Project Report

Data Structure and Algorithms

เรื่อง การแก้ปัญหา Word Ladder

เสนอ

รศ.ดร.รังสิพรรณ มฤคทัต

(Assoc.Prof. Rangsipan Marukatat, Ph.D)

จัดทำโดย

นาย นภัสรพี สถิตธรรม รหัสนักศึกษา 6513012

นายสรวิชญ์ ภัทรกุลดิลก รหัสนักศึกษา 6513122

นายทินภัทร พุ่มโพธิงาม รหัสนักศึกษา 6513166

นายวิชญ์ เสมาเงิน รหัสนักศึกษา 6513175

Department of computer Engineering

Faculty of Engineering, Mahidol University

คำนำ

รายงานฉบับนี้เป็นส่วนหนึ่งของวิชา Data Structure and Algorithms (EGCO 221) โดยคณะผู้จัดทำได้จัดทำขึ้นเพื่ออธิบายการทำงานของโปรแกรมแก้ปัญหา Word Ladder ซึ่งเป็นโปรแกรมที่ใช้ในการคำนวณหาเส้นทางที่สั้นที่สุดที่ในการ Ladder Word จากคำหนึ่งไปยังอีกคำหนึ่ง โดยใช้ Data Structure และ Algorithms มาประยุกต์ใช้ในการแก้ปัญหานี้ โดยในรายงานประกอบไปด้วยคู่มือการใช้งานโปรแกรม การอธิบายการทำงานของ Code และ Algorithms

คณะผู้จัดทำ

สารบัญ

คู่มือการใช้งาน

User Manual

1. หน้าแรกจะให้ผู้ใช้งาน Input เข้าไปโดย S = search , L = ladder , Q = quit โดยจะใส่ได้เพียง S,s L,l Q,q เท่านั้น ถ้าใส่นอกเหนือจากนี้จะต้องใส่ใหม่ไปเรื่อยๆ

```
... please wait : loading a file...
Enter menu >> (S = search, L = ladder, Q = quit)
```

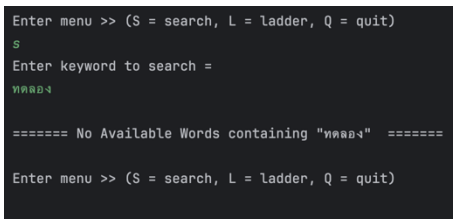
2. ถ้าหากกรอก S , s ไปจะเป็นหน้า Search

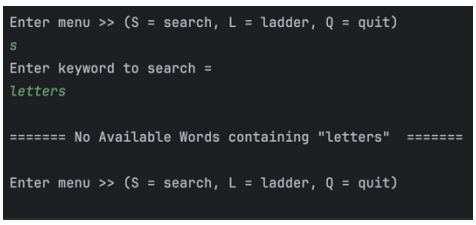
```
Enter menu >> (S = search, L = ladder, Q = quit)
s
Enter keyword to search =
```

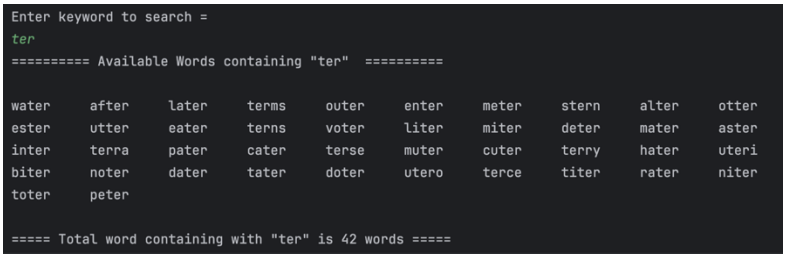
2.1. ผู้ใช้งานสามารถใส่ตัวอักษรเพื่อค้นหาคำได้ ถ้าหากมีคำที่มีตัวอักษรที่ผู้ใช้ค้นหาไป โปรแกรมจะแสดงคำที่มีตัวอักษรที่ผู้ใช้งานกรอกทั้งหมดที่มีในไฟล์

```
Enter menu >> (S = search, L = ladder, Q = quit)
s
Enter keyword to search =
eq
========== Available Words containing "eq" : ==========

equal     equip     deque

Enter menu >> (S = search, L = ladder, Q = quit)
```

