Aplikasi Algoritma Dijkstra dalam Membetulkan Kesalahan Tipografi Menggunakan JavaScript

Michael Jonathan Halim - 13521124[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
[1]13521124@std.stei.itb.ac.id

***Abstrak*—Kesalahan tipografi merupakan sebuah kesalahan yang terjadi akibat slip tangan saat proses mengetik. Kesalahan dalam mengetik bisa menghasilkan kata-kata yang ambigu bahkan arti yang salah. Pada makalah ini, akan dibuat suatu program yang dapat menentukan apakah sebuah kata *typo* atau tidak dan membetulkan sebuah kata jika *typo* dengan memanfaatkan graf berarah dan berbobot. Dengan memanfaatkan algoritma dijkstra, program dapat mencari kata yang serupa dengan kata yang *typo* dengan menghitung dan mencari jarak terkecil dari setiap kata terhadap kata yang *typo* sehingga didapatkan kata yang diinginkan. Oleh karena itu, penggunaan algoritma dijkstra dapat membantu membetulkan kesalahan tipografi secara efektif.**

***Keywords*—Typo, Graf, Dijkstra.**

I. PENDAHULUAN

Teknologi zaman sekarang sudah sangat canggih. Teknologi yang semakin berkembang tentu juga membantu menyelesaikan permasalahan lebih cepat. Contohnya dapat kita temukan dalam sehari-hari seperti berkomunikasi dengan orang lain dalam jarak jauh, mencari hal-hal yang ada di seputar dunia dengan cepat, bahkan membuat program juga sudah bisa lebih cepat dibanding zaman dahulu karena banyak sekali *library build-in* dan *extension* yang sudah dibuat untuk mempermudah programmer sekarang dan kedepannya.

Sebagai pengguna internet, tentu kita sering sekali menggunakan kakas seperti *keyboard* untuk berinteraksi dengan orang lain ataupun teknologi itu sendiri. Namun, hal yang masih dipermasalahkan hingga sekarang adalah kemampuan mengetik seseorang berbeda-beda karena dipengaruhi faktor-faktor seperti ukuran tangan, bentuk *keyboard* yang dipakai, bahkan juga usia.

Akibat dari kekurangmampuan mengetik seseorang adalah sering terjadinya kesalahan tipografi dalam menulis sesuatu, seperti untuk tugas, menulis cerita, dan *coding*. Tentu akibat dari kesalahan tipografi adalah keambiguan bahasa yang disampaikan. Contohnya jika kita ingin mengirimkan pesan "ketika" kepada teman kita, namun kita salah mengetik dan yang terkirim adalah "ketiak", teman kita akan menangkap maksud yang berbeda dari yang seharusnya.

Hal ini juga bisa dikaitkan dengan teknologi seperti search engine. Dalam *search engine*, tentu kita ingin mencari sesuatu yang kita ingin ketahui atau kita butuhkan. Namun, apabila kita salah mengetik, maka *search engine* juga akan bingung dengan apa yang sebenarnya ingin kita cari.

Kesalahan tipografi sendiri adalah kesalahan yang dibuat pada saat proses mengetik. Kesalahan tipografi terjadi ketika kita menekan tombol yang berbeda dari yang seharusnya namun letak dari tombol itu berdekatan dengan tombol yang seharusnya ditekan. Kesalahan tipografi juga bisa terjadi karena kita menekan dua tombol secara langsung karena ukuran dari tombol *keyboard* yang kecil sehingga jari kita tidak secara sengaja menekan dua tombol berdekatan.

Oleh karena itu, diperlukan sebuah algoritma untuk menyelesaikan permasalahan kesalahan tipografi. Salah satu algoritma yang bisa digunakan untuk membuat aplikasi pembetulan kesalahan tipografi adalah algoritma dijkstra yang memanfaat struktur data graf yaitu sekumpulan objek terstrukur yang di antaranya memiliki hubungan atau keterkaitan tertentu.

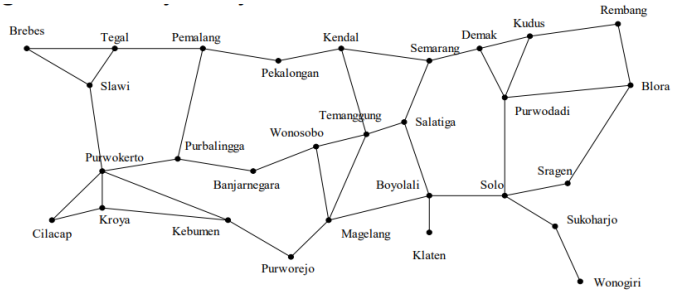
Gambar 1.1 Contoh penggunaan keyboard yang tidak sesuai dengan ukuran jari tangan
Sumber : www.reviewpro.com/tr/man-sending-text-message-and-sms-with-smartphone-guy-texting-and-using-mobile-phone-late-at-night-in-dark-communication-or-sexting-concept-finger-typing-with-cellphone-keyboard-light-from-screen-2/

Di makalah ini, penulis akan membahas bagaimana sebuah aplikasi dapat menangani kasus kesalahan tipografi oleh pengguna dengan memanfaatkan struktur data graf yang merepresentasikan *layout keyboard* dan meminimalisir kesalahan pengetikan dengan mencari lintasan terpendek dari graf *layout keyboard*. Algoritma yang akan dipakai pada aplikasi ini adalah algoritma Dijkstra dengan graf berarah yang berbobot.

II. LANDASAN TEORI

A. Graf

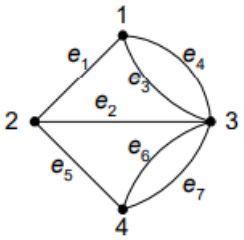
Graf merupakan suatu struktur data yang digunakan untuk merepresentasikan objek-objek diskrit dan hubungan antara objek-objek tersebut.



