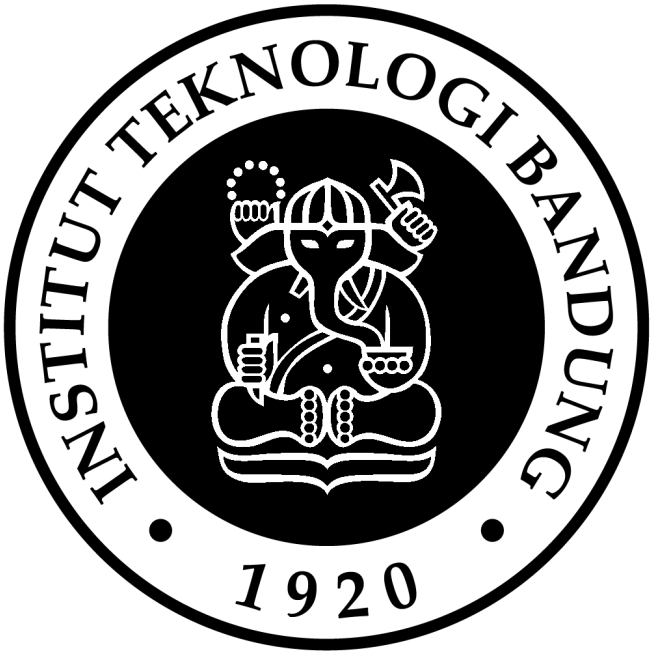
Tugas Kecil 3 IF2211 Strategi Algoritma
Penyelesaian Permainan Word Ladder Menggunakan Algoritma UCS, Greedy Best First Search, dan A*



Disusun oleh :

Atqiya Haydar Luqman (13522163)

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

2024

Daftar Isi

1. Pendahuluan

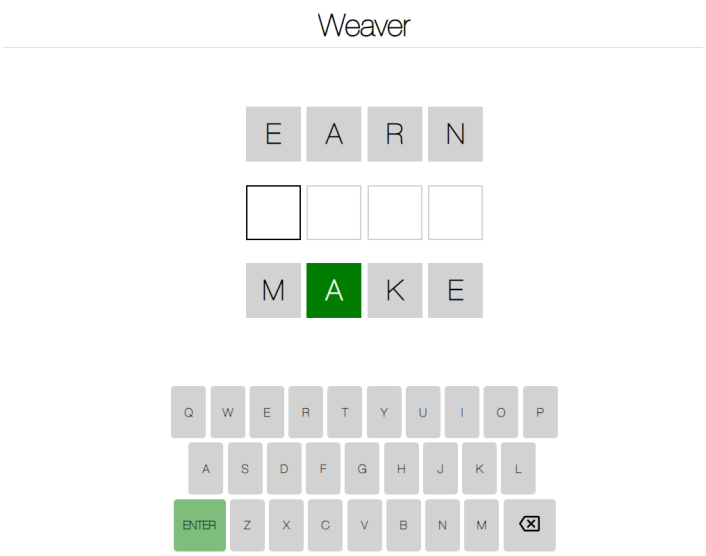
Tugas Kecil 3 ini merupakan bagian dari rangkaian tanggung jawab akademik kami sebagai mahasiswa Teknik Informatika di Institut Teknologi Bandung. Kami ditugaskan untuk merancang solusi penyelesaian permainan Word Ladder menggunakan algoritma Uniform Cost Search (UCS), Greedy Best First Search (GBFS), dan A* (AStar). Tujuan utama dari tugas ini adalah menghasilkan keluaran berupa path dari kata awal ke kata akhir yang dimasukkan, serta mencatat waktu eksekusi dan penggunaan memori oleh setiap algoritma.

Laporan ini disusun dengan tujuan memberikan pemahaman yang jelas dan komprehensif tentang solusi yang kami implementasikan, serta hasil analisis dari setiap algoritma yang digunakan. Melalui laporan ini, kami berharap dapat menggambarkan kontribusi kami dalam pengembangan solusi yang efektif dan efisien untuk permainan Word Ladder.

2. Analisis dan Implementasi Algoritma Pencarian Kata

a. Permainan Word Ladder

Word ladder (juga dikenal sebagai Doublets, word-links, change-the-word puzzles, paragrams, laddergrams, atau word golf) adalah salah satu permainan kata yang terkenal bagi seluruh kalangan. Word ladder ditemukan oleh Lewis Carroll, seorang penulis dan matematikawan, pada tahun 1877. Pada permainan ini, pemain diberikan dua kata yang disebut sebagai start word dan end word. Untuk memenangkan permainan, pemain harus menemukan rantai kata yang dapat menghubungkan antara start word dan end word. Banyaknya huruf pada start word dan end word selalu sama. Tiap kata yang berdekatan dalam rantai kata tersebut hanya boleh berbeda satu huruf saja. Pada permainan ini, diharapkan solusi optimal, yaitu solusi yang meminimalkan banyaknya kata yang dimasukkan
pada rantai kata. Berikut adalah ilustrasi serta aturan permainan.



Gambar 2.1 Ilustrasi Permainan
(https://wordwormdormdork.com/)

b. Uniform Cost Search Algorithm

Definisi Algoritma

Uniform Cost Search (UCS) merupakan salah satu algoritma pencarian jalur yang digunakan untuk menavigasi melalui grafik berbobot dengan tujuan mencari jalur terpendek dari titik awal tertentu ke titik tujuan yang diinginkan. Algoritma ini beroperasi dengan cara memeriksa node-node dalam graf secara sistematis, tetapi dengan memprioritaskan jalur-jalur yang memiliki biaya keseluruhan (total biaya yang ditempuh dari titik awal hingga node saat ini) yang paling rendah.

Pada setiap langkah, UCS akan memilih node berikutnya untuk dieksplorasi berdasarkan biaya jalur terkecil yang telah ditemukan hingga saat ini. Dengan kata lain, UCS selalu memilih jalur dengan biaya yang paling minimal hingga saat ini, sehingga memastikan bahwa jalur yang ditemukan adalah jalur terpendek yang diketahui dari titik awal ke setiap node. Hal ini membuat UCS sangat berguna dalam mencari jalur terpendek dalam graf yang memiliki bobot pada setiap edge-nya.