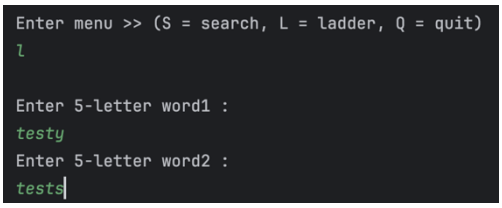
2.2. ถ้าผู้ใช้งานใส่ตัวอักษรที่ไม่ใช่ภาษาอังกฤษ หรือมากกว่า 5 ตัวอักษร โปรแกรมจะขึ้น “No available Words containing with (ที่ผู้ใช้ใส่) ” หากผู้ใข้ใส่ถูกต้องจะเป็น และจะบอกจำนวณคำที่เสิร์ชเจอในไฟล์ทั้งหมด

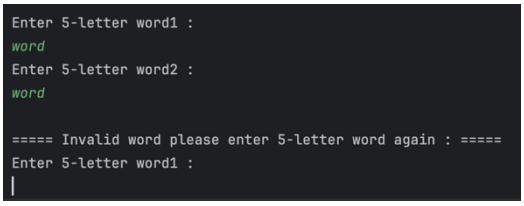




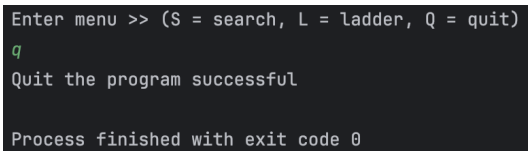


3. ถ้าผู้ใช้งานกรอก L (ladder) จะกลายเป็นหน้า ให้ใส่ word ที่ 1 และ word ที่ 2 โดยจะต้องใส่ word จะต้องเป็นตัวอักษรภาษาอังกฤษ 5 ตัวตามตัวอย่าง และเป็นศัพท์ที่มีในไฟล์ words_5757.txt ทั้งสองคำ โดยถ้าหากใส่นอกเหนือจากนี้จะต้องเริ่มใส่จาก word ที่ 1 ใหม่ ตามตัวอย่าง





4. ถ้าผู้ใช้งานกรอก Q (quit) จะเป็นการออกโปรแกรม



Graph Data Structure

(โครงสร้างข้อมูลของกราฟ)

1. Package org.jgrapht

```
import org.jgrapht.Graph;
import org.jgrapht.GraphPath;
import org.jgrapht.Graphs;
import org.jgrapht.alg.interfaces.ShortestPathAlgorithm;
import org.jgrapht.alg.shortestpath.DijkstraShortestPath;
import org.jgrapht.graph.DefaultWeightedEdge;
import org.jgrapht.graph.SimpleWeightedGraph;
```

เป็นการเรียกใช้งานไลบรารี่ของภาษา Java ที่อยู่ใน Package org.jgrapht ซึ่งนำมาใช้ในการสร้างกราฟ และการดำเนินการต่าง ๆ บนกราฟ ซึ่งประเภทของกราฟที่เราเลือกใช้ในโปรแกรมนี้ คือ Simple directed graph โดยอัลกอริทึมที่เราเลือกใช้ ได้แก่

- Dijkstra's Algorithm : ในเมธอด ladder() เพื่อหาเส้นทางที่สั้นที่สุดระหว่างคำศัพท์ที่ผู้ใช้ระบุ

2. Class WordGraph

เป็นคลาสที่ใช้ในการจัดการกับกราฟของคำศัพท์ โดยมีการสร้างกราฟเพื่อเก็บคำศัพท์และความสัมพันธ์ระหว่างคำศัพท์ต่าง ๆ ในรูปแบบของกราฟไม่ถ่วงน้ำหนัก (Unweighted Graph) และไม่กำหนดทิศทาง (Undirected Graph) โดยมีลักษณะสำคัญดังนี้

```
class WordGraph {  2 usages  PHUPH41622 +1
    private final Graph<String, DefaultWeightedEdge> wordGraph = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);
```