Gambar 2.1 Contoh graf yang merepresentasikan peta jaringan jalan raya yang menghubungkan kota-kota di Provinsi Jawa Tengah
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

Sebuah graf terdiri dari 2 komponen, yaitu simpul (*vertex*) dan sisi (*edge*). Simpul atau node dapat direpresentasikan dengan objek titik, lingkaran, kotak, dan bentuk lainnya. Simpul harus diberikan penamaan agar tiap simpul menjadi unik. Sisi adalah garis-garis yang menghubungkan simpul-simpul dari graf. Dengan kata lain, graf dapat ditulis sebagai G = (V, E), yang dalam hal ini :

1. V = himpunan tidak kosong dari simpul-simpul = $\{v_1,v_2,..,v_n\}$
2. E = himpunan sisi = $\{e_1,e_2,..,e_n\}$.



Gambar 2.2 Sebuah graf yang memiliki representasi angka untuk simpul dan representasi e untuk sisi
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

Dari gambar di atas, bisa diketahui bahwa e_1 hingga e_7 adalah sisi-sisi dari graf dan 1 hingga 4 adalah simpul-simpul dari graf.

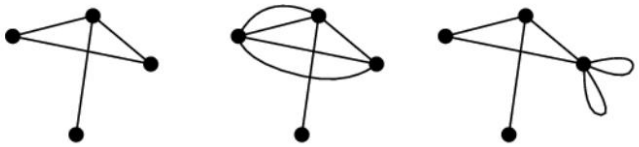
B. Jenis-Jenis Graf

Berdasarkan ada tidaknya gelang atau sisi ganda pada suatu graf, suatu graf dapat digolong menjadi dua jenis :

1. Graf Sederhana (*simple graph*)
   Graf sederhana merupakan graf yang tidak memiliki gelang ataupun sisi ganda.
2. Graf Tak-Sederhana (*unsimple graph*)
   Graf tak-sederhana merupakan graf yang memiliki sisi ganda atau sisi gelang.

Graf tak-sederhana juga dapat dibedakan menjadi dua jenis lagi, yaitu

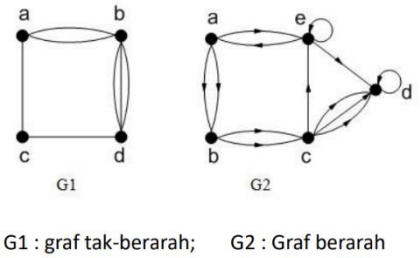
1. Graf Ganda (*multi graph*)
   Graf ganda merupakan graf yang mengandung sisi ganda.
2. Graf Semu (*pseudo graph*)
   Graf semu merupakan graf yang mengandung sisi gelang.



Gambar 2.3 Contoh graf sederhana, graf tak-sederhana dengan sisi ganda (graf ganda), dan graf tak-sederhana dengan sisi gelang (graf semu).
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

Berdasarkan orientasi arah pada sisi, graf dapat digolongkan menjadi dua jenis juga :

1. Graf tak-berarah (*undirected graph*)
   Graf tak berarah merupakan graf yang sisinya tidak mempunyai orientasi arah.
2. Graf berarah (directed graph)
   Graf berarah merupakan graf yang setiap sisinya diberikan orientasi arah.



Gambar 2.4 Contoh graf tak-berarah (G_1) dan graf berarah (G_2)
Sumber :
https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf

C. Terminologi Graf

Dalam graf sendiri terdapat beberapa terminologi, yakni

1. Ketetanggaan (*Adjacent*)
   Dua buah simpul dikatakan bertetangga jika keduanya terhubung langsung oleh suatu sisi.

2. Bersisian (*Incidency*)
   Sebuah sisi dikatakan bersisian dengan dua simpul acak jika sisi itu menghubungkan dua simpul tersebut.

3. Simpul Terpencil (*Isolated Vertex*)
   Simpul terpencil merupakan simpul yang tidak terhubung sama sekali dengan sisi lain.

4. Graf Kosong (*Null Graph*)
   Sebuah graf dikatakan kosong jika graf tersebut memiliki himpunan sisi yang kosong.

5. Derajat (*Degree*)
   Derajat suatu simpul merupakan jumlah sisi yang bersisian dengan simpul tersebut.

6. Lintasan (*Path*)
   Lintasan pada graf merupakan suatu lintasan yang memiliki simpul awal dan simpul akhir, juga terdapat sisi-sisi dan simpul-simpul yang berhubungan dengan kedua simpul tersebut sehingga terbentuk lintasan dari simpul awal ke simpul akhir.

7. Siklus (*Cycle*)
   Siklus adalah lintasan yang berawal dan berakhir pada simpul yang sama.

8. Kerterhubungan (*Connected*)
   Dua simpul dikatakan terhubung jika terdapat lintasan dari simpul pertama ke simpul kedua. Sebuah graf dikatakan graf terhubung jika untuk setiap pasang simpul terdapat lintasan antara pasangan tersebut.

9. Upagraf (*Subgraph*)
   Sebuah graf G memiliki upagraf G_1 jika himpunan simpul dan himpunan sisinya merupakan himpunan bagian dari himpunan simpul dan himpunan sisi G.

10. Upagraf Merentang (*Spanning Subgraph*)
    Sebuah upagraf dikatakan merentang jika upagraf tersebut mengandung semua simpul dari graf utamanya.

11. *Cut-Set*
    Cut-set dari graf terhubung G adalah himpunan sisi yang bila dibuang dari G menyebabkan graf G tersebut tidak terhubung.