Salah satu keunggulan utama dari UCS adalah kemampuannya untuk menemukan jalur terpendek, meskipun graf yang dihadapi memiliki bobot yang berbeda-beda pada setiap edge-nya. Namun, kelemahan dari UCS adalah kemungkinan terjebak dalam pencarian jika biaya menuju tujuan tidak terbatas. Oleh karena itu, UCS cocok digunakan pada graf yang memiliki bobot edge yang relatif terbatas dan terbatasnya jumlah node yang dieksplorasi.

Definisi f(n) Pada Algoritma UCS

Dalam konteks algoritma pencarian, f(n) adalah fungsi heuristik atau estimasi biaya total yang diperlukan untuk mencapai tujuan dari suatu titik n dalam ruang pencarian. Dalam algoritma Uniform Cost Search (UCS), f(n) merupakan biaya total yang diperlukan untuk mencapai node n dari titik awal.

Perbedaan UCS dan BFS

UCS adalah algoritma pencarian yang memprioritaskan node-node yang memiliki biaya (cost) terendah dari titik awal ke node tersebut. Ini berbeda dengan Breadth-First Search (BFS), di mana BFS tidak mempertimbangkan biaya pada setiap langkahnya, melainkan hanya memeriksa node secara berlapis-lapis berdasarkan kedalaman dari titik awal. Dalam hal ini, UCS dan BFS memiliki perbedaan prinsipal dalam cara mereka memilih node berikutnya untuk dieksplorasi.

Implementasi Algoritma

Pada implementasinya, biaya akan diwakili oleh properti *cost* dari setiap node dalam struktur data *Node*.

```
static class Node {
        String word;
        ArrayList<String> path;
        int cost;

        Node(String word, ArrayList<String> path, int cost) {
            this.word = word;
            this.path = new ArrayList<>(path);
            this.path.add(word);
            this.cost = cost;
        }
    }
```

Fungsi prioritas pada *PriorityQueue* menggunakan nilai *cost* untuk memilih node berikutnya yang akan dieksplorasi.

```
PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
```

Langkah Implementasi :

1. Inisialisasi Variabel
   Algoritma dimulai dengan inisialisasi variabel-variabel seperti *PriorityQueue* untuk menyimpan node-node yang akan dieksplorasi,

Set untuk menyimpan node-node yang sudah dikunjungi, dan variabel nodeVisited untuk menghitung jumlah node yang telah dikunjungi.

```
Set<String> visited = new HashSet<>();
int nodeVisited = 0;
```

2. Penyisipian Node Awal ke PriorityQueue
   Node awal (*startWord*) disisipkan ke dalam *PriorityQueue* dengan biaya awal 0. Biaya untuk setiap node dihitung dengan menambahkan biaya jalur yang telah ditempuh dari titik awal hingga node saat ini.

3. Iterasi Melalui PriorityQueue
   Algoritma akan mengulangi langkah-langkah berikut selama *PriorityQueue* tidak kosong:
   - Mengambil node dengan biaya terendah dari *PriorityQueue*.
   - Memeriksa apakah node yang diambil adalah node tujuan (*endWord*). Jika ya, algoritma mengembalikan path yang ditemukan.
   - Jika node bukan node tujuan, algoritma akan mengeksplorasi tetangga-tetangga dari node tersebut.
   - Tetangga-tetangga yang belum pernah dikunjungi akan dimasukkan ke dalam *PriorityQueue* dengan biaya yang sesuai.

4. Perhitungan Node yang Dikunjungi
   Selama iterasi, variabel *nodeVisited* diinkrementasi setiap kali algoritma mengunjungi sebuah node.

c. Greedy Best First Search Algorithm

Definisi Algoritma

Greedy Best-First Search adalah salah satu algoritma pencarian dalam kecerdasan buatan yang digunakan untuk menemukan jalur terbaik dari suatu titik awal menuju tujuan. Algoritma ini bertujuan untuk memprioritaskan jalur yang paling menjanjikan berdasarkan suatu kriteria tertentu, tanpa mempertimbangkan secara menyeluruh apakah jalur tersebut merupakan jalur terpendek atau optimal.

Cara kerja Greedy Best-First Search dimulai dengan mengevaluasi biaya atau nilai heuristik dari setiap langkah yang mungkin dilakukan dari titik awal ke tujuan. Heuristik ini biasanya menggambarkan perkiraan jarak langsung dari suatu titik ke tujuan tanpa mempertimbangkan rintangan atau hambatan lainnya. Setelah nilai heuristik untuk setiap langkah dinilai, algoritma akan memilih langkah dengan nilai heuristik paling rendah, yang secara intuitif adalah langkah yang paling dekat dengan tujuan.

Proses ini diulangi secara berulang, di mana pada setiap langkah algoritma akan memilih langkah dengan nilai heuristik terendah dari titik saat ini menuju tujuan. Namun, perlu dicatat bahwa karena algoritma ini bersifat serakah (greedy), langkah-langkah yang diambil mungkin tidak selalu menghasilkan jalur terpendek atau optimal. Algoritma Greedy Best-First Search akan terus memperluas jalur dengan biaya terendah, tanpa mempertimbangkan secara menyeluruh kondisi jalur yang telah dilalui atau opsi yang mungkin lebih baik di masa depan.

Meskipun Greedy Best-First Search cenderung menghasilkan jalur yang cepat ditemukan dan sering kali menguntungkan dalam kasus-kasus di mana heuristik yang digunakan cukup akurat, namun kelemahan utamanya adalah kecenderungannya untuk terjebak dalam lokal minimum atau maksimum (tergantung pada apakah algoritma digunakan untuk mencari minimum atau maksimum) yang mungkin tidak menghasilkan solusi

optimal secara global. Selain itu, keputusan serakah algoritma ini dapat mengabaikan informasi penting tentang struktur masalah yang dapat membantu dalam menemukan solusi yang lebih baik secara keseluruhan.

Definisi g(n) Pada Algoritma Greedy Best First Search

Dalam konteks algoritma Greedy Best-First Search, g(n) adalah fungsi yang menyatakan biaya aktual untuk mencapai simpul n dari titik awal (atau simpul awal) dalam graf atau ruang pencarian.

Permasalahan Solusi yang Diberikan Algoritma Greedy Best First Search

Secara teoritis, algoritma Greedy Best-First Search tidak menjamin solusi optimal untuk persoalan Word Ladder. Algoritma Greedy Best-First Search memilih langkah yang memiliki nilai heuristik (biasanya jarak perkiraan dari kata saat ini ke kata tujuan) paling rendah pada setiap langkahnya. Namun, karena sifat serakah algoritma ini, ia cenderung untuk memilih langkah yang paling dekat dengan tujuan pada setiap langkah, tanpa mempertimbangkan secara menyeluruh bagaimana langkah-langkah tersebut akan memengaruhi keseluruhan jalur. Akibatnya, algoritma ini dapat terjebak dalam sebuah jalur yang tidak optimal, terutama jika nilai heuristiknya tidak cukup akurat untuk mencapai solusi optimal.

Untuk persoalan Word Ladder, di mana terdapat banyak kemungkinan jalur yang mungkin dan banyak kemungkinan kata-kata yang dapat digunakan sebagai langkah-langkah, algoritma Greedy Best-First Search mungkin tidak mampu mengeksplorasi secara menyeluruh ruang pencarian kata-kata untuk menemukan solusi optimal. Sebagai contoh, algoritma tersebut mungkin terjebak dalam mengambil langkah-langkah yang tampaknya mendekati tujuan namun akhirnya tidak menghasilkan jalur terpendek atau optimal.

Implementasi Algoritma

Mendefinisikan sebuah kelas *GBFS* yang berisi metode *GreedyBestFirstSearch* untuk menjalankan algoritma Greedy Best First Search.

```
public static SearchResult GreedyBestFirstSearch(String
startWord, String endWord, Set<String> dictionary)
```

Mendefinisikan kelas dalam kelas (*SearchResult*) yang berfungsi sebagai kontainer untuk menyimpan jalur solusi dan jumlah simpul yang telah dikunjungi.

```
static class SearchResult {
        ArrayList<String> path;
        int nodeVisited;

        SearchResult(ArrayList<String> path, int nodeVisited) {
            this.path = path;
            this.nodeVisited = nodeVisited;
        }
    }
```

Menggunakan stuktur data *PriorityQueue* untuk menyimpan simpul yang akan dieksplorasi berdasarkan nilai heuristik mereka.

```
PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node ->
node.heuristic));
```

Setiap simpul dalam *PriorityQueue* direpresentasikan oleh objek *Node* yang menyimpan kata, nilai heuristik, dan jalur yang telah dilalui.

```
static class Node {
        String word;
        int heuristic;
        ArrayList<String> path;

        Node(String word, int heuristic) {
            this.word = word;
            this.heuristic = heuristic;
            this.path = new ArrayList<>(Arrays.asList(word));
```

```
        }

        Node(String word, int heuristic, ArrayList<String>
path) {
            this.word = word;
            this.heuristic = heuristic;
            this.path = new ArrayList<>(path);
            this.path.add(word);
        }
    }
```

Metode *getNeighbors* Digunakan untuk mendapatkan tetangga-tetangga yang mungkin dari suatu kata. Tetangga-tetangga ini adalah kata-kata yang hanya memiliki satu perbedaan huruf dari kata sumber. Untuk setiap huruf dalam kata sumber, huruf tersebut diganti satu per satu dengan huruf-huruf alfabet, dan jika kata yang dihasilkan ada dalam kamus kata-kata valid, itu ditambahkan sebagai tetangga.

Metode *calculateHeuristic* Digunakan untuk menghitung nilai heuristik antara dua kata. Dalam implementasi ini, nilai heuristik dihitung sebagai jumlah perbedaan antara huruf-huruf pada posisi yang sesuai dari dua kata.

Langkah Implementasi :

1. Terima input berupa kata awal (startWord), kata akhir (endWord), dan kumpulan kata yang valid (dictionary).
2. Buat sebuah PriorityQueue untuk menyimpan simpul yang akan dieksplorasi, diurutkan berdasarkan nilai heuristik.
3. Buat sebuah Set untuk menyimpan kata-kata yang telah dikunjungi.
4. Tambahkan simpul awal ke dalam PriorityQueue.
5. Lakukan iterasi sampai PriorityQueue kosong:
    - Ambil simpul dengan nilai heuristik terendah dari PriorityQueue.
    - Tandai simpul tersebut sebagai dikunjungi.
    - Jika simpul tersebut adalah tujuan akhir, kembalikan jalur solusi.
    - Jika bukan, tambahkan tetangga-tetangga yang mungkin ke dalam PriorityQueue.

6. Jika tidak ada jalur solusi yang ditemukan, kembalikan null.

d. A* Algorithm

Definisi Algoritma

Algoritma A* adalah algoritma pencarian yang terinformasi, atau pencarian terbaik, yang dirumuskan dalam hal graf berbobot. Dimulai dari simpul awal tertentu dari graf, ia bertujuan untuk menemukan jalur ke simpul tujuan yang diberikan dengan biaya terkecil (jarak terpendek, waktu terpendek, dll.). Ini dilakukan dengan mempertahankan sebuah pohon jalur yang berasal dari simpul awal dan memperluas jalur-jalur tersebut satu edge pada satu waktu sampai mencapai simpul tujuan.

Pada setiap iterasi dari loop utamanya, A* perlu menentukan jalur mana yang akan diperluas. Ini dilakukan berdasarkan biaya jalur dan perkiraan biaya yang diperlukan untuk memperluas jalur sampai ke tujuan. Secara khusus, A* memilih jalur yang meminimalkan

$$f(n) = g(n) + h(n)$$

dimana n adalah simpul berikutnya pada jalur, g(n) adalah biaya jalur dari simpul awal ke n, dan h(n) adalah fungsi heuristik yang memperkirakan biaya jalur termurah dari n ke tujuan. Fungsi heuristik ini spesifik untuk masalah tertentu. Jika fungsi heuristiknya dapat diterima – artinya tidak pernah mengevaluasi biaya aktual untuk mencapai tujuan – A* dijamin akan mengembalikan jalur terkecil dari awal ke tujuan.

Admissable Heuristik Pada Algoritma A*

Heuristik yang digunakan pada algoritma A* dikatakan admissible jika memenuhi dua kondisi:

- Fungsi heuristik tidak pernah melebihi biaya aktual dari simpul saat ini ke simpul tujuan.
- Nilai heuristik untuk simpul tujuan harus nol.

Jadi, secara teoritis, heuristik yang digunakan dalam algoritma A* dianggap admissible jika memberikan perkiraan yang tidak melebihi biaya aktual dan memiliki nilai nol pada simpul tujuan.

Perbandingan Efisiensi Algoritma A* dengan UCS

Algoritma UCS akan memeriksa semua kemungkinan langkah dengan biaya yang sama dari titik awal sampai mencapai kata tujuan. Ini berarti akan memeriksa secara bertahap semua kemungkinan jalur dengan biaya yang semakin meningkat, tanpa mempertimbangkan informasi tambahan tentang tujuan.

Di sisi lain, algoritma A* menggunakan informasi heuristik untuk memilih jalur yang paling menjanjikan terlebih dahulu. Dengan mempertimbangkan estimasi biaya tersisa dari simpul saat ini ke simpul tujuan, algoritma A* cenderung lebih efisien daripada UCS dalam menemukan jalur terpendek. Namun, efisiensi ini tergantung pada kualitas heuristik yang digunakan. Jika heuristik yang digunakan tidak admissible atau tidak akurat, maka keuntungan efisiensi algoritma A* dapat berkurang.

Implementasi Algoritma

Langkah Implementasi :

1. Membuat sebuah *PriorityQueue* untuk menyimpan simpul yang akan dieksplorasi berdasarkan nilai f(n)

```
PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node ->
node.fValue));
```

2. Membuat sebuah *Map* untuk menyimpan biaya terbaik yang diketahui untuk mencapai setiap simpul

```
Map<String, Integer> gValues = new HashMap<>();
```

3. Membuat sebuah *Set* untuk menyimpan simpul-simpul yang sudah dieksplorasi

```
Set<String> visited = new HashSet<>();
```

4. Tambahkan simpul awal ke dalam *PriorityQueue* dengan nilai g(n) = 0 dan nilai f(n) = g(n) + h(n), di mana h(n) adalah heuristik yang menghitung jumlah karakter yang berbeda antara simpul awal dan simpul tujuan.

```
pq.offer(new Node(startWord, 0,
calculateHeuristic(startWord, endWord)));
```

5. Lakukan iterasi hingga *PriorityQueue* tidak kosong :
    - Ambil simpul dengan nilai f(n) terkecil dari pq.
    - Tandai simpul tersebut sebagai telah dieksplorasi dengan menambahkannya ke dalam visited.
    - Jika simpul tersebut merupakan simpul tujuan, kembalikan jalur yang ditemukan dan jumlah simpul yang dikunjungi.
    - Jika belum, cari tetangga-tetangga dari simpul tersebut :
        - Untuk setiap tetangga, hitung biaya baru untuk mencapainya dari simpul awal.
        - Jika biaya baru lebih baik dari biaya sebelumnya, perbarui nilai g(n), hitung kembali nilai f(n), dan tambahkan tetangga tersebut ke dalam pq.

6. Jika tidak ada jalur yang ditemukan, kembalikan null.

3. Source Code Program Implementasi

Tata Cara Menjalankan Program

1. Clonse Repositori

```
git clone https://github.com/AtqiyaHaydar/Tucil3_13522163
```

2. Pergi ke direktori 'Tucil3_13522163'

```
cd ./Tucil3_13522163/
```

3. Pergi ke foler *src*

```
cd ./src/
```

4. Jalankan file Java

```
javac Main.java UCS.java GBFS.java AStar.java
```

5. Kompilasi Main

```
java Main
```

Source Code

Main Program `main.java`

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
import java.util.Set;
import java.util.HashSet;
import java.util.ArrayList;
import java.lang.management.ManagementFactory;
import java.lang.management.MemoryUsage;

public class Main {
  static class SearchResult {
    ArrayList<String> path;
```

```
    int nodeVisited;

    SearchResult(ArrayList<String> path, int nodeVisited) {
        this.path = path;
        this.nodeVisited = nodeVisited;
    }
  }
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    Set<String> dictionary = loadDictionary("dictionary.txt");

    System.out.println("Welcome to Word Ladder Game!");

    Integer choice;
    do {
      System.out.println("Choose a Solving Algorithm: ");
      System.out.println("1. Uniform Cost Search");
      System.out.println("2. Greedy Best First Search");
      System.out.println("3. A* Search");
      System.out.println("4. Exit");

      do {
        System.out.print("Choice: ");
        choice = Integer.parseInt(scanner.nextLine());

        if (choice != 1 && choice != 2 && choice != 3 && choice
!= 4) {
          System.out.println("Invalid choice!");
        }
      } while (choice != 1 && choice != 2 && choice != 3 &&
choice != 4);

      if (choice.equals(4)) {
        System.out.println("Goodbye!");
        break;
      }

      String startWord, endWord;
      do {
        System.out.print("Enter the start word: ");
        startWord = scanner.nextLine();

        System.out.print("Enter the end word: ");
        endWord = scanner.nextLine();

        if (startWord.length() != endWord.length()) {
```

```
          System.out.println("The words must be the same
length!");
        }

        if (!dictionary.contains(startWord.toLowerCase()) ||
!dictionary.contains(endWord.toLowerCase())) {
          System.out.println("Start word or end word must be an
english word!");
        }
      } while (startWord.length() != endWord.length() ||
!dictionary.contains(startWord.toLowerCase()) ||
!dictionary.contains(endWord.toLowerCase()));

      Integer startTime = (int) System.currentTimeMillis();
      MemoryUsage heapMemoryUsage =
ManagementFactory.getMemoryMXBean().getHeapMemoryUsage();

      SearchResult result = null;
      UCS.SearchResult UCSresult = null;
      GBFS.SearchResult GBFSresult = null;
      AStar.SearchResult AStarresult = null;
      if (choice.equals(1)) {
        System.out.println("Solving with Uniform Cost Search");
        UCSresult = UCS.UniformCostSearch(startWord, endWord,
dictionary);

        if (UCSresult != null) result = new
SearchResult(UCSresult.path, UCSresult.nodeVisited);
      }
      else if (choice.equals(2)) {
        System.out.println("Solving with Greedy Best First
Search");
        GBFSresult = GBFS.GreedyBestFirstSearch(startWord,
endWord, dictionary);

        if (GBFSresult != null) result = new
SearchResult(GBFSresult.path, GBFSresult.nodeVisited);
      }
      else if (choice.equals(3)) {
        System.out.println("Solving with A* Search");
        AStarresult = AStar.AStarSearch(startWord, endWord,
dictionary);

        if (AStarresult != null) result = new
SearchResult(AStarresult.path, AStarresult.nodeVisited);
      }
```

```
      Integer endTime = (int) System.currentTimeMillis();
      Integer elapsedTime = endTime - startTime;

      long usedMemory = heapMemoryUsage.getUsed();

      if (result != null) {
        ArrayList<String> path = result.path;
        int nodeVisited = result.nodeVisited;
        for (String word: path) {
          System.out.println(word);
        }
        System.out.println("Number of nodes visited: " +
nodeVisited);
        System.out.println("Elapsed time: " + elapsedTime +
"ms");
        System.out.println("Memory used: " + usedMemory / 1024
+ " kb");
      } else {
        System.out.println(startWord +" is either not in the
dictionary or cannot be transformed into any other word.");
      }

      System.out.println("\nNew Game? (1: Yes, 2: No)");
      Integer newGame;
      do {
        System.out.print("Choice: ");
        newGame = Integer.parseInt(scanner.nextLine());

        if (newGame != 1 && newGame != 2) {
          System.out.println("Invalid choice!");
        }
      } while (newGame != 1 && newGame != 2);

      if (newGame.equals(2)) {
        System.out.println("Goodbye!");
        break;
      }
      else if (newGame.equals(1)) {
        System.out.println("\n << ========== New Game
========== >>");
      }

    } while (choice == 1 || choice == 2 || choice == 3);

    scanner.close();
```

```
  }

  // Function to load dictionary from file
  private static Set<String> loadDictionary(String filename) {
    Set<String> dictionary = new HashSet<>();
    try {
        Scanner fileScanner = new Scanner(new File(filename));
        while (fileScanner.hasNextLine()) {

dictionary.add(fileScanner.nextLine().trim().toLowerCase());
        }
        fileScanner.close();
      } catch (FileNotFoundException e) {
        System.out.println("Dictionary file not found: " +
e.getMessage());
      }
    return dictionary;
  }
}
```

Penjelasan

- Program akan menerima masukkan berupa String, yaitu startWord dan endWord
- Program akan menghasilkan keluaran class *SearchResult* yang memiliki atribut *path* (rute dari startWord menuju endWord) dan *nodeVisited* (jumlah node yang dikunjungi)

Uniform Cost Search `UCS.java`

```
import java.util.*;

public class UCS {
    static class Node {
        String word;
        ArrayList<String> path;
        int cost;

        Node(String word, ArrayList<String> path, int cost) {
            this.word = word;
            this.path = new ArrayList<>(path);
            this.path.add(word);
            this.cost = cost;
```

```
        }
    }

    static class SearchResult {
        ArrayList<String> path;
        int nodeVisited;

        SearchResult(ArrayList<String> path, int nodeVisited) {
            this.path = path;
            this.nodeVisited = nodeVisited;
        }
    }

    public static SearchResult UniformCostSearch(String
startWord, String endWord, Set<String> dictionary) {
        PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node -> node.cost));
        Set<String> visited = new HashSet<>();
        int nodeVisited = 0;

        pq.offer(new Node(startWord, new ArrayList<>(), 0));

        while (!pq.isEmpty()) {
            Node current = pq.poll();
            visited.add(current.word);

            if (current.word.equals(endWord)) {
                return new SearchResult(current.path,
nodeVisited);
            }

            List<String> neighbors = getNeighbors(current.word,
endWord, dictionary);
            // System.out.println("Neighbors dari " +
current.word + " Adalah " + neighbors);
            for (String neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    int newCost = current.cost + 1;
                    pq.offer(new Node(neighbor, current.path,
newCost));
                }
            }

            nodeVisited++;
        }
```

```
        return null;
    }

    private static List<String> getNeighbors(String word,
String endWord, Set<String> dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] wordChars = word.toCharArray();

        for (int i = 0; i < word.length(); i++) {
            char originalChar = wordChars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != originalChar) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (dictionary.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            wordChars[i] = originalChar;
        }

        return neighbors;
    }
}
```

Penjelasan

- Kelas UCS adalah implementasi algoritma Uniform Cost Search (UCS) dalam Java. Algoritma ini mencari jalur terpendek antara dua kata dalam kamus berbobot.
- Kelas Node merepresentasikan simpul dalam pencarian dengan atribut kata, jalur, dan biaya.
- Kelas SearchResult menyimpan hasil pencarian.
- Metode UniformCostSearch melakukan pencarian menggunakan PriorityQueue untuk mengurutkan simpul berdasarkan biaya terkecil.
- Metode getNeighbors menghasilkan tetangga-tetangga dari sebuah kata dalam kamus dengan mengubah satu karakter pada setiap posisi.

Greedy Best First Search `GBFS.java`

```
import java.util.*;

public class GBFS {
    static class SearchResult {
        ArrayList<String> path;
        int nodeVisited;

        SearchResult(ArrayList<String> path, int nodeVisited) {
            this.path = path;
            this.nodeVisited = nodeVisited;
        }
    }

    public static SearchResult GreedyBestFirstSearch(String
startWord, String endWord, Set<String> dictionary) {
        PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node ->
node.heuristic));
        Set<String> visited = new HashSet<>();

        pq.offer(new Node(startWord, 0));

        int nodeVisited = 0;
        while (!pq.isEmpty()) {
            Node current = pq.poll();
            visited.add(current.word);
            nodeVisited++;

            if (current.word.equals(endWord)) {
                return new SearchResult(current.path,
nodeVisited);
            }

            List<String> neighbors = getNeighbors(current.word,
endWord, dictionary);
            for (String neighbor : neighbors) {
                if (!visited.contains(neighbor)) {
                    int heuristic =
calculateHeuristic(neighbor, endWord);
                    pq.offer(new Node(neighbor, heuristic,
current.path));
                }
            }
        }
```

```
        return null;
    }

    private static List<String> getNeighbors(String word,
String endWord, Set<String> dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] wordChars = word.toCharArray();

        for (int i = 0; i < word.length(); i++) {
            char originalChar = wordChars[i];
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != originalChar) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (dictionary.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            wordChars[i] = originalChar;
        }

        return neighbors;
    }

    private static int calculateHeuristic(String word, String
endWord) {
        int heuristic = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != endWord.charAt(i)) {
                heuristic++;
            }
        }
        return heuristic;
    }

    static class Node {
        String word;
        int heuristic;
        ArrayList<String> path;

        Node(String word, int heuristic) {
            this.word = word;
            this.heuristic = heuristic;
            this.path = new ArrayList<>(Arrays.asList(word));
        }
```

```
        Node(String word, int heuristic, ArrayList<String>
path) {
            this.word = word;
            this.heuristic = heuristic;
            this.path = new ArrayList<>(path);
            this.path.add(word);
        }
    }
}
```

Penjelasan

- Kelas GBFS adalah implementasi algoritma Greedy Best-First Search (GBFS) dalam Java.
- Metode GreedyBestFirstSearch melakukan pencarian jalur terpendek antara dua kata dalam kamus dengan menggunakan heuristik yang mengutamakan simpul yang paling dekat dengan tujuan.
- Kelas Node merepresentasikan simpul dalam pencarian dengan atribut kata, heuristik, dan jalur yang sudah dilewati.
- Metode calculateHeuristic digunakan untuk menghitung heuristik antara kata saat ini dan kata tujuan.
- Metode getNeighbors menghasilkan tetangga-tetangga dari sebuah kata dalam kamus.

A* `AStar.java`

```
import java.util.*;

public class AStar {
    static class SearchResult {
        ArrayList<String> path;
        int nodeVisited;

        SearchResult(ArrayList<String> path, int nodeVisited) {
            this.path = path;
            this.nodeVisited = nodeVisited;
        }
    }

    public static SearchResult AStarSearch(String startWord,
```

```
String endWord, Set<String> dictionary) {
        PriorityQueue<Node> pq = new
PriorityQueue<>(Comparator.comparingInt(node -> node.fValue));
        Map<String, Integer> gValues = new HashMap<>();
        Set<String> visited = new HashSet<>();

        pq.offer(new Node(startWord, 0,
calculateHeuristic(startWord, endWord)));

        int nodeVisited = 0;
        while (!pq.isEmpty()) {
            Node current = pq.poll();
            visited.add(current.word);
            nodeVisited++;

            if (current.word.equals(endWord)) {
                return new SearchResult(current.path,
nodeVisited);
            }

            List<String> neighbors = getNeighbors(current.word,
endWord, dictionary);
            for (String neighbor : neighbors) {
                int newGValue =
gValues.getOrDefault(current.word, Integer.MAX_VALUE) + 1;
                if (newGValue < gValues.getOrDefault(neighbor,
Integer.MAX_VALUE)) {
                    gValues.put(neighbor, newGValue);
                    int fValue = newGValue +
calculateHeuristic(neighbor, endWord);
                    pq.offer(new Node(neighbor, newGValue,
fValue, current.path));
                }
            }
        }

        return null;
    }

    private static List<String> getNeighbors(String word,
String endWord, Set<String> dictionary) {
        List<String> neighbors = new ArrayList<>();
        char[] wordChars = word.toCharArray();

        for (int i = 0; i < word.length(); i++) {
            char originalChar = wordChars[i];
```

```
            for (char c = 'a'; c <= 'z'; c++) {
                if (c != originalChar) {
                    wordChars[i] = c;
                    String newWord = new String(wordChars);
                    if (dictionary.contains(newWord)) {
                        neighbors.add(newWord);
                    }
                }
            }
            wordChars[i] = originalChar;
        }

        return neighbors;
    }

    private static int calculateHeuristic(String word, String
endWord) {
        int heuristic = 0;
        for (int i = 0; i < word.length(); i++) {
            if (word.charAt(i) != endWord.charAt(i)) {
                heuristic++;
            }
        }
        return heuristic;
    }

    static class Node {
        String word;
        int gValue;
        int fValue;
        ArrayList<String> path;

        Node(String word, int gValue, int fValue) {
            this.word = word;
            this.gValue = gValue;
            this.fValue = fValue;
            this.path = new ArrayList<>(Arrays.asList(word));
        }

        Node(String word, int gValue, int fValue,
ArrayList<String> path) {
            this.word = word;
            this.gValue = gValue;
            this.fValue = fValue;
            this.path = new ArrayList<>(path);
            this.path.add(word);
```

```
            }
        }
}
```

Penjelasan

- Kelas AStar merupakan implementasi algoritma A* (A-star) dalam Java untuk mencari jalur terpendek antara dua kata dalam sebuah kamus.
- Metode AStarSearch menggunakan PriorityQueue untuk mengurutkan simpul berdasarkan nilai f, yang merupakan penjumlahan dari g-value (biaya aktual dari titik awal) dan h-value (heuristik perkiraan biaya ke titik tujuan). Algoritma iteratif, mengeksplorasi simpul secara berurutan hingga menemukan simpul tujuan. Setiap simpul yang dieksplorasi disimpan dalam set untuk menghindari pengulangan.
- Metode calculateHeuristic menghitung heuristik antara dua kata dengan mengestimasi biaya dari kata saat ini ke kata tujuan berdasarkan jumlah karakter yang berbeda.

4. Tangkapan Layar Input dan Output

Uniform Cost Search (UCS)

Test Case 1

```
Enter the start word: dive
Enter the end word: torn
Solving with Uniform Cost Search
dive
dire
dore
tore
torn
Number of nodes visited: 4629
Elapsed time: 96ms
Memory used: 81769 kb
```

Test Case 2

```
Enter the start word: nudge
Enter the end word: forks
Solving with Uniform Cost Search
nudge
pudge
podge
porge
porgy
porky
forky
forks
Number of nodes visited: 8793
Elapsed time: 137ms
Memory used: 43008 kb
```

Test Case 3

```
Enter the start word: hide
Enter the end word: tone
Solving with Uniform Cost Search
hide
tide
tine
tone
Number of nodes visited: 2194
Elapsed time: 44ms
Memory used: 41472 kb
```

Test Case 4

```
Enter the start word: water
Enter the end word: earth
Solving with Uniform Cost Search
water
dater
dates
dares
darts
dirts
girts
girth
garth
earth
Number of nodes visited: 286444
Elapsed time: 1738ms
Memory used: 67476 kb
```

Test Case 5

```
Enter the start word: breeze
Enter the end word: mingle
Solving with Uniform Cost Search
breeze
breese
greese
greete
greets
greats
treats
treads
trends
teends
teenes
tennes
bennes
benjes
bunjes
bunjee
bungee
bungle
bingle
mingle
Number of nodes visited: 206509
Elapsed time: 1294ms
Memory used: 141913 kb
```

Test Case 6

```
Enter the start word: juggle
Enter the end word: tumble
Solving with Uniform Cost Search
juggle
jungle
bungle
burgle
burble
bumble
tumble
Number of nodes visited: 1089
Elapsed time: 71ms
Memory used: 82794 kb
```

Greedy Best First Search

Test Case 1

```
Enter the start word: undo
Enter the end word: vest
Solving with Greedy Best First Search
undo
undy
unde
unce
unco
unto
into
inti
anti
ants
ante
anta
ansa
anoa
anow
anon
aeon
jeon
jean
jeat
jest
vest
Number of nodes visited: 44
Elapsed time: 18ms
Memory used: 76434 kb
```

Test Case 2

```
Enter the start word: coal
Enter the end word: fend
Solving with Greedy Best First Search
coal
foal
feal
feel
feed
fend
Number of nodes visited: 6
Elapsed time: 0ms
Memory used: 80530 kb
```

Test Case 3

```
Enter the start word: brawl
Enter the end word: yield
Solving with Greedy Best First Search
brawl
braws
brews
brers
biers
fiers
fiefs
liefs
liens
siens
sient
fient
fiend
field
yield
Number of nodes visited: 86
Elapsed time: 3ms
Memory used: 81554 kb
```

Test Case 4

```
Enter the start word: glove
Enter the end word: crisp
Solving with Greedy Best First Search
glove
clove
close
chose
chuse
cruse
crise
crisp
Number of nodes visited: 9
Elapsed time: 0ms
Memory used: 81554 kb
```

Test Case 5

```
Enter the start word: mingle
Enter the end word: fidget
Solving with Greedy Best First Search
mingle
dingle
pingle
pinole
pintle
wintle
winkle
windle
widdle
fiddle
faddle
fuddle
puddle
puddly
muddly
muddle
cuddle
cuddie
cuddin
cudden
hudden
hidden
ridden
ridded
ridged
fidged
fidget
Number of nodes visited: 285
Elapsed time: 6ms
Memory used: 81554 kb
```

Test Case 6

```
Enter the start word: jostle
Enter the end word: mumble
Solving with Greedy Best First Search
jostle
justle
hustle
hurtle
hurdle
huddle
puddle
buddle
bundle
bungle
burgle
burble
bumble
mumble
Number of nodes visited: 39
Elapsed time: 1ms
Memory used: 84626 kb
```

A*

Test Case 1

```
Enter the start word: raze
Enter the end word: whip
Solving with A* Search
raze
rale
wale
wald
waid
whid
whip
Number of nodes visited: 128
Elapsed time: 5ms
Memory used: 84626 kb
```

Test Case 2

```
Enter the start word: leap
Enter the end word: well
Solving with A* Search
leap
leal
weal
well
Number of nodes visited: 4
Elapsed time: 1ms
Memory used: 85650 kb
```

Test Case 3

```
Enter the start word: wind
Enter the end word: dusk
Solving with A* Search
wind
wink
dink
dunk
dusk
Number of nodes visited: 6
Elapsed time: 0ms
Memory used: 85650 kb
```

Test Case 4

```
Enter the start word: heart
Enter the end word: blame
Solving with A* Search
heart
heare
beare
blare
blame
Number of nodes visited: 6
Elapsed time: 0ms
Memory used: 85650 kb
```

Test Case 5

```
Enter the start word: breath
Enter the end word: border
Solving with A* Search
breath
breach
bleach
blench
clench
clunch
glunch
gaunch
paunch
paunce
pawnce
pawnee
pawner
panner
banner
bander
bonder
border
Number of nodes visited: 425
Elapsed time: 12ms
Memory used: 85650 kb
```

Test Case 6

```
Enter the start word: bird
Enter the end word: wing
Solving with A* Search
bird
bind
wind
wing
Number of nodes visited: 5
Elapsed time: 1ms
Memory used: 88722 kb
```

5. Analisis Perbandingan Solusi

Dalam menyelesaikan permasalahan Word Ladder, ketiga algoritma pencarian - Uniform Cost Search (UCS), Greedy Best-First Search (GBFS), dan A* - memiliki pendekatan yang berbeda. UCS memprioritaskan jalur dengan biaya terendah, memastikan solusi optimal, namun mungkin menjadi lambat tanpa pengetahuan tambahan. GBFS memilih jalur yang terlihat paling dekat dengan tujuan, lebih cepat tetapi tidak menjamin solusi optimal. A* menggabungkan biaya langkah aktual dengan nilai heuristik, menawarkan keseimbangan antara kecepatan dan keoptimalan, dengan catatan bahwa solusi optimal hanya dijamin jika nilai heuristiknya konsisten dan admissible. Dalam Word Ladder, keakuratan heuristik menjadi kunci dalam menentukan efektivitas dan keoptimalan solusi yang dihasilkan oleh GBFS dan A*.

Dari hasil tangkapan layar yang ada, terlihat bahwa Algoritma Uniform Cost Search (UCS) mengunjungi lebih banyak node dibandingkan dengan Greedy Best-First Search (GBFS). Hal ini menyebabkan waktu eksekusi dan penggunaan memori yang dibutuhkan oleh UCS lebih besar. Hal ini dapat dijelaskan karena UCS secara eksplisit mempertimbangkan biaya setiap langkah, yang mengakibatkan eksplorasi lebih banyak node untuk menemukan jalur dengan biaya terendah.

Sementara itu, Algoritma Greedy Best-First Search (GBFS) mengunjungi jumlah node yang lebih sedikit, sehingga menghasilkan waktu eksekusi yang lebih cepat dan penggunaan memori yang lebih rendah. Hal ini disebabkan oleh pendekatan GBFS yang hanya mempertimbangkan nilai heuristik untuk memilih simpul yang tampaknya paling dekat dengan tujuan, tanpa mempertimbangkan total biaya langkah. Meskipun GBFS dapat memberikan solusi dengan cepat, namun tidak menjamin solusi optimal karena cenderung terjebak dalam simpul yang tampaknya paling dekat dengan tujuan.

Algoritma A* menggabungkan keunggulan dari UCS dan GBFS. Dengan menggabungkan biaya langkah aktual dengan nilai heuristik, A* dapat menemukan solusi optimal jika nilai heuristiknya konsisten dan admissible. Dalam Word Ladder, nilai heuristik dapat dihitung sebagai jumlah perbedaan antara huruf-huruf pada posisi yang sesuai dari dua kata. Dengan memanfaatkan nilai heuristik ini, A* dapat memilih jalur yang paling menjanjikan untuk dieksplorasi, mengurangi jumlah simpul yang perlu dievaluasi dibandingkan dengan UCS, namun tetap mempertahankan jaminan solusi optimal.

Kesimpulan akhir dari analisis ini adalah bahwa UCS memastikan solusi optimal namun memerlukan waktu eksekusi dan penggunaan memori yang lebih besar. GBFS dapat memberikan solusi dengan cepat namun tidak menjamin solusi optimal. A* menawarkan keseimbangan antara kecepatan dan keoptimalan, memanfaatkan nilai heuristik untuk mengurangi kompleksitas pencarian sambil memastikan solusi optimal jika nilai heuristiknya konsisten dan admissible. Dengan demikian, pilihan algoritma tergantung pada kebutuhan spesifik dari masalah yang dihadapi serta keseimbangan antara waktu eksekusi, penggunaan memori, dan keoptimalan solusi yang diinginkan.

6. Pranala Repository Kode Program

https://github.com/AtqiyaHaydar/Tucil3_13522163

7. Kesimpulan

- UCS tidak sama dengan BFS. UCS menjamin bahwa jalur yang ditemukan memiliki biaya terendah, sedangkan BFS menemukan jalur terpendek berdasarkan jumlah langkah yang ditempuh, tanpa mempertimbangkan biaya pada setiap langkahnya.
- Secara teoritis, algoritma Greedy Best-First Search tidak menjamin solusi optimal untuk persoalan Word Ladder. Untuk persoalan Word Ladder, di mana terdapat banyak kemungkinan jalur yang mungkin dan banyak kemungkinan kata-kata yang dapat digunakan sebagai langkah-langkah, algoritma Greedy Best-First Search mungkin tidak mampu mengeksplorasi secara menyeluruh ruang pencarian kata-kata untuk menemukan solusi optimal. Sebagai contoh, algoritma tersebut mungkin terjebak dalam mengambil langkah-langkah yang tampaknya mendekati tujuan namun akhirnya tidak menghasilkan jalur terpendek atau optimal.
- Secara teoritis heuristik yang digunakan dalam algoritma A* dianggap admissible jika memberikan perkiraan yang tidak melebihi biaya aktual dan memiliki nilai nol pada simpul tujuan.
- Dengan mempertimbangkan estimasi biaya tersisa dari simpul saat ini ke simpul tujuan, algoritma A* cenderung lebih efisien daripada UCS dalam menemukan jalur terpendek. Namun, efisiensi ini tergantung pada kualitas heuristik yang digunakan. Jika heuristik yang digunakan tidak admissible atau tidak akurat, maka keuntungan efisiensi algoritma A* dapat berkurang.

8. Daftar Pustaka

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf

https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian2-2021.pdf

https://medium.com/@pranjalpanchawate/uniform-cost-search-ucs-is-special-case-of-a-algorithm-ca7f828c62fe#:~:text=Uniform%20Cost%20Search%20(UCS)%20is%20an%20algorithm%20employed%20for%20navigating,cost%20from%20the%20starting%20point.

https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

https://en.wikipedia.org/wiki/A*_search_algorithm#:~:text=A*%20is%20an%20informed%20search,shortest%20time%2C%20etc.).