WordGraph : ซึ่งเป็นกราฟที่ใช้เก็บข้อมูลคำศัพท์ โดยมีลักษณะเป็นกราฟแบบน้ำหนัก (Weighted Graph) ที่โหนดเป็น String (คำศัพท์) และเส้นเชื่อมระหว่างโหนดเป็น DefaultWeightedEdge (เส้นเชื่อมเริ่มต้นของ JGraphT library)

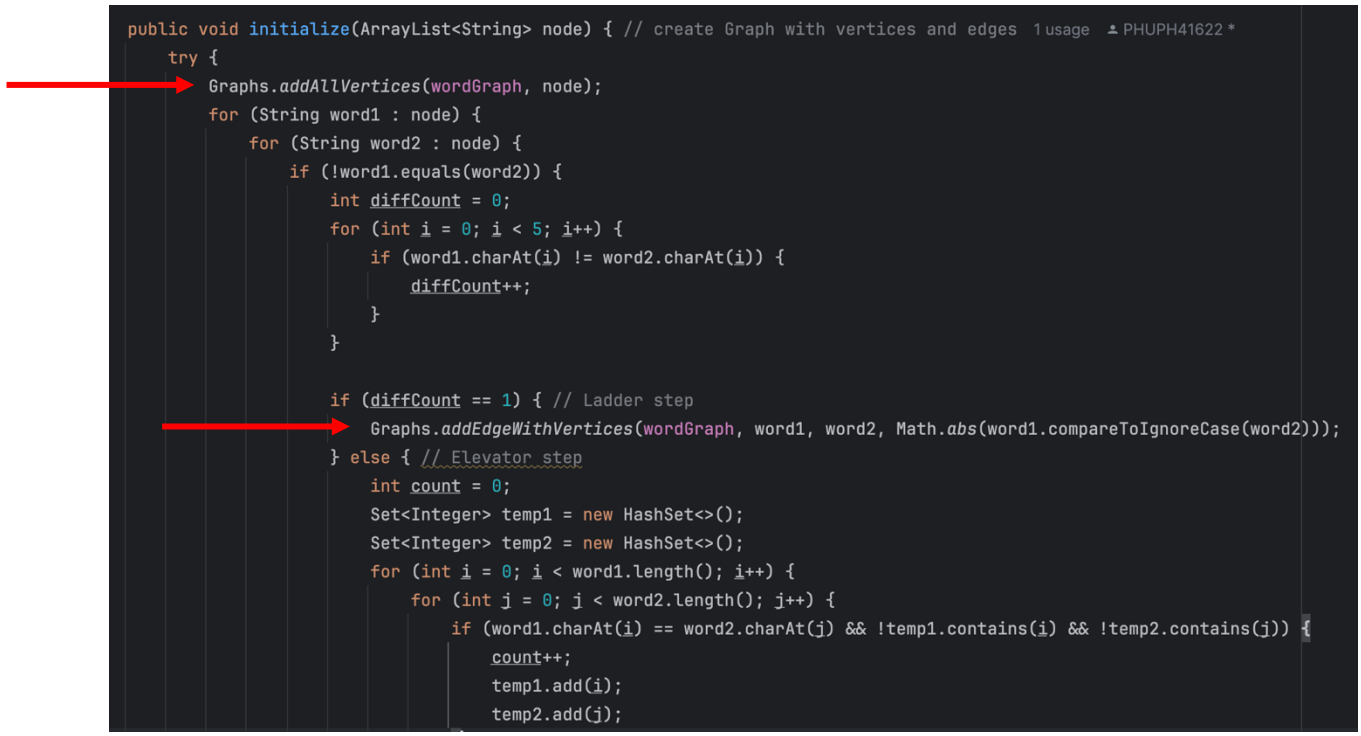
1. Attribute:

- **WordGraph:** เป็นกราฟที่ใช้เก็บคำศัพท์และความสัมพันธ์ระหว่างคำศัพท์ โดยมีลักษณะเป็นกราฟไม่ถ่วงน้ำหนัก (Unweighted Graph) และไม่กำหนดทิศทาง (Undirected Graph) ซึ่งถูกสร้างขึ้นโดยใช้คลาส SimpleWeightedGraph จากไลบรารี JGraphT.
- **keyboard:** เป็นอ็อบเจกต์ของคลาส Scanner ที่ใช้ในการรับข้อมูลจากผู้ใช้ผ่านทางคีย์บอร์ด โดยถูกใช้ในเมธอดต่าง ๆ ในคลาส WordGraph

```
class WordGraph {  2 usages  PHUPH41622 +1
    Scanner keyboard = new Scanner(System.in);  5 us
```

2. **Methods**

- **initialize(ArrayList<String> node):** เป็นเมธอดที่ใช้ในการสร้างกราฟโดยรับคำศัพท์เข้ามาในรูปแบบของ ArrayList และสร้างโหนด (vertices) และเชื่อมเส้น (edges) ระหว่างคำศัพท์ตามเงื่อนไขที่กำหนด โดยมีการตรวจสอบความแตกต่างของคำศัพท์ที่มีความแตกต่างกันเพียง 1 ตัวอักษรเพื่อเชื่อมเส้นระหว่างคำศัพท์นั้น ๆ โดยใช้เมธอด Graphs.addAllVertices() เพื่อเพิ่มโหนดและ Graphs.addEdgeWithVertices() เพื่อเพิ่มเส้นเชื่อมระหว่างคำศัพท์



```
public void initialize(ArrayList<String> node) { // create Graph with vertices and edges  1 usage  PHUPH41622 *
    try {
        Graphs.addAllVertices(wordGraph, node);
        for (String word1 : node) {
            for (String word2 : node) {
                if (!word1.equals(word2)) {
                    int diffCount = 0;
                    for (int i = 0; i < 5; i++) {
                        if (word1.charAt(i) != word2.charAt(i)) {
                            diffCount++;
                        }
                    }

                    if (diffCount == 1) { // Ladder step
                        Graphs.addEdgeWithVertices(wordGraph, word1, word2, Math.abs(word1.compareToIgnoreCase(word2)));
                    } else { // Elevator step
                        int count = 0;
                        Set<Integer> temp1 = new HashSet<>();
                        Set<Integer> temp2 = new HashSet<>();
                        for (int i = 0; i < word1.length(); i++) {
                            for (int j = 0; j < word2.length(); j++) {
                                if (word1.charAt(i) == word2.charAt(j) && !temp1.contains(i) && !temp2.contains(j)) {
                                    count++;
                                    temp1.add(i);
                                    temp2.add(j);
```

Graphs.addAllVertices()

- เป็นการเพิ่มโหนด (vertices) ในกราฟ wordGraph โดยใช้ข้อมูลจาก node ซึ่งเป็น ArrayList ของคำศัพท์ ซึ่งหมายความว่าโหนดในกราฟจะเป็นคำศัพท์ทั้งหมดใน node โดยโค้ดจะสร้างโหนดใหม่ในกราฟสำหรับแต่ละคำศัพท์ใน node โดยใช้เมธอด Graphs.addAllVertices() จากไลบรารี JGraphT

```
if (diffCount == 1) { // Ladder step
    Graphs.addEdgeWithVertices(wordGraph, word1, word2, Math.abs(word1.compareToIgnoreCase(word2)));
} else { // Elevator step
    int count = 0;
    Set<Integer> temp1 = new HashSet<>();
    Set<Integer> temp2 = new HashSet<>();
    for (int i = 0; i < word1.length(); i++) {
        for (int j = 0; j < word2.length(); j++) {
            if (word1.charAt(i) == word2.charAt(j) && !temp1.contains(i) && !temp2.contains(j)) {
                count++;
                temp1.add(i);
                temp2.add(j);
            }
        }
    }
    if (count == 5) {
        Graphs.addEdgeWithVertices(wordGraph, word1, word2, weight: 0);
```

if (diffCount == 1) { ... } else { ... }:

- เป็นการตรวจสอบค่า diffCount ซึ่งเป็นจำนวนตัวอักษรที่แตกต่างกันระหว่างคำศัพท์ word1 และ word2 หาก diffCount เท่ากับ 1 หมายความว่าคำศัพท์เหล่านี้มีเพียงตัวเดียวที่แตกต่างกันเท่านั้น จึงถูกสร้างเส้นเชื่อมระหว่างคำศัพท์เหล่านี้ในกราฟเพื่อแทนความสัมพันธ์แบบ "ladder" หรือ "ladder step" โดยใช้ Graphs.addEdgeWithVertices() โดยที่น้ำหนักของเส้นเชื่อมจะเป็นผลต่างของการเรียงลำดับอักขระของคำศัพท์ด้วย Math.abs(word1.compareToIgnoreCase(word2))) ซึ่งใช้เพื่อแสดงความห่างระหว่างคำศัพท์ที่แตกต่างกัน ในกรณีที่ diffCount ไม่เท่ากับ 1 หมายความว่าคำศัพท์เหล่านี้ไม่เป็นเพื่อนบ้านกัน จึงตรวจสอบในส่วนของ "elevator step" โดยตรวจสอบว่าคำศัพท์ทั้งสองมีตัวอักษรที่เหมือนกันทั้งหมดหรือไม่ หากมีตัวอักษรที่เหมือนกันทั้งหมด ซึ่งมีจำนวนตัวอักษรเท่ากับ 5 จะถูกเพิ่มเส้นเชื่อมในกราฟโดยใช้ Graphs.addEdgeWithVertices() โดยกำหนดน้ำหนักเป็น 0 เพื่อแสดงความสัมพันธ์แบบ "elevator" หรือ "elevator step" โดยที่คำศัพท์เหล่านี้มีส่วนที่เหมือนกันทั้งหมด
- **search():**

เป็นเมธอดที่ใช้ในการค้นหาคำศัพท์ที่มีส่วนประกอบที่ตรงกับคำค้นหาที่ผู้ใช้ระบุ โดยแสดงผลลัพธ์ออกทางหน้าจอ

- **ladder():**

เป็นเมธอดที่ใช้ในการหาเส้นทางที่สั้นที่สุดระหว่างคำศัพท์ที่ผู้ใช้ระบุ โดยใช้วิธีการค้นหาทางที่สั้นที่สุด (shortest path) ระหว่างคำศัพท์โดยใช้ Dijkstra's Algorithm ซึ่งถูกเรียกใช้ผ่านทางคลาส DijkstraShortestPath จากไลบรารี JGraphT

โครงสร้างข้อมูลต่างๆ

Other Data Structures

1. node ประกาศเป็นประเภท ArrayList<String> ซึ่งจะทำหน้าที่เก็บคำทั้งหมดที่อ่านมาได้จากไฟล์เพื่อนำไปสร้างเป็นกราฟต่อไป
2. wg ประกาศเป็นประเภท WordGraph ซึ่งถูกนิยามขึ้นในโปรแกรมนี้เพื่อใช้สำหรับการ search และ ladder โดยก่อนที่เราจะทำการ search และ ladder เราจะต้องนำ node ไปสร้างเป็นกราฟโดย wg.initailize(node);

```
16  public static void main(String[] args) {
17      String path = "src/main/java/Project2_6513122/";
18      String filename = "words_5757.txt";
19      Scanner keyboard = new Scanner(System.in);
20      WordGraph wg = new WordGraph();
21      boolean play = true, init = false, isMenu = true;
22      while(play) {
23          try {
24              Scanner filescan = new Scanner(new File( pathname: path+filename));
25              ArrayList<String> node = new ArrayList<>();
26              while(filescan.hasNext()) {
27                  String line = filescan.nextLine();
28                  node.add(line);
29              }
30              if(!init) {
31                  System.out.println("... please wait : loading a file...");
32                  wg.initialize(node);
33                  init = true;
34              }
```

3. WordGraph ประกาศเป็น pointer ประเภท Graph<String, DefaultWeightedEdge> เพื่อชี้ SimpleWeightedGraph<> ซึ่ง wordGraph จะนำ node มาใส่ใน WordGraph ให้เป็น vertices ของกราฟ และ weighted edge ของกราฟก่อนจะใส่ก็จะมีการเช็คว่าระหว่าง vertices เป็นกรณี Ladder หรือ Elevator ถ้าเป็น Ladder edge จะเป็นความต่างของตัวอักษรสองตัวที่ต่างกัน และ Elevator edge จะเป็น

