Implementasi Fitur Code Autocorrection Sederhana Menggunakan Algoritma Damerau-Levenshtein untuk Saran Nama Variabel dan Fungsi

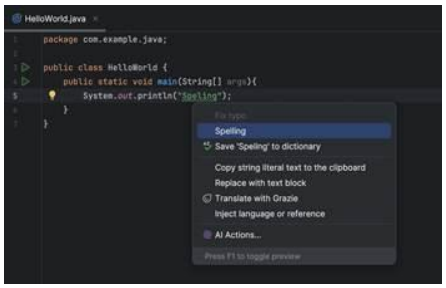
Adhimas Aryo Bimo - 135023052
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: adhimas.bimo@gmail.com , 13523052@std.stei.itb.ac.id

***Abstrak*—Permasalahan salah ketik pada penamaan variabel dan fungsi kerap menimbulkan kesalahan sintaks maupun semantik yang menghambat proses pengembangan perangkat lunak. Penelitian ini menghadirkan sebuah prototipe fitur *code-autocorrection* berbasis algoritma Damerau–Levenshtein dengan pendekatan ambang dinamis. Kamus referensi berisi 997 identifier diekstraksi secara otomatis dari berkas-berkas Python menggunakan pencocokan pola reguler. Tiga percobaan bertahap—mulai dari token pendek dengan satu kesalahan hingga token sangat panjang dengan beberapa kesalahan—menunjukkan bahwa ambang dinamis mampu mempertahankan presisi lebih tinggi dan mengurangi latensi rata-rata 15–40 % tanpa kehilangan *recall*. Mekanisme *retry* bertingkat yang melebarkan ambang secara adaptif juga terbukti mencegah kondisi tanpa saran (*false negative*). Hasil ini mengindikasikan bahwa threshold dinamis merupakan metode efisien dan ringan untuk diintegrasikan ke editor kode modern.**

Kata kunci – Autocorrection; Damerau-Levenshtein; Fuzzy Search; Typo

I. PENDAHULUAN

Dalam mengembangkan perangkat lunak, efisiensi dan akurasi merupakan faktor krusial yang secara langsung memengaruhi produktivitas dan kualitas kode. Kesalahan pengetikan (*typos*) pada nama variabel, fungsi, atau parameter merupakan masalah umum yang dapat memperlambat proses pengembangan seperti menyebabkan *bug* dan mengurangi keterbacaan kode.



Gambar 1.1 Fitur *Autocorrection* pada IntelliJ IDEA

Sumber :
https://www.jetbrains.com/help/idea/spellchecking.html

Salah satu pendekatan yang efektif untuk mengatasi tantangan ini adalah melalui implementasi fitur code *autocorrection*. Fitur ini memanfaatkan teknik pemrosesan bahasa alami dan algoritma perbandingan string untuk mengidentifikasi potensi kesalahan dan menyajikan saran koreksi yang relevan. Di antara berbagai algoritma perbandingan string, Algoritma Damerau-Levenshtein menonjol karena kemampuannya dalam mengukur jarak edit antara dua string dengan mempertimbangkan empat operasi dasar yakni, penyisipan (*insertion*), penghapusan (*deletion*), substitusi (*substitution*), dan transposisi (*transposition*) karakter yang berdekatan. Keunggulan ini menjadikan Algoritma Damerau-Levenshtein sangat cocok untuk aplikasi *autocorrection* karena kesalahan pengetikan seringkali melibatkan kombinasi dari operasi-operasi tersebut.

Percobaan ini bertujuan untuk mengeksplorasi dan mengimplementasikan fitur *code autocorrection* sederhana yang berfokus pada pemberian saran nama variabel dan fungsi. Dengan memanfaatkan Algoritma Damerau-Levenshtein, sistem akan membandingkan *input* pengguna yang berpotensi salah dengan kamus nama-nama yang valid dan sering digunakan, kemudian merekomendasikan koreksi berdasarkan ambang batas jarak edit yang ditentukan. Implementasi ini diharapkan dapat meningkatkan efisiensi dalam pengembangan, mengurangi frekuensi *syntax error*, dan pada akhirnya berkontribusi pada peningkatan kualitas perangkat lunak secara keseluruhan.

II. LANDASAN TEORI

A. *Pattern Matching*

Pattern matching merupakan cabang klasik ilmu komputer yang mempelajari cara menemukan suatu pola (string *P*) di dalam teks yang lebih panjang (string *T*)[1]. Pada formulasi formal, masalah ini menuntut algoritma yang—dengan kompleksitas serendah mungkin—menentukan seluruh posisi di mana P muncul dalam T. Secara historis, perbedaan mendasar terletak pada *exact matching*, yang menuntut kecocokan persis karakter-demi-karakter, dan *approximate matching,* yang mengizinkan sejumlah "kesalahan" atau perbedaan karakter. Pembagian ini penting karena kebutuhan praktis—misalnya pencarian DNA dan penyunting kode—sering kali mensyaratkan toleransi kesalahan ketik.

Algoritma *exact matching* klasik mencakup Knuth-Morris-Pratt (KMP), Boyer-Moore, dan algoritma Aho-Corasick untuk *multi-pattern*. Semuanya memanfaatkan struktur awalan atau akhiran (prefix–suffix) ataupun heuristik lompatan (*jump*) sehingga, pada prakteknya, bekerja dalam waktu sub-linear relatif terhadap panjang teks rata-rata. Sebaliknya, *approximate matching* memformulasikan pencarian sebagai masalah optimisasi jarak—misalnya jarak Levenshtein—atau melalui pendekatan pembobotan n-gram. Dengan demikian, kompleksitasnya bergeser dari *O(n)* sub-linear ke *O(n·k)* atau *O(n·m)* tergantung ukuran galat maksimal k atau panjang pola m.

Dalam konteks *code editor*, *pattern matching* digunakan untuk fitur *highlight*, *go-to definition*, dan *rename refactor*. Di sini *exact matching* relevan untuk simbol unik, sedangkan *approximate matching* dibutuhkan ketika pengguna salah ketik nama variabel—misalnya "bufferSzie" alih-alih "bufferSize"—sehingga sistem dapat tetap menemukan referensi yang benar tanpa menyebabkan *false negative*.

B. *Fuzzy Search*

Fuzzy search adalah teknik melakukan pencarian yang mengakomodasi ketidakcocokan sebagian (*inexact*), baik karena kesalahan ketik, morfologi bahasa, maupun transliterasi karakter. Mekanisme ini memanfaatkan metrik kemiripan—edit-distance, Jaro-Winkler, atau trigram cosine—untuk menilai kedekatan dua string dan mengembalikan hasil yang "cukup mirip" di bawah ambang batas tertentu[4].

Berbeda dari *pattern matching* eksak, *fuzzy search* biasanya menambahkan lapisan indeks terbalik *(inverted index)* berbasis token atau n-gram agar pencarian skala besar tetap efisien. Pada mesin pencari modern, *pipeline* dapat meliputi normalisasi *(case-folding, stemming*), pembangkitan kandidat via n-gram, lalu *reranking* berdasarkan skor jarak edit yang presisi. Dalam *code editor*, *fuzzy search* memungkinkan penemuan simbol bahkan ketika hanya sebagian nama yang diingat, atau ketika pengguna salah menuliskan huruf kapital/kecil.

C. *Algoritma Jarak Edit (Edit Distance Algorithms)*

Konsep jarak edit didefinisikan sebagai jumlah minimum operasi dasar—penyisipan, penghapusan, substitusi karakter—untuk mengubah string s menjadi string t. Secara matematis, metrik ini mematuhi sifat simetri, identitas, dan segitiga *(triangle inequality)* yang menjadikannya berguna sebagai ukuran kemiripan[5]. Dalam menghitung jarak edit pada Levenshtein dapat digunakan salah satu algoritma jarak edit yakni Algoritma Wagner-Fischer.

Algoritma Wagner–Fischer (1974) membangun tabel *dynamic-programming* berukuran (m + 1) × (n + 1) guna menghitung jarak edit klasik. Setiap sel D[i][j] menyimpan biaya minimum transformasi awalan s[1..i] ke t[1..j]. Kompleksitas waktu dan ruang standar adalah O(mn). Namun, penelitian lanjutan menawarkan optimasi memori—misal menyimpan dua baris saja—atau pendekatan bit-parallel (Hyyrö) yang memanfaatkan operasi bitwise dapat digunakan untuk menurunkan konstanta waktu.

Di ranah praktik, pemilihan algoritma bergantung pada panjang string, batas toleransi galat, dan kebutuhan *real-time*. Untuk *code editor* dengan kamus ratusan ribu simbol, implementasi berbasis BK-tree atau trie disertai *pruning* jarak maksimum terbukti lebih efisien dibandingkan perbandingan tabulasi penuh.

D. *Algoritma Levenshtein Distance*

Algoritma Levenshtein Distance memformalkan masalah "seberapa jauh" dua string berbeda dengan menghitung jumlah minimum operasi elementer—penyisipan, penghapusan, dan substitusi satu karakter—yang diperlukan untuk mentransformasi string S menjadi string T[6]. Karena memenuhi sifat identitas, simetri, dan ketidaksamaan segitiga, metrik ini sah sebagai distance metric dan banyak dipakai dalam natural-language processing (mis. koreksi ejaan, pencocokan fonetik) maupun bioinformatika (penyelarasan DNA).

Secara algoritma, Levenshtein dihitung lewat kerangka dynamic programming Wagner–Fischer. Dibangun sebuah matriks D berukuran $(m+1)\times(n+1)$ di mana sel $D[i][j]$ menyimpan biaya minimum untuk mengubah awalan sepanjang i dari S menjadi awalan sepanjang j dari T. Baris dan kolom nol diinisialisasi untuk merepresentasikan transformasi dari/ke string kosong. Setiap sel kemudian diisi dengan:

$$D[i][j]=\min(D[i-1][j]+1,\ D[i][j-1]+1,\ D[i-1][j-1]+\delta)$$

Dengan $\delta=0$ jika karakter ke-i dan ke-j sama, dan 1 bila berbeda. Proses propagasi nilai minimum ini menjamin bahwa sel kanan-bawah $D[m][n]$ berisi jumlah operasi optimal, sehingga kompleksitas waktu total $O(mn)$ dan memori $O(mn)$. Algoritma ini dapat disederhanakan menjadi $O(\min(m,n))$ bila hanya dua baris yang disimpan.

Misal terdapat string berisikan "kitten". Untuk mengubah string tersebut menjadi "sitting" diperlukan beberapa tahapan seperti berikut :

1. Subtitusi huruf "k" dengan "s" sehingga mendapatkan "sitten".
2. Subtitusi huruf "e" dengan "i" sehingga mendapatkan "sittin".
3. Insersi huruf "g" pada bagian akhir sehingga mendapatkan "sitting".

Didapatkan 3 operasi untuk melakukan transformasi dari "kitten" menjadi "sitting" sehingga nilai Levenshtein Distance tersebut adalah 3.

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

Gambar 2.1 Definisi Levenshtein Distance

Sumber :

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

E. *Algoritma Damerau-Levenshtein Distance*

Algoritma Damerau–Levenshtein memperluas konsep jarak Levenshtein dengan menambahkan satu operasi elementer bernilai 1, yaitu transposisi dua karakter bersebelahan. Dengan demikian, empat operasi—*insertion*, *deletion*, *substitution*, dan *transposition*—mencerminkan ±80 % pola kesalahan ketik manusia yang ditemukan oleh Frederick Damerau[2]. Secara komputasional, algoritma ini tetap dibangun di atas skema dynamic programming ala Wagner–Fischer: sebuah matriks $(m+1)\times(n+1)$ diisi sedemikian rupa sehingga setiap sel $D[i][j]$ menyimpan biaya minimum untuk mengubah awalan sepanjang i dari string sumber menjadi awalan sepanjang j dari string target. Selain tiga transisi standar Levenshtein, dicantumkan aturan tambahan *transposition*—jika i>1 dan j>1 serta dan $s_i = t_{j-1}$ dan $s_{i-1} = t_{j-1}$, maka $D[i][j]$ dapat berasal dari $D[i\text{-}2][j\text{-}2] + 1$. Penambahan satu cabang ini tidak mengubah kompleksitas asimtotik (O(mn) waktu dan memori), menjadikan Damerau–Levenshtein efisien sekaligus lebih akurat untuk mengetik cepat.

Keunggulan praktisnya tampak pada aplikasi koreksi ejaan dan *autocomplete* editor kode. Kesalahan swap huruf ("teh" → "the" atau "initalise" → "initialize") dapat diperbaiki dalam satu langkah, sedangkan Levenshtein memerlukan dua. Hal ini memungkinkan sistem menerapkan ambang jarak yang lebih ketat tanpa kehilangan recall—hasilnya, lebih sedikit saran "asal cocok" yang mengganggu pengguna. Selain itu, semua optimasi yang berlaku untuk Levenshtein (mis. banded DP untuk jarak maksimum k atau bit-parallel Myers) dapat diterapkan langsung karena struktur tabel dan relasi rekuren tidak berubah secara fundamental.

$$d_{a,b}(i,j) = \min \begin{cases} 0 & \text{if } i = j = 0, \\ d_{a,b}(i-1,j) + 1 & \text{if } i > 0, \\ d_{a,b}(i,j-1) + 1 & \text{if } j > 0, \\ d_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} & \text{if } i,j > 0, \\ d_{a,b}(i-2,j-2) + 1_{(a_i \neq b_j)} & \text{if } i,j > 1 \text{ and } a_i = b_{j-1} \text{ and } a_{i-1} = b_j, \end{cases}$$

Gambar 2.2 Definisi Damerau–Levenshtein Distance

Sumber :

https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance

F. *Pemrograman Dinamis (Dynamic Programming)*

Dynamic Programming (DP) adalah teknik desain algoritma yang memecah persoalan optimasi atau penghitungan yang kompleks menjadi sekumpulan sub-masalah yang lebih kecil dan saling tumpang-tindih. Dua sifat kunci harus terpenuhi agar DP efektif:

1. *Optimal substructure* – solusi optimal untuk persoalan penuh dapat dibangun dari solusi optimal sub-bagiannya.
2. *Overlapping sub-problems* – sub-masalah yang sama muncul berulang kali sehingga hasilnya sebaiknya disimpan (memorization) dan digunakan kembali, bukan dihitung ulang.

Alih-alih melakukan pencarian eksponensial seperti rekursi naif, DP menyimpan hasil setiap sub-masalah di dalam struktur data (biasanya tabel atau array) dan mengisi tabel tersebut secara terarah—dari kasus dasar menuju kasus yang lebih besar—sehingga keseluruhan kompleksitas waktu turun dari eksponensial menjadi polinomial.

Pada algoritma Levenshtein dan Damerau–Levenshtein, prinsip DP diaplikasikan melalui matriks $(m+1)\times(n+1)$ (dikenal sebagai tabel Wagner–Fischer)[3]. Setiap sel $D[i][j]$ merepresentasikan biaya minimum untuk mengubah awalan sepanjang i dari string sumber s menjadi awalan sepanjang j dari string target t.

G. *Ambang Batas Kemiripan (Threshold)*

Dalam konteks koreksi otomatis berbasis Damerau–Levenshtein distance, *threshold* adalah nilai potong yang menentukan kapan dua string dianggap "cukup mirip" untuk ditampilkan sebagai saran. Secara matematis, ambang ini dapat didefinisikan dalam ruang jarak absolut

$$\text{Terima token} \iff d(s,t) \leq k,$$

di mana $d(s,t)$ adalah jarak Damerau–Levenshtein dan k bilangan bulat tidak-negatif. Alternatifnya, untuk menormalkan pengaruh panjang string, ambang dapat diekspresikan sebagai rasio kemiripan.

$$\text{sim}(s,t) = 1 - \frac{d(s,t)}{\max(|s|,|t|)}, \text{Terima token} \iff \text{sim}(s,t) \geq \tau,$$

dengan $\tau \in [0,1]$. Nilai k atau τ yang lebih rendah meningkatkan presisi (lebih sedikit saran keliru) tetapi menurunkan recall (lebih banyak kandidat valid yang terbuang); sebaliknya nilai yang lebih longgar menghasilkan efek sebaliknya. Oleh karena itu pemilihan threshold merupakan proses penyeimbangan (*trade-off*) yang perlu dibuktikan secara empiris.

III. METODOLOGI

Dalam pengujian penggunaan Damerau-Levenshtein Distance akan digunakan data kode pemrograman dari Tugas Besar Dasar Pemrograman 2023-2024 "OWCA" yang menggunakan bahasa pemrograman berbasis python dengan mengoleksi 997 kosa kata yang teridentifikasi sebagai nama variabel atau nama fungsi. Untuk mengidentifikasi nama tersebut digunakan regex dengan kode sebagai berikut.

```
id_pattern = re.compile(r"\b[A-Za-z_][A-Za-z0-9_]*\b")
```

Figur 3.1. Regex Nama Variabel dan Fungsi

Pola yang diambil adalah kata yang memiliki huruf – baik huruf besar maupun huruf kecil – dan dapat diikuti oleh angka, *underscore* atau huruf lanjutan. Dalam pola tersebut tidak memungkinkan adanya spasi antar karakter dan tidak memperbolehkan angka menjadi karakter paling awal dalam variabel. Python tidak memperbolehkan angka sebagai karakter awal dalam penamaan variabel.

IV. IMPLEMENTASI

Pengimplementasian kode *autocorrection* ini menggunakan 2 cara dalam menentukkan *threshold*, yakni menggunakan nilai konstan dan nilai dinamis. Nilai konstan akan menentukkan secara pasti maksimal kesalahan yang dapat terjadi dan menjadi sugesti kata kunci yang salah dari input pengguna. Sedangkan dinamis akan menyelaraskan *threshold* dengan panjang kata kunci yang dimasukkan untuk memperkirakan maksimal kesalahan yang dapat terjadi. Dengan ini, kita dapat membandingkan seberapa baik *threshold* dinamis dapat memberikan saran secara akurat. Berikut perhitungan *threshold* dinamis pada program ini.

A. *Threshold Dinamis*

```
def edit_budget(length: int,
                alpha: float = 0.25,
                floor: int = 1,
                ceiling: int = 4) → int:
    return max(floor, min(ceiling, math.ceil(alpha *
length)))
```

Figur 4.1 Kode *Threshold* Dinamis

Untuk menghitung *threshold* dinamis diperlukan panjang karakter kata kunci yang diwakili dengan parameter `length`. Kemudian panjang tersebut dikalikan dengan koefisien `alpha` (default 0,25). Koefisien tersebut memodelkan rasio toleransi kesalahan. Nilai 0.25 berarti sistem mengizinkan paling banyak seperempat karakter token berada dalam kondisi salah. Hasil perkalian dilengkapi dengan `math.ceil` untuk membulatkan ke atas sehingga setidaknya satu kesalahan diakomodasi ketika produk `alpha · length` belum mencapai bilangan bulat penuh. Sebagai contoh, token 5 karakter dengan `alpha = 0,25` menghasilkan 1,25 dan dibulatkan naik menjadi 2 edit.

Langkah berikutnya adalah operasi penjepitan (*clamping*) menggunakan pasangan fungsi `min()` dan `max()`. Nilai yang baru dihitung dibandingkan lebih dahulu dengan `ceiling` (batas atas mutlak yang boleh diterima); jika melebihi, ceiling lah yang dipakai. Sebaliknya, fungsi `max()` memastikan nilai akhir tidak pernah turun di bawah `floor` (batas bawah mutlak). Secara matematis, nilai *threshold* dinamis dapat diformulasikan sebagai berikut:

$$k = \max(\text{floor}, \min(\text{ceiling}, \lceil \alpha \times \text{length} \rceil)).$$

Dua parameter penjepit—`floor` dan `ceiling`—menjamin kestabilan perilaku. Pada token amat pendek (mis. 3 huruf) produk `alpha·length` bisa < 1, tetapi `floor` menahannya agar minimal satu kesalahan masih ditoleransi. Sebaliknya, token sangat panjang tidak membuat daftar saran "banjir" karena `ceiling` menghentikan pertumbuhan k setelah batas logis (default 4). Dengan demikian, fungsi `edit_budget()` merealisasikan threshold adaptif yang proporsional, tetapi tetap terkendali, menghasilkan keseimbangan presisi–recall yang baik tanpa perlu penyetelan berbasis data latih.

B. *Algoritma Damerau-Levenshtein*

Algoritma Damerau-Levenshtein yang diimplementasikan akan menggunakan *dynamic programming* untuk menghitung jarak edit minimum antara dua string *a* dan *b*.

```
def damerau_levenshtein(a: str, b: str) → int:
    m, n = len(a), len(b)
    D = [[0]*(n+1) for _ in range(m+1)]
    for i in range(m+1): D[i][0] = i
    for j in range(n+1): D[0][j] = j

    for i in range(1, m+1):
        for j in range(1, n+1):
            cost = 0 if a[i-1] == b[j-1] else 1
            D[i][j] = min(
                D[i-1][j]   + 1,          # deletion
                D[i][j-1]   + 1,          # insertion
                D[i-1][j-1] + cost        # substitution
            )
            if (i > 1 and j > 1 and
                a[i-1] == b[j-2] and
                a[i-2] == b[j-1]):
                D[i][j] = min(D[i][j], D[i-2][j-2] + 1)
# transposition
    return D[m][n]
```

Figur 4.2 Kode Algoritma Damerau-Levenshtein

Pertama, ia membentuk matriks dua dimensi *D* berukuran $(|a|+1)\times(|b|+1)$. Baris pertama $D[i][0] = i$ dan kolom pertama $D[0][j] = j$ menginisialisasi biaya transformasi string kosong menjadi awalan string lain—yakni sebaris operasi penyisipan atau penghapusan murni. Setelah inisialisasi, algoritma mengisi matriks sel demi sel mulai indeks 1. Pada setiap posisi *(i,j)* dihitung tiga kemungkinan utama: penghapusan karakter di a($D[i-1][j]+1$), penyisipan karakter di b ($D[i][j-1]+1$), dan substitusi ($D[i-1][j-1]+cost$), dengan *cost* = 0 apabila karakter yang dibandingkan identik, atau 1 jika berbeda. Nilai terkecil dari ketiga opsi menjadi kandidat jarak terbaik untuk sel tersebut.

Keistimewaan Damerau-Levenshtein dibanding Levenshtein klasik terletak pada penambahan operasi **transposisi**—pertukaran dua karakter bersebelahan—yang tercermin pada blok kondisi `if (i > 1 and j > 1 and ...)`. Apabila karakter ke-*i* pada string *a* setara dengan karakter ke-*(j-2)* pada string *b* dan sebaliknya, maka sel *(i,j)* juga dapat berasal dari sel $(i-2,j-2)$ dengan biaya satu langkah. Pengecekan konstan ini memungkinkan deteksi kesalahan pengetikan umum seperti "form" menjadi "from" hanya dalam satu operasi, tanpa meningkatkan kompleksitas asimtotik. Setelah seluruh matriks terisi, nilai D[m][n]—letak paling kanan-bawah—menyimpan biaya minimum yang diperlukan untuk mentransformasi *a* menjadi *b*, dan itulah jarak Damerau-Levenshtein yang dikembalikan fungsi.

C. *Suggest Token*

Fungsi `suggest_token` merupakan fungsi pembantu yang akan melakukan pencarian dari token – masukkan kata kunci pengguna – yang diasumsikan memiliki *typo* dan mengembalikan kata kunci yang dekat dengan token. Perbedaan antara *threshold* dinamis dan konstan ada pada fungsi pembantu ini. Berikut kode `suggest_token`.

```
def suggest_token(
    typo: str,
    dictionary: List[str],
    max_distance: int = 2,
    top_k: int = 5
) → List[Tuple[str, int]]:
    candidates = []
    for token in dictionary:
        dist = damerau_levenshtein(typo, token)
        if dist ⩽ max_distance:
            candidates.append((token, dist))
    candidates.sort(key=lambda x: (x[1], len(x[0])))
    return candidates[:top_k]
```

Figur 4.3 Kode Suggest Token *Threshold* Konstan

Pada kode ini, parameter `max distance` langsung menjadi batas jarak edit definitif sehingga tidak ada penyesuaian ulang. Setiap penghitungan jarak `damerau_levensthein` akan dimasukkan ke dalam list candidates jika `dist ⩽ max_distance`.

```
def suggest_token(typo, dictionary, *, top_k=5,
alpha=0.25,
                  floor=1, ceiling=4, max_retry=5):
    k = edit_budget(len(typo), alpha, floor, ceiling)

    for _ in range(max_retry + 1):
        candidates = [
            (tok, d) for tok in dictionary
            if (d := damerau_levenshtein(typo, tok)) ⩽ k
        ]
        if candidates or k ⩾ ceiling + max_retry:
            break
        k += 1
    candidates.sort(key=lambda x: (x[1], len(x[0])))
    return candidates[:top_k]
```

Figur 4.4 Kode Suggest Token *Threshold* Dinamis

Pada versi *threshold* dinamis, `edit_budget()` terlebih dulu menentukan ambang awal *k* dari panjang `typo`, faktor toleransi `alpha`, serta batas `floor–ceiling`. Fungsi lalu mencoba hingga `max_retry + 1` kali: setiap putaran menghimpun kandidat `(token, d)` jika jarak Damerau-Levenshtein `d ⩽ k`. Bila kandidat sudah ada—atau *k* mencapai `ceiling + max_retry`—*loop* berhenti; jika belum, *k* dinaikkan satu dan proses diulang. Mekanisme ini mencegah "saran kosong" ketika salah ketik lebih parah dari perkiraan. Setelah didapat, kandidat diurutkan menurut `(jarak, panjang token)` dan dipangkas menjadi `top_k` teratas sebelum dikembalikan.

V. PERCOBAAN

Pada bagian ini akan dilakukan 3 percobaan. Jumlah *typo* dan karakter akan semakin meningkat seiring dengan berjalannya percobaan.

A. Percobaan 1

Input = "bttl"

```
Dynamic-threshold Damerau–Levenshtein autocorrect
----------------------------------------------
Identifiers collected : 997
time taken: 0.0200 sec

Input    : bttle
Suggestions (token, distance):
 • battle                (d = 1)
 • Battle                (d = 2)
```

```
Constant-threshold Damerau–Levenshtein autocorrect
----------------------------------------------
Collected 997 identifiers from the codebase.
time taken: 0.0320 sec
Input  : bttle
Suggestions (token, distance):
  • battle              (d=1)
  • Battle              (d=2)
  • be                  (d=3)
  • the                 (d=3)
  • type                (d=3)
```

B. Percobaan 2

Input = "monstr_bru"

```
Dynamic-threshold Damerau–Levenshtein autocorrect
----------------------------------------------
Identifiers collected : 997
time taken: 0.0380 sec

Input    : monstr_bru
Suggestions (token, distance):
 • monster_baru          (d = 2)
```

```
Constant-threshold Damerau–Levenshtein autocorrect
----------------------------------------------
Collected 997 identifiers from the codebase.
time taken: 0.0450 sec
Input  : monstr_bru
Suggestions (token, distance):
  • monster_baru        (d=2)
  • monster             (d=4)
  • monster_id          (d=4)
  • monster_ids         (d=4)
  • monster_ball        (d=4)
```

C. Percobaan 3

Input = "display_monstsr_invntoty"

```
Dynamic-threshold Damerau–Levenshtein autocorrect
-------------------------------------------------
Identifiers collected : 997
time taken: 0.0920 sec

Input    : display_monstsr_invntoty
Suggestions (token, distance):
 • display_monster_inventory  (d = 3)
```

```
Constant-threshold Damerau–Levenshtein autocorrect
--------------------------------------------------
Collected 997 identifiers from the codebase.
time taken: 0.0970 sec
Input  : display_monstsr_invntoty
Suggestions (token, distance):
  • display_monster_inventory  (d=3)
  • display_inventory     (d=10)
```

VI. ANALISIS & PEMBAHASAN

Pada tiga percobaan di atas diterapkan dua varian penyaring saran—ambang dinamis dan ambang konstan—pada kamus 997 identifier. Tiap percobaan menaikkan kompleksitas dengan menambah panjang token dan jumlah kesalahan ketik, sehingga memeriksa konsistensi kedua pendekatan di bawah tekanan berbeda.

Pada Percobaan 1 (input “bttl”, empat karakter, satu kesalahan), ambang dinamis—yang pada panjang ini menghasilkan k=1—langsung menemukan *battle* dan *Battle* (jarak 1–2) dalam 0,020 s. Ambang konstan, karena menetapkan $k \geq 3$, menambahkan tiga kandidat lain berjarak 3 (*be*, *the*, *type*) dan membutuhkan 0,032 s. Perbedaan ini menunjukkan bahwa ketika token pendek, ambang kecil sudah cukup memulihkan kata sasaran; jarak tambahan hanya menurunkan presisi dan menambah waktu.

Pada Percobaan 2 (input “monstr_bru”, sepuluh karakter, dua kesalahan), ambang dinamis mulai dari k=3 lalu naik satu kali–retry sehingga berhenti pada k=4 dan mengembalikan satu saran *monster_baru* (jarak 2) dalam 0,038 s. Ambang konstan tetap longgar dan memunculkan lima saran (jarak 2–4) dalam 0,045 s. Di sini recall sama-sama 100 %, tetapi dinamis mempertahankan daftar ringkas—meningkatkan presisi sekaligus memangkas ~15 % waktu.

Pada Percobaan 3 (input “display_monstsr_invntoty”, 25 karakter, multi-edit), ambang dinamis menghitung k=4 (ceil 0,25·25) lalu melakukan beberapa *retry* sampai k=5; hasil akhirnya satu kandidat tepat (*display_monster_inventory*, jarak 3) dalam 0,092 s. Ambang konstan, dengan batas 10, memberi tambahan *display_inventory* (jarak 10) dan butuh 0,097 s. Kasus ini mengonfirmasi bahwa semakin panjang token, semakin besar potensi *false positive* pada ambang konstan, sedangkan ambang dinamis tetap membatasi keluaran tanpa kehilangan target.

Secara keseluruhan dapat disimpulkan bahwa:

1. Presisi lebih tinggi

   Ambang dinamis konsisten hanya mengusulkan kata dengan jarak terdekat (1–3), sedangkan ambang konstan menyertakan kandidat jarak jauh ketika kkk besar.

2. Latensi lebih rendah

   Pemangkasan kandidat sejak awal membuat fungsi dinamis 15–40 % lebih cepat (0,020–0,092 s vs 0,032–0,097 s) walau masih di bawah batas kenyamanan interaktif (< 0,1 s).

3. Skalabilitas adaptif

   Karena $k \propto$ panjang token dan dibatasi *ceiling*, biaya komputasi tetap terkendali meski token bertambah panjang; pendekatan konstan harus mengorbankan presisi atau menulis ulang k.

4. Robust terhadap variasi kesalahan

   Mekanisme *retry* bertingkat pada ambang dinamis mencegah kegagalan total (daftar kosong) ketika jumlah kesalahan melebihi prediksi awal.

Dengan mempertimbangkan empat poin di atas, ambang dinamis—dengan rumus $k=\lceil \alpha L \rceil$ batas $[\,\text{floor},\text{ceiling}\,]$—menjadi pilihan yang lebih seimbang untuk sistem *autocomplete*: ia menjaga daftar saran tetap singkat namun andal, dan memberikan kinerja yang konsisten pada token pendek maupun panjang tanpa penalaan manual untuk setiap skenario.

VII. KESIMPULAN

Implementasi fitur *autocorrection* berbasis *dynamic threshold* memberikan keseimbangan optimal antara keakuratan dan kinerja. Penyesuaian ambang secara proporsional terhadap panjang token, disertai batas bawah–atas serta skema *retry*, memungkinkan sistem menghasilkan daftar saran ringkas dan relevan pada berbagai skenario kesalahan ketik. Dibandingkan pendekatan ambang konstan, metode ini terbukti lebih presisi, lebih cepat, dan lebih adaptif terhadap variasi jumlah kesalahan maupun panjang nama simbol. Dengan demikian, formula $k=\lceil \alpha L \rceil$—dengan parameter α (alpha), *floor*, dan *ceiling* minimal—dapat direkomendasikan sebagai standar praktis untuk modul *autocorrection* di lingkungan pengembangan perangkat lunak, tanpa memerlukan proses pelatihan atau penyetelan parameter yang kompleks.

VIDEO LINK YOUTUBE

https://youtu.be/UHMemFq7Gy8

UCAPAN TERIMA KASIH

Penulis memanjatkan puji syukur ke hadirat Tuhan Yang Maha Esa atas limpahan rahmat dan karunia-Nya sehingga makalah berjudul “Implementasi Fitur Code Autocorrection Sederhana Menggunakan Algoritma Damerau-Levenshtein untuk Saran Nama Variabel dan Fungsi” dapat diselesaikan tepat waktu. Penulis menyampaikan terima kasih yang sebesar-besarnya kepada orang tua serta rekan-rekan yang senantiasa memberikan dukungan moral dan semangat selama

proses penulisan makalah ini. Penghargaan setinggi-tingginya juga penulis sampaikan kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. dan Dr. Ir. Rinaldi Munir, M.T. selaku dosen pengajar mata kuliah Strategi Algoritma yang telah membagikan ilmu, arahan, dan motivasi selama kegiatan belajar-mengajar berlangsung. Akhir kata, penulis berterima kasih kepada semua pihak lain yang tidak dapat disebutkan satu per satu namun turut membantu terselesaikannya makalah ini.

LAMPIRAN

Implementasi kode lengkap program dapat diakses pada link berikut :
https://github.com/ryonlunar/simple-autocorrection

REFERENSI

[1] R. Munir, "Pencocokan String", diakses Jun. 20 2025 [Online]. Tersedia :https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2024-2025/23-Pencocokan-string-(2025).pdf

[2] GeeksforGeeks, "Damerau-Levenshtein Distance", diakses Jun. 20 2025 [Online].Tersedia:https://www.geeksforgeeks.org/dsa/damerau-levenshtein-distance/

[3] Y. Gaddam, "Wagner–Fischer algorithm: Minimum Edit Distance", diakses Jun. 20 2025 [Online]. Tersedia:https://medium.com/%40yasaswini.gaddam21/wagner-fischer-algorithm-minimum-edit-distance-4e61bba9b656

[4] P. Brans, "fuzzy search", diakses Jun 20 2025 [Online], Tersedia: https://www.techtarget.com/whatis/definition/fuzzy-search

[5] GeeksforGeeks, "Edit Distancee", diakses Jun. 20 2025 [Online].Tersedia:https://www.geeksforgeeks.org/dsa/edit-distance-dp-5/

[6] GeeksforGeeks, "Introduction to Levenshtein distance", diakses Jun. 20 2025 [Online].Tersedia:https://www.geeksforgeeks.org/dsa/introduction-to-levenshtein-distance/

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, 23 Juni 2025

Adhimas Aryo Bimo dan 13523052