12. Graf Berbobot
    Graf berbobot adalah graf yang setiap sisinya diberi sebuah harga. Makalah ini akan fokus pada pembahasan yang berkaitan dengan graf yang berbobot dan berarah.

D. Beberapa Graf Khusus

Dalam graf juga terdapat beberapa graf khusus, seperti

1. Graf Lengkap
   Graf lengkap adalah graf sederhana yang setiap simpulnya memiliki sisi ke semua simpul lainnya.

2. Graf Lingkaran
   Graf lingkaran adalah graf sederhana yang setiap simpulnya berderajat dua.

3. Graf Teratur
   Graf teratur adalah graf yang setiap simpulnya memiliki derajat yang sama.

4. Graf Bipartite
   Graf bipartite adalah graf yang himpunan simpulnya dapat dipisah menjadi dua bagian sehingga setiap sisinya menghubungkan sebuah simpul pada himpunan terpisah pertama ke himpunan terpisah kedua.

E. Kesalahan Tipografi

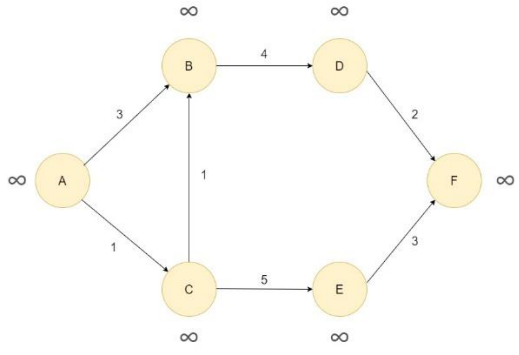
Kesalahan tipografi (atau biasa dikenal dengan istilah *typo*) adalah kesalahan yang dibuat pada saat seseorang mengetik. Hal ini biasanya terjadi karena slip tangan atau jari saat proses mengetik tetapi bukan termasuk kesalahan yang timbul akibat ketidaktahuan penulis. Beberapa kesalahan ketik bisa mudah dikenali seperti kata ‘hujan’ menjadi ‘hjuan’.

F. Algoritma Dijkstra

Algoritma Dijkstra merupakan suatu algoritma yang digunakan untuk memecahkan permasalahan dalam mencari jarak terpendek (*shortest path problem*) dari suatu *node* sebagai *start node*, ke *node* lain sebagai *finish node* dalam graf berarah dan berbobot. Algoritma ini sering digunakan dalam permasalahan kehidupan sehari-hari seperti alat GPS untuk mencari rute terpendek dari suatu lokasi ke lokasi lain.

Untuk algoritmanya sendiri, terdapat beberapa langkah yang dapat dilakukan sebagai berikut:

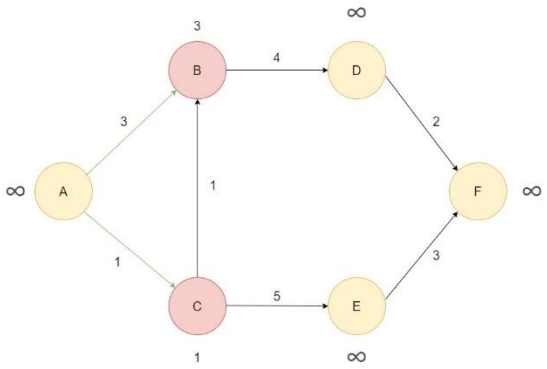
1. Setiap *node* memiliki bobot masing-masing dan di-*assign* dengan nilai *infinity* sebagai permulaan.



Gambar 2.5 Ilustasi Graf Bearah dan Berbobot dengan

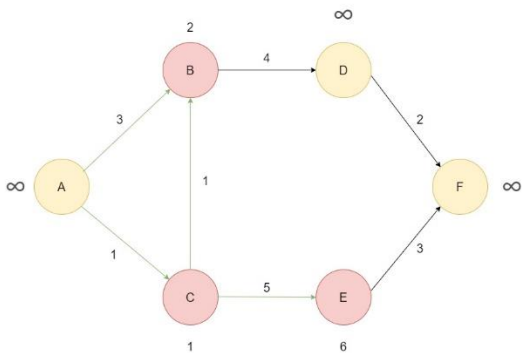
setiap *node* di-*assign* bobot sebesar *infinity*

2. Memulai proses penyimpanan bobot (biaya untuk menuju *node* tersebut) pada *node-node* tetangga dari *start node*. Contoh, dari *start node* ke *node* B memerlukan biaya sebesar tiga, maka B sekarang memiliki bobot sebesar tiga.



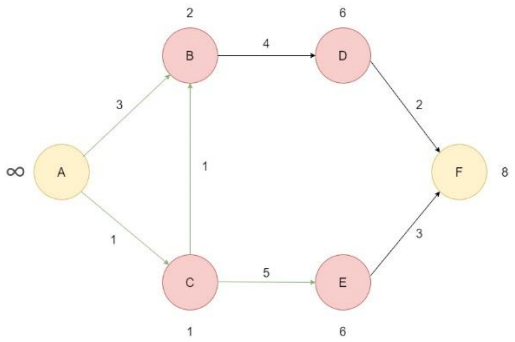
Gambar 2.6 Ilustasi langkah 2 Algoritma Dijkstra

3. Mencari *node* dengan bobot terendah yang belum diproses. Arti dari proses disini adalah melakukan perhitungan untuk biaya ke *node-node* tetangganya. Contoh, selesai *start node* menghitung biaya ke *node* B dan C, maka *node-node* yang belum diproses adalah B dan C.
4. Menyimpan bobot dari *node* sekarang (*node* dengan bobot terendah untuk sekarang).
5. Melakukan iterasi untuk setiap tetangga dari *node* sekarang, hitung bobot yang diperlukan untuk mencapai masing-masing tetangga dari *node* sekarang dengan perhitungan bobot *node* sekarang ditambah dengan biaya untuk ke *node* tetangga tersebut. Simpan hitungan tersebut untuk bobot *node-node* tetangga jika nilainya lebih kecil dari bobot *node* tetangga tersebut atau belum ada.



Gambar 2.7 Ilustasi langkah 5 Algoritma Dijkstra

6. Jika sudah selesai proses perhitungan, simpan *node* tersebut ke dalam list *node-node* yang sudah diproses.
7. Lakukan langkah 3-6 hingga semua *node* termasuk *final node* terproses.



Gambar 2.8 Ilustasi Algoritma Dijkstra Selesai Dilakukan

8. Bobot yang disimpan pada *final node* adalah biaya terkecil untuk dicapai dari *start node*.

III. Graf Keyboard Qwerty

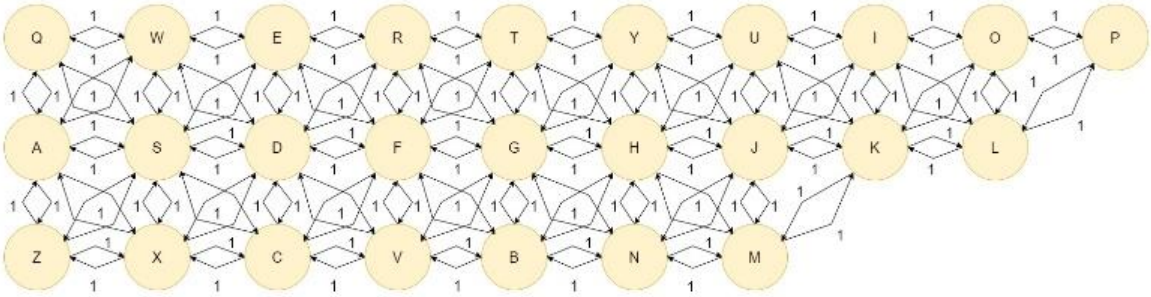
Keyboard pada umumnya memiliki tata letak yang bernama QWERTY. Tata letak QWERTY ini pertama kali diperkenalkan oleh Christopher Latham Sholes pada tahun 1870. Alfabet-alfabet dalam *keyboard* diatur menjadi tiga baris. Baris pertama mengandung sepuluh huruf secara berurutan yaitu ‘q’, ‘w’, ‘e’, ‘r’, ‘t’, ‘y’, ‘u’, ‘i', ‘o’, dan ‘p’. Baris kedua mengandung sembilan huruf ‘a’, ‘s’, ‘d’, ‘f’, ‘g’, ‘h’, ‘j’, ‘k’, dan ‘l’. Baris ketiga mengandung tujuh huruf yaitu ‘z’, ‘x’, ‘c’, ‘v’, ‘b’, ‘n’, dan ‘m’. Pemodelan tata letak *keyboard* ini tentu bisa kita buat menjadi sebuah graf. Graf yang dibuat pada makalah ini bertujuan untuk membantu pembuatan program pembetulan kesalahan tipografi.

IV. Analisis Permasalahan

A. Penggunaan Graf dalam Pembetulan Kesalahan Tipografi

Graf berarah dan berbobot dari tata letak *keyboard* QWERTY dapat membantu kita dalam memperbaiki kesalahan tipografi. Telah kita ketahui bahwa salah satu penyebab kesalahan tipografi karena slip tangan atau jari sehingga huruf yang diketik tidak sesuai dengan yang diinginkan. Huruf yang diketik namun tidak sesuai akibat slip tangan pasti berdekatan dengan huruf yang seharusnya. Oleh karena itu, graf disini dibuat berbobot untuk menyatakan jarak dari suatu huruf ke huruf lainnya. Agar graf tidak rumit, dibuat sisi-sisi untuk huruf-huruf yang saling berdekatan dengan bobot satu. Namun, permasalahan kita sekarang adalah bagaimana kita bisa mengetahui jarak dari suatu huruf ke huruf lainnya.

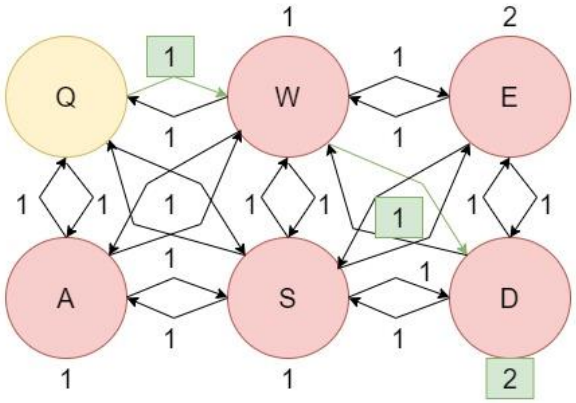
Jarak antara suatu huruf dengan huruf lainnya pada *keyboard* adalah jarak terpendek dari graf berarah dan berbobot yang telah direpresentasikan pada makalah ini. Maka, diperlukan suatu algoritma yang dapat mencari jarak terpendek dari suatu *node* huruf ke *node* huruf lainnya yaitu algoritma Dijkstra.



Gambar 3.1 Graf Keyboard QWERTY

B. Aplikasi Dijkstra dalam Pembetulan Kesalahan Tipografi

Seperti yang sudah dibahas dalam landasan teori, langkah-langkah yang dilakukan pada graf *keyboard* QWERTY ini akan sama seperti yang sudah dijelaskan. Contoh, jika kita ingin mencari jarak terpendek dari huruf Q ke D, dengan algoritma Dijkstra, kita akan dapatkan hasil sebagai berikut.



Gambar 4.1 Ilustrasi Aplikasi Dijkstra pada Pencarian Jarak Terpendek dari Huruf Q ke D

Dari gambar di atas, bisa didapatkan bahwa jarak terpendeknya adalah dua.

C. Algoritma Utama dalam Pembetulan Kesalahan Tipografi

Untuk algoritma utamanya sendiri, tentu kita perlu mempersiapkan kamus dari bahasa yang akan kita koreksi. Pembetulan kesalahan tipografi bertujuan untuk mengoreksi bahasa yang tidak tepat menjadi bahasa yang seharusnya, maka dari itu kita juga harus mendefinisikan bahasa yang seharusnya itu seperti apa.

Setelah kita mendefinisikan kamus dari bahasa yang ingin kita koreksi, terdapat beberapa langkah untuk memeriksa suatu bahasa sebagai berikut:

1. Memeriksa apakah bahasa itu termasuk dalam kamus yang sudah kita definisikan atau tidak. Jika iya, kita tidak perlu membetulkan kesalahan karena bahasa tersebut sudah benar. Jika tidak, maka kita perlu memeriksa lebih lanjut dan simpan kata tersebut untuk membentuk kalimat baru.
2. Iterasi semua kata yang sudah kita definisikan dalam kamus kita. Jika terdapat kata dengan panjang yang sama, maka akan kita cari *distance* dari kata tersebut dengan kata yang ingin diperbaiki.
3. *Distance* suatu kata dapat dihitung dengan total jarak huruf-huruf yang terdapat pada kata pada kamus dengan huruf-huruf yang terdapat pada kata yang ingin diperbaiki sesuai dengan urutannya. Contoh, untuk kata ‘hujan’ dengan ‘hjuan’ akan menghasilkan distance sebesar dua karena huruf u dengan j (pada urutan huruf kedua) memiliki jarak yaitu satu dan huruf j dengan u (pada urutan huruf ketiga) memiliki jarak yaitu satu juga sehingga didapatkan total jarak sebesar dua.
4. Bandingkan *distance* yang sudah didapatkan dengan *distance* terendah yang sudah disimpan. Jika belum ada *distance* terendah, maka simpanlah *distance* tersebut sebagai *distance* terendah dan simpan juga kata yang memiliki *distance* terendah tersebut.
5. Jika semua kata dalam kamus sudah diiterate, simpanlah kata dengan *distance* terendah untuk membentuk kalimat baru.
6. Lakukan langkah 1-5 untuk semua kata yang ada pada kalimat.
7. Sesudah semua kata dalam kalimat diproses, gabungkan kata-kata tersebut.

V. Aplikasi Website Perbaikan Typo Dalam Search Engine Menggunakan Javascript

Implementasi pembetulan kesalahan tipografi bisa digunakan untuk aplikasi-aplikasi yang memerlukan input dari pengguna dan perlu pemeriksaan apakah input sesuai yang diinginkan pengguna atau tidak. Contoh dari salah satu aplikasi yang terkenal adalah *search engine*. Pada *search engine*, pengguna akan menginput apa yang ingin dicari olehnya, setelah itu dilakukan *query* pada *database* untuk kalimat yang sudah diinput. Namun, apabila terjadi kesalahan pada input, tentu program juga akan bingung dan bisa saja menghasilkan *query* yang tidak tepat.

Untuk pembuatan aplikasi *website* perbaikan typo ini, seperti yang sudah dibahas pada sebelumnya, kita perlu mempersiapkan kamus untuk bahasa yang ingin dikoreksi. Untuk contoh program pada makalah ini, disiapkan kamus bahasa untuk bahasa Indonesia sesuai KBBI. Simpan kamus bahasa tersebut dalam sebuah *file* .txt.

Aplikasi ini dibuat dengan bahasa pemrograman *Javascript* yang bertujuan sebagai *back-end* dari *website* yang akan kita buat. Pertama, kita perlu men-*setup* data dari yang sudah kita

siapkan pada *file* .txt tadi yaitu kamus kita. Kita memerlukan sebuah fungsi untuk membaca *file* .txt tersebut sehingga bisa kita pakai dalam algoritma utama kita. Kita gunakan fungsi *fetch* untuk membaca dan menerjemahkan file kamusnya dan kita gunakan fungsi *split* untuk memisahkan tiap kata yang ada di dalam *file* dan kita simpan hasilnya dalam sebuah *array*. Berikut adalah kode untuk membaca *file* .txt kamus bahasa.

```
// Fungsi untuk membaca file txt kamus bahasa Indonesia
  const syncReadFile = async () => {
    const contents = await fetch("indonesian-words.txt")
      .then((response) => response.text())
      .then((text) => {
        return text;
      });

    const arr = contents.split(/\r?\n/);
    let temp = [];
    for (let item of arr) {
      let found = false;
      for (let char of item) {
        if (char === "-") {
          found = true;
        }
      }
      if (found) {
        continue;
      } else {
        temp.push(item);
      }
    }

    return temp;
  };
```

Setelah kita mendefinisikan fungsi baca *file*, kita juga perlu untuk men-setup *dictionary* dari setiap kata dengan kata tersebut sebagai *key* and *distance* sebagai *value*. *Distance* di-*set* nol untuk semua kata sebagai permulaan sebelum digunakan. Berikut adalah kode untuk setup *array words* dan *dictionary* untuk setiap kata dalam kamus.

```
// Menyimpan kata-kata bahasa Indonesia dalam array
let words = await syncReadFile("./indonesian-words.txt");

// Object untuk inisiasi distance tiap kata
let dictionary = {};
words.forEach((item) => (dictionary[item] = 0));
```

Langkah selanjutnya setelah kita setup kamus adalah membuat graf dari *keyboard* QWERTY dalam *javascript*. Struktur data graf dalam program ini akan menggunakan *object* dengan suatu *node* sebagai *key* dan sebuah *object* baru yang berisi *node-node* tetangganya beserta bobot untuk menuju tetangganya sebagai *value* dari *node* tersebut. Salah satu contoh graf untuk *node* Q adalah { Q : { W : 1, S : 1, A : 1 } }. Namun, sebelum membuat graf utama, kita buat graf sementara dengan struktur data yang berbeda. Tujuan dari graf sementara ini adalah agar pembuatan graf utama menjadi lebih gampang dan programmer lain menjadi lebih mudah untuk memahami struktur dari graf tersebut. Berikut adalah kode sekilas untuk setup graf *temporary* untuk beberapa huruf dalam *keyboard*.

```
// Graph temporary untuk layout keyboard QWERTY
  let graph_temporary_qwerty = {
    Q: [
      ["Q", "W", 1],
      ["Q", "S", 1],
      ["Q", "A", 1],
    ],
    W: [
      ["W", "Q", 1],
      ["W", "A", 1],
      ["W", "S", 1],
      ["W", "E", 1],
      ["W", "D", 1],
    ],
    E: [
      ["E", "W", 1],
      ["E", "R", 1],
      ["E", "F", 1],
      ["E", "D", 1],
      ["E", "S", 1],
    ],
    …
  }
```

Setelah kita selesai men-*setup* graf sementara, kita lanjutkan dengan membuat graf utama dari *keyboard*. Kita akan mengiterasi setiap *key* pada graf sementara kita lalu kita tambahkan informasi *node* tetangga dari huruf tersebut beserta bobotnya. Berikut adalah kode untuk membuat graf utama *keyboard* QWERTY.

```
// Membuat graph dari layout keyboard QWERTY
    const graph = {};
    Object.keys(graph_temporary_qwerty).forEach((item) =>
(graph[item] = {}));

    for (let item of Object.keys(graph_temporary_qwerty)) {
      for (let edge of graph_temporary_qwerty[item]) {
        graph[edge[0]][edge[1]] = edge[2];
      }
    }
```

Langkah selanjutnya setelah kita membuat graf *keyboard* adalah mengimplementasikan algoritma Dijkstra pada program. Sesuai dengan yang sudah dibahas pada landasan teori, untuk membuat algoritma Dijkstra, salah satu langkah yang perlu dilakukan adalah mencari *node* dengan bobot terendah yang belum diproses. Node dengan kriteria tersebut dapat dicari dengan mengiterasi seluruh node yang sudah menyimpan

sebuah bobot. Kita gunakan struktur data *hash table* untuk menyimpan bobot dari setiap *node*. Kita buat fungsi yang dapat mengiterasi *hash table* tersebut dan mengembalikan *node* dengan bobot (biaya untuk menuju *node* tersebut dari *start node*) terendah. Berikut adalah kode untuk mencari *node* dengan bobot terendah yang belum diproses.

```
// Mencari node dengan bobot terendah yang belum diproses
const minimumCostNode = (costs, processed) => {
  let minimum = null;
  // Iterasi setiap node pada object costs
  for (let node of Object.keys(costs)) {
    // Mencari node dengan bobot terendah
    if (minimum === null || costs[node] < costs[minimum]) {
      if (!processed.includes(node)) {
        minimum = node;
      }
    }
  }
  return minimum;
};
```

Untuk algoritma dijkstra, sesuai dengan penjelasan pada landasan teori, kita buat sebuah fungsi bernama dijkstra untuk menghitung *distance* terdekat dari sebuah huruf ke huruf lainnya pada *keyboard*. Fungsi ini akan menerima 3 parameter yaitu graf, huruf pertama, dan huruf kedua. Untuk grafnya sendiri kita buat variabel lokal baru agar kita tidak mengubah graf utama kita. Setelah itu, karena huruf B adalah tujuan dari huruf A, maka kita *remove* sisi-sisi dari *node* huruf B ke huruf lain. Lalu, kita inisiasi *object costs* yaitu bobot untuk setiap node dalam graf dan array *processed* untuk menyimpan *node-node* yang sudah diproses. Berikut adalah kode untuk fungsi algoritma dijkstra.

```
const dijkstra = (graph, A, B) => {
  let temp_graph = { ...graph };
  delete temp_graph[B];
  temp_graph[B] = {};

  // Inisiasi Object Cost Minimum Ke Setiap Node
  const costs = Object.assign({ [`${B}`]: Infinity }, temp_graph[`${A}`]);

  // Inisiasi Array Untuk Node Yang Sudah Diproses
  // Sudah diproses artinya kita sudah pernah menghitung bobot untuk menjangkaunya dari starting node
  const processed = [];

  // Menentukan node dengan bobot terendah yang belum diproses
  let node = minimumCostNode(costs, processed);

  // Looping untuk terus mencari node dengan bobot terendah
  while (node) {
    // Menyimpan bobot node sekarang
    let cost = costs[node];

    // Menyimpan node tetangga dari node sekarang
    let children = temp_graph[node];

    // Looping untuk setiap node tetangga dan menghitung bobot baru untuk menjangkau node tetangga tersebut
    for (let child in children) {
      let newCost = cost + children[child];
      if (!costs[child]) {
        costs[child] = newCost;
      }

      // Jika bobot baru lebih kecil dari bobot semula
      if (newCost < costs[child]) {
        costs[child] = newCost;
      }
    }

    // Memasukkan node ke list node yang sudah diproses
    processed.push(node);

    // Mencari kembali node dengan bobot terendah yang belum diproses
    node = minimumCostNode(costs, processed);
  }

  // Mengembalikan bobot node tujuan
  return costs[`${B}`];
};
```