4. temp1, temp2 ประกาศเป็น pointer ประเภท Set<Integer> เพื่อชี้ HashSet<> และ temp1, temp2 จะทำหน้าที่เก็บ index ของ charArray1 และ charArray2 ตามลำดับ โดยจะเก็บใน temp1, temp2 ก็ต่อเมื่อ element ของ index เหมือนกัน เช่น ตรวจสอบคำว่า weeny กับ weedy temp1 จะเก็บ 0, 1, 2, 4 และ temp2 จะเก็บ 0, 1, 2, 4 เช่นกัน และเหตุผลที่ใช้ HashSet เก็บก็เนื่องจาก HashSet จะไม่เก็บ duplicate element ซึ่งเหมาะสมกับอัลกอริทึมในส่วนนี้เพราะเมื่อเราตรวจสอบคำที่มีตัวอักษรซ้ำกันจะเกิดปัญหาเก็บ index ซ้ำได้แต่ HashSet จะช่วยแก้ปัญหาในส่วนนี้ได้ และเราไม่ได้สนลำดับของ element ที่เก็บใน HashSet

```
    if (diffCount == 1) { // Ladder step
        Graphs.addEdgeWithVertices(wordGraph, word1, word2, Math.abs(word1.compareToIgnoreCase(word2)));
    } else { // Elevator step
        int count = 0;
        Set<Integer> temp1 = new HashSet<>();
        Set<Integer> temp2 = new HashSet<>();
        for (int i = 0; i < charArray1.length; i++) {
            for (int j = 0; j < charArray2.length; j++) {
                if (charArray1[i] == charArray2[j] && !temp1.contains(i) && !temp2.contains(j)) {
                    count++;
                    temp1.add(i);
                    temp2.add(j);
                }
            }
        }
        if (count == 5) {
            Graphs.addEdgeWithVertices(wordGraph, word1, word2, weight: 0);
        }
    }
}
```

5. subword ประกาศเป็น String ทำหน้าที่เก็บคำที่ user ป้อนมา

```
public void search() {
    int count = 0;
    int total = 0;
    System.out.println("Enter keyword to search = ");
    String subword = keyboard.nextLine();
    List<String> validWords = new ArrayList<>();
    for (String word : wordGraph.vertexSet()) {
        if (word.contains(subword)) {
            validWords.add(word);
        }
    }
```

6. validWords ประกาศเป็นประเภท List<String> ทำหน้าที่เก็บ vertices ของ wordGraph ทั้งหมดที่ contains subword
7. allnodes ประกาศเป็นประเภท List<String> เพื่อใช้เก็บ vertices ของกราฟที่ส่งมาเป็น argument เพื่อทำการแสดง element ทั้งหมดของกราฟนั้นผ่าน allnodes
8. sumCost ประกาศเป็น Integer ใช้เก็บผลรวมของค่าน้ำหนักของเส้นเชื่อมทั้งหมดในเส้นทางที่เดินผ่าน

```
private void printGraphPath(GraphPath<String, DefaultWeightedEdge> gpath) {
    int sumCost = 0;
    List<String> allnodes = gpath.getVertexList();
    System.out.printf("start => %s\n", allnodes.get(0));
    for (int i = 0; i < allnodes.size()-1; i++) {
        int cost = (int)wordGraph.getEdgeWeight(wordGraph.getEdge(allnodes.get(i), allnodes.get(i+1)));
        sumCost += cost;
        System.out.printf("[ %s     ->     %s ]  ", allnodes.get(i), allnodes.get(i+1));
        if (cost == 0) {
            System.out.printf("====  ( elevator + %d)\n", cost);
        } else {
            System.out.printf("====  ( ladder   + %d)\n", cost);
        }
    }
    System.out.printf("end => %s\n\n", allnodes.get(allnodes.size()-1));
    System.out.println("=== Transformation cