notasi Huruf	Kunci
My Sweet Lord	D C C B B A D D B A A	G Maj
He’s So Fine	D D B A A D D D B B A	G Maj

Tabel 3.2 Pemodelan chorus kedua lagu setelah ditranspose

B. Algoritma Penyelesaian

Pada permasalahan ini, terlebih dahulu dibuat fungsi yang dapat meng-*transpose* suatu melodi dari kunci x ke kunci y. Fungsi tersebut diimplementasikan pada fungsi 'tranpose' yang menghitung interval antara *initial key* dengan *final key* dan meng-*transpose* setiap not pada melodi dengan mengubahnya menjadi not dengan jarak sebanyak jarak awal ditambah interval yang sudah dicari sebelumnya. Penggantian not dengan not dengan jarak sebanyak interval dilakukan dengan mencari not pada *array* 'keys' dengan *index* yang sesuai dengan jarak yang dicari.

```
keys = [('A'), ('A#'), ('B'), ('C'), ('C#'), ('D'),
        ('D#'), ('E'), ('F'), ('F#'), ('G'), ('G#')]

def transpose(melody, initialkey, finalkey):
    print("Transposing melody 1 from key", initialkey, "to", finalkey)
    idxInitial = keys.index(initialkey)
    idxFinal = keys.index(finalkey)
    if (idxFinal > idxInitial):
        interval = idxFinal - idxInitial
    else:
        interval = idxInitial - idxFinal
    notes = melody.split()
    for i in range(len(notes)):
        idxNote = keys.index(notes[i])
        notes[i] = keys[(idxNote + interval) % 12]
    return(notes)
```

Gambar 3.2 Fungsi transpose

Perlu diperhatikan juga bahwa pada *array* 'keys', tidak terdapat not *flat* demi memudahkan penulisan dan pencarian. Oleh karena itu, dibutuhkan juga fungsi untuk mengonversi nada *flat* pada melodi menjadi bentuk identiknya dalam bentuk *sharp*. Fungsi ini bernama 'removeFlats' dan memanfaatkan *dictionary* bernama 'flatToSharp' yang berisikan *key* berupa nada *flat* serta *value* berupa bentuk *sharp* yang bersesuaian dengan nada *flat* tersebut.

```
flatToSharp = {
    'Bb': 'A#', 'Db': 'C#', 'Eb': 'D#', 'Gb': 'F#', 'Ab': 'G#'
}

def removeFlats(melody1, melody2):
    for flat in flatToSharp.keys():
        melody1 = melody1.replace(flat, flatToSharp[flat])
        melody2 = melody2.replace(flat, flatToSharp[flat])
```

Gambar 3.3 Fungsi removeFlats

Dengan adanya fungsi 'transpose' dan 'removeFlats', pencocokkan *string* dengan algoritma *string matching* dan Levenshtein Distance dapat dilakukan dengan menggunakan fungsi 'identifyMelodies'. Pada 'identifyMelodies', diterima parameter berupa dua *string* berisi melodi yang ingin dicocokkan serta kunci atau nada dasar dari kedua melodi tersebut. Pertama-tama, fungsi tersebut akan mengonversi nada *flat* dari *input* menjadi bentuk *sharp* dengan fungsi 'removeFlats' apabila memang ada nada *flat*. Setelah memastikan sudah tidak ada nada *flat*, fungsi akan memeriksa apakah kedua melodi tersebut sudah berada pada kunci atau nada dasar yang sama. Apabila belum, digunakan fungsi 'transpose' terhadap melodi pertama untuk mengubah melodi tersebut dari kunci awalnya sehingga sesuai dengan kunci dari melodi 2. Setelah kedua melodi sudah diubah sehingga berada pada nada dasar yang sama, dilakukan tahap pencocokkan dengan menggunakan algoritma Knuth-Morris-Pratt, Boyer-Moore, dan juga Levenshtein Distance. Apabila keduanya merupakan melodi yang persis sama, algoritma KMP dan BM akan menangkap kesamaan tersebut dan menampilkan pesan bahwa kedua melodi sama. Apabila tidak, dilakukan pencarian persentase dengan menggunakan Levenshtein Distance.

```
def identifyMelodies(melody1, key1, melody2, key2):
    # convert all flats to sharp to simplify search
    if key1 in flatToSharp:
        key1 = flatToSharp[key1]
    if key2 in flatToSharp:
        key2 = flatToSharp[key2]
    removeFlats(melody1, melody2)
    # check if both melodies are on different keys
    if key1 != key2:
        print("Melodies are in different keys")
        melody1 = ' '.join(transpose(melody1, key1, key2))
        print("Melody 1 transposed:", melody1)
    # check for match percentage
    verdictKMP = KMP(melody2, melody1)
    verdictBM = BM(melody2, melody1)
    if verdictKMP and verdictBM:
        print("Melodies are exactly the same")
    similarity = findSimilarity(melody1, melody2)
    print(f"Melodies are similar by {similarity * 100}%")
```

Gambar 3.4 Fungsi identifyMelodies

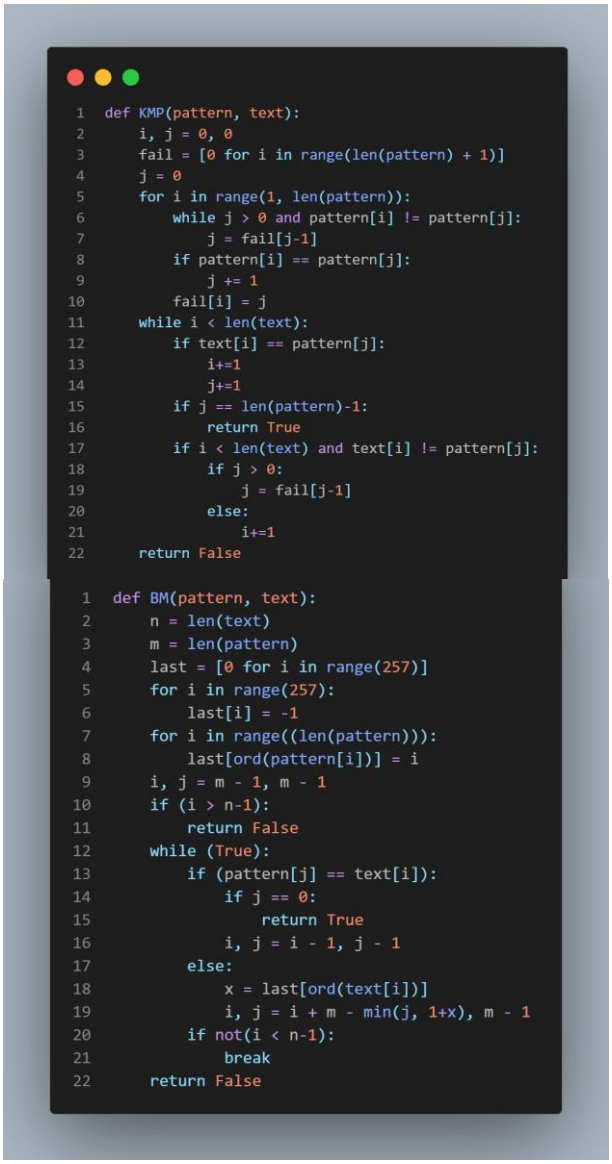
Algoritma Levenshtein Distance diterapkan pada fungsi 'levenshteinDistance' yang akan mengembalikan Levenshtein Distance antara suatu teks dan pattern, dimana pada kasus ini keduanya adalah melodi dalam bentuk *string*. Fungsi 'levenshteinDistance' akan dipanggil pada 'findSimilarity', dimana akan dicari *substring* dari teks dan pattern dengan *upper bound* berupa panjang dari *string* pattern. Setiap *substring* akan dicari Levenshtein Distancenya sehingga akan dihasilkan persentase berupa panjang dari pattern yang dijadikan *upper bound* dikurangi oleh jarak minimum yang didapat melalui Levenshtein Distance dibagi panjang pattern.

```
def levenshteinDistance(text, pattern):
    n = len(text)
    m = len(pattern)
    d = [[] for i in range(n+2)]
    for i in range(n+2):
        d[i] = [0 for i in range(m+2)]
    for i in range(n + 1):
        d[i][0] = i
    for j in range(m+1):
        d[0][j] = j
    for j in range(1, m+1):
        for i in range(1, n+1):
            if text[i-1] == pattern[j-1]:
                d[i][j] = d[i-1][j-1]
            else:
                d[i][j] = min(d[i-1][j]+1, min(d[i][j-1]+1, d[i-1][j-1]+1))
    return d[n][m]

def findSimilarity(text, pattern):
    substrings = divideSubstring(text, pattern)
    minDistance = len(pattern)
    for i in range(len(substrings)):
        distance = levenshteinDistance(substrings[i], pattern)
        if distance < minDistance:
            minDistance = distance
    return ((len(pattern) - minDistance)) / (len(pattern))
```

Gambar 3.6 Algoritma Levenshtein Distance

Adapun algoritma Knuth-Morris-Pratt serta Boyer-Moore yang digunakan untuk *string matching* pada implementasi adalah sebagai berikut, dengan keduanya menghasilkan *boolean* yang menandakan apakah kedua *string* berupa teks dan pattern yang dibandingkan sama atau tidak:



```python
def KMP(pattern, text):
    i, j = 0, 0
    fail = [0 for i in range(len(pattern) + 1)]
    j = 0
    for i in range(1, len(pattern)):
        while j > 0 and pattern[i] != pattern[j]:
            j = fail[j-1]
        if pattern[i] == pattern[j]:
            j += 1
        fail[i] = j
    while i < len(text):
        if text[i] == pattern[j]:
            i+=1
            j+=1
        if j == len(pattern)-1:
            return True
        if i < len(text) and text[i] != pattern[j]:
            if j > 0:
                j = fail[j-1]
            else:
                i+=1
    return False
```

```python
def BM(pattern, text):
    n = len(text)
    m = len(pattern)
    last = [0 for i in range(257)]
    for i in range(257):
        last[i] = -1
    for i in range((len(pattern))):
        last[ord(pattern[i])] = i
    i, j = m - 1, m - 1
    if (i > n-1):
        return False
    while (True):
        if (pattern[j] == text[i]):
            if j == 0:
                return True
            i, j = i - 1, j - 1
        else:
            x = last[ord(text[i])]
            i, j = i + m - min(j, 1+x), m - 1
        if not(i < n-1):
            break
    return False
```

Gambar 3.5 Algoritma KMP dan BM

Pada algoritma di atas, *preprocessing* pada algoritma Boyer-Moore dilakukan dengan menjadikan fungsi L() pada baris 4 hingga 8, dimana 'last' merupakan *array* berisi kemunculan terakhir dari setiap karakter yang berupa karakter ASCII.

Sesuai dengan kasus yang terdapat pada bagian 'Gambaran Permasalahan', akan dimasukkan *input* berupa melodi serta kunci atau nada dasar dari lagu "My Sweet Lord" dan "He's So Fine". Untuk membuktikan bahwa fungsi 'transpose' dapat bekerja dengan baik, akan digunakan *input* melodi serta kunci mengikuti data pada tabel 3.1, dimana lagu "My Sweet Lord" belum dinaikkan sebanyak 1 ½ sehingga kedua lagu masih berada pada kunci atau nada dasar yang berbeda.

```python
# "my sweet lord" (MSL) and "hes so fine" (HSF)
melodyMSL = "B A A G# G# F# B B G# F# F#"
keyMSL = "E"
melodyHSF = "D D B A A D D D B B A"
keyHSF = "G"
identifyMelodies(melodyMSL, keyMSL, melodyHSF, keyHSF)
```

Gambar 3.7 Input "My Sweet Lord" dan "He's So Fine"

Setelah dijalankan, didapatkan *output* dari fungsi 'identifyMelodies' dari kedua lagu tersebut adalah sebagai berikut:

```
C:\Users\Sarah Azka A\Desktop>py makalah.py
Melodies are in different keys
Transposing melody 1 from key E to G
Melody 1 transposed: D C C B B A D D B A A
Melodies are similar by 71.42857142857143%
```

Gambar 3.8 Hasil

Berdasarkan gambar 3.8, terdapat bahwa fungsi 'identifyMelodies' berhasil mengidentifikasi adanya perbedaan kunci atau nada dasar serta melakukan *tranpose* untuk menyamakan perbedaan kunci tersebut. Tak hanya itu, dapat dilihat bahwa hasil *transpose* dari melodi 1 menggunakan fungsi 'transpose' bersesuaian dengan hasil *transpose* manual yang dilakukan sebelumnya pada tabel 3.2, dengan lagu "My Sweet Lord" dikonversi dari kunci E mayor sehingga berakhir pada kunci G mayor. Selain itu, kemiripan antara melodi 1 yang adalah 4 bar *chorus* dari "My Sweet Lord" dengan melodi 2 yang adalah 4 bar *chorus* dari "He's So Fine" terbukti memiliki kemiripan sebesar kurang lebih 71.4% apabila dibulatkan.