Langkah berikutnya adalah membuat kode HTML untuk *front-end website*-nya. Untuk tampilan *website*-nya cukup sederhana saja, tampilan mengandung judul dari program, *input field* untuk teks, *button* untuk melakukan pemeriksaan *query*, dan teks paragraf untuk menampilkan koreksi dari *query*. Berikut adalah kode untuk *front-end* dari *website*.

```
<!DOCTYPE html>
<body style="background-color: rgb(51, 50, 50); font-family:'Gill Sans MT'; ">
    <div style="display:flex; flex-direction: column; justify-content: center; align-items: center; height: 60vh">
        <h1 style="font-size: 50px; color: white;">MICHAEL'S SEARCH</h1>
        <div style="width:30%; display: flex; gap: 8px;">
            <input type="text" name="input" id="input" class="input" style="width: 100%; padding: 8px; font-size: 16px;" placeholder="Apa yang ingin Anda cari hari ini?"/>
            <button id="click-me">Search</button>
        </div>
        <p id="checkQuery" style="font-size: 20px; color: white;"></p>
    </div>
    <script type="module" src="main.js"></script>
  </body>
</html>
```

Kita akan hubungkan *button* pada *front-end* dengan *back-end* algoritma utama kita, maka kita inisiasi variabel untuk *event button* tersebut.

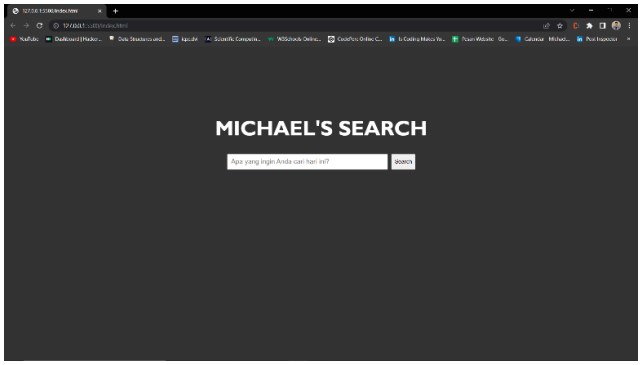
```
// Mendapatkan element button
var clickMe = document.getElementById("click-me");
```

Algoritma utama pada program ini terletak pada *event onclick* pada *button* tadi. Kita inisiasi terlebih dahulu *array* untuk menyimpan kata-kata yang sudah diperiksa dan diperbaiki jika terdapat *typo*. Lalu, kita dapatkan *value* dari *input* pengguna dan kita gunakan fungsi *split* untuk memecah kalimat *query* dari pengguna menjadi kata per kata. Setelah itu, kita mulai *looping* untuk setiap kata yang sudah kita pisah dari input pengguna. Jika kata tersebut terdapat pada kamus yang sudah kita definisikan, maka kita tidak perlu memperbaikinya. Jika kata tersebut tidak terdapat pada kamus yang sudah kita definisikan, kita periksa apakah terdapat kata pada kamus yang memiliki panjang yang sama. Jika terdapat kata dengan panjang yang sama, kita hitung *distance* dari kata tersebut dan kita periksa apakah *distance* tersebut merupakan *distance* terendah untuk sekarang atau tidak. Jika *distance* tersebut adalah *distance* terendah, kita simpan kata tersebut dan *distance*-nya. Kita lakukan untuk semua kata dalam kamus hingga kita temukan kata dengan *distance* terendah. Lalu, kita masukkan kata yang sudah diproses dalam *array* untuk kata-kata baru. Setelah semua kata dalam *input* sudah diproses, kita gabungkan kembali kata-kata tersebut dan kita tampilkan pada *front-end website*. Apabila tidak terdapat perubahan, maka *website* akan menampilkan kalimat bahwa tidak terdapat *typo*. Jika terdapat perubahan, *website* akan menampilkan perbaikan dari kalimat input pengguna. Berikut adalah kode untuk algoritma utama dari *back-end website*.

```
  // Add event onclick untuk button click-me
  clickMe.onclick = function () {
  // Kumpulan kata yang sudah diperbaiki
    let newArrayWords = [];
    let change = false; // Jika terjadi perubahan
  // Mendapatkan input dari value
    let input = document.getElementById("input").value;
  // Memisahkan tiap kata
    let array_of_string = input.split(" ");

    // Looping untuk proses tiap kata
    for (let val of array_of_string) {
  // Definisi variabel bebas untuk batas distance typo
      let minim = 10;
      let ans; // Kata baru

      let found = false; // Jika terjadi perubahan
  // Mengubah semua huruf menjadi lowercase
      val = val.toLowerCase();

      if (words.includes(val)) {
        // Jika kata tersebut terdapat pada kamus bahasa
Indonesia dari txt
        ans = "Tidak Ada Yang Typo";
      } else {
        // Menjadikan kata jadi uppercase
        val = val.toUpperCase();

        // Inisiasi distance tiap kata dalam kamus
        for (let string of words) {
          let distance = 0;
          dictionary[string] = distance;
        }

        // Menghitung distance dari setiap kata
        for (let item of Object.keys(dictionary)) {
          let dist = 0;
          let idx = 0;

        // Menjadikan kata jadi uppercase
          item = item.toUpperCase();
          if (item.length == val.length) {
   // Typo adalah ketika panjang kata sama namun terdapat
perbedaan huruf
            for (let char of item) {
              if (char === val[idx]) {
                dist += 0;
              } else {
                dist += dijkstra(graph, char, val[idx]);
              }
              idx++;
            }

   // Jika ditemukan kata yang mirip dan distance lebih kecil
            if (dist < minim && dist != 0) {
              minim = dist;
              ans = item;
              found = true;
            }
          }
        }
      }

      if (found) {
        // Jika ditemukan perubahan
        ans = ans.toLowerCase();
       // Memasukkan kata baru ke list kalimat baru
        newArrayWords.push(ans);
        change = true;
      } else {
        // Jika tidak ditemukan perubahan
        // Memasukkan kata awal ke list kalimat baru
        newArrayWords.push(val);
      }
    }

    // Jika terdapat perubahan di antara kata yang ada dalam
kalimat
    if (change) {
      let newString = newArrayWords.join(" ");
      document.getElementById("checkQuery").innerHTML
=
        "Apakah " + input.toLowerCase() + " typo menjadi "
```

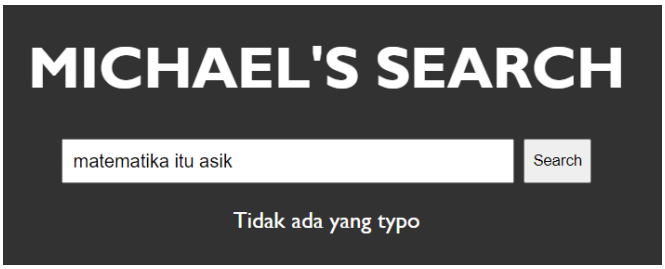
```
+ newString + " ? ";
    } else {
      // Jika tidak ada yang typo
      document.getElementById("checkQuery").innerHTML
= "Tidak ada yang typo";
    }
  };
```

Berikut adalah contoh tampilan dari aplikasi *website* yang telah dibahas pada makalah ini.



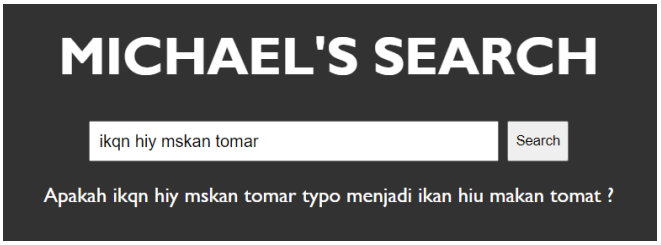
Gambar 5.1 Tampilan *website* untuk perbaikan *typo* dalam *search engine*

VI. Test Case

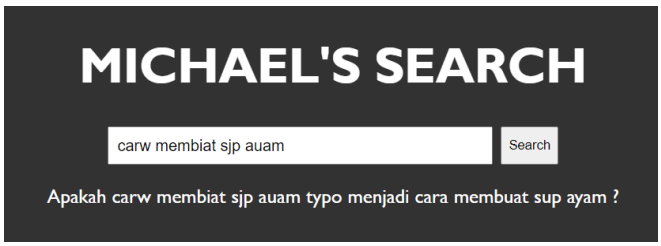


Gambar 6.1 Contoh untuk kalimat tidak *typo* pertama

Gambar 6.2 Contoh untuk kalimat tidak *typo* kedua



Gambar 6.3 Contoh untuk kalimat *typo* pertama



Gambar 6.4 Contoh untuk kalimat *typo* kedua

VII. Kesimpulan dan Saran

Struktur data graf dapat digunakan untuk memecahkan banyak permasalahan seperti salah satunya adalah mengatasi kesalahan tipografi. Graf sendiri memiliki beberapa jenis dan salah satunya yang sering digunakan adalah graf berarah dan berbobot. Dalam makalah ini, graf jenis tersebut dapat membantu kita mengetahui seberapa *error* sebuah pengetikan terjadi dan membuat program dapat mencari kata yang dapat menjadi salah satu referensi untuk perbaikan.

Namun, masih terdapat kelemahan dari program ini yang belum dapat ditangani seperti salah satunya adalah kamus bahasa Indonesia yang masih kurang lengkap sehingga tidak semua kasus kesalahan dapat ditangani oleh program. Alangkah baiknya jika program ini ingin dikembangkan, pengembang dapat menambahkan kata-kata bahasa Indonesia sesuai KBBI yang belum terdapat pada *file* .txt.

VIII. Ucapan

Saya panjatkan puji dan syukur kepada Tuhan Yang Maha Esa karena atas segala berkat dan rahmat-Nya, makalah ini dapat diselesaikan. Saya selaku penulis mengucapkan terima kasih kepada Ibu Dr. Fariska Zakhralativa Ruskanda sebagai dosen pengampu mata kuliah Matematika Diskrit yang telah mengajari penulis. Mohon maaf bila ada kesalahan dalam pembuatan makalah ini. Semoga makalah ini dapat berguna bagi pembaca.

IX. Lampiran

Berikut adalah link github untuk aplikasi yang telah dibuat pada makalah ini : https://github.com/maikeljh/makalah-matdis-FixTypo-13521124

References

[1] Aprilliant, Audhi. 2021. https://audhiaprilliant.medium.com/graph-theory-for-typo-corrector-using-python-how-it-can-improve-the-string-matching-algorithm-59df8ead3acb, diakses pada 1 Desember 2022.

[2] Munir, Rinaldi. 2020. https://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2020-2021/Graf-2020-Bagian1.pdf, diakses pada 1 Desember 2022.

[3] Navone, Estefania Cassingena. 2020. https://www.freecodecamp.org/news/dijkstras-shortest-path-algorithm-visual-introduction/, diakses pada 1 Desember 2022.

Pernyataan

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

Bandung, 10 Desember 2021

Michael Jonathan Halim
13521124