IV. Kesimpulan

Algoritma *string matching* seperti Knuth-Morris-Pratt, Boyer-Moore, dan juga algoritma Levenshtein Distance dapat digunakan untuk beragam persoalan dalam kehidupan. Pada makalah ini, dapat dilihat bahwa algoritma tersebut sangat bermanfaat dalam penentuan dan identifikasi plagiarisme musik. Melalui konsep pencocokkan *string*, perlindungan hak cipta, penghormatan hak kekayaan intelektual, serta penentuan plagiarisme dari suatu karya musik dapat dibantu dengan memanfaatkan algoritma yang ada untuk menghasilkan bukti yang dapat mendukung dugaan terhadap karya yang tertuduh plagiasi atau bahkan sebagai bukti yang dapat melindungi suatu karya yang tertuduh suatu karya yang tertuduh plagiasi dari dugaan yang salah.

V. Saran

Plagiarisme bukanlah suatu tindakan yang terpuji. Penggunaan teknologi dan algoritma seperti pemanfaatan konsep *string matching* sebagai alat bantu dalam proses pendeteksian dan identifikasi plagiarisme pada karya-karya seperti karya musik penting untuk diterapkan demi

mengungkap kebenaran dibalik kasus plagiarisme. Karena pada dasarnya, sebaik-baiknya suatu karya ialah karya yang bersifat orisinil dan dibuat berdasarkan ide milik diri sendiri, bukan milik orang lain.

Akan tetapi, perlu diingat bahwa kemiripan antara suatu karya dengan karya lainnya juga dapat terjadi atas dasar ketidaksengajaan. Oleh karena itu, jika terdapat suatu karya yang terduga merupakan hasil plagiat karena mirip dengan karya lainnya, hendaknya dilakukan pengecekkan yang menyeluruh terhadap hal-hal yang meliputi karya yang tertuduh tersebut demi menghindari adanya kemungkinan bahwa karya tersebut merupakan hasil dari kemiripan yang dilakukan secara tidak sadar atau tidak sengaja.

LINK VIDEO YOUTUBE

http://bit.ly/VideoMakalah13520083

UCAPAN TERIMA KASIH

Segala puji bagi Allah SWT yang telah memberikan kemudahan serta kelancaran bagi penulis sehingga makalah ini dapat diselesaikan secara tepat waktu. Penulis mengucapkan terima kasih yang sebesar-besarnya kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen mata kuliah IF2211 Strategi Algoritma kelas K02 yang telah membimbing dan membantu dalam penulisan makalah ini. Penulis juga ingin mengucapkan terima kasih kepada orang tua penulis serta teman-teman penulis atas dukungan yang telah diberikan selama pengerjaan makalah ini. Selain itu, penulis menyadari masih terdapat kekurangan dan kesalahan kata dalam makalah ini sehingga penulis meminta maaf. Penulis berharap makalah ini dapat menjadi manfaat bagi para pembacanya dan dapat dimanfaatkan untuk hal-hal yang bermanfaat bagi masyarakat luas.

REFERENSI

[1] Munir, Rinaldi. "Pencocokan String (String/Pattern Matching)". https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Pencocokan-string-2021.pdf. Diakses tanggal 15 Mei 2022, pukul 13.07 GMT+7.

[2] https://en.wikipedia.org/wiki/Levenshtein_distance. "Levenshtein Distance". Diakses tanggal 20 Mei 2022, pukul 15.46 GMT+7.

[3] Ryle, John. "Look what they've done to my song, ma". https://johnryle.com/?article=look-what-theyve-done-to-my-song-ma. Diakses tanggal 20 Mei 2022, pukul 12.10 GMT+7.

[4] McGuire, Patrick. "What Is 'Subconscious Plagiarism?' Just Ask George Harrison". https://flypaper.soundfly.com/discover/what-is-subconscious-plagiarism-just-ask-george-harrison/. Diakses tanggal 19 Mei 2022, pukul 19.07 GMT+7.

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, 20 Mei 2022

Sarah Azka Arief
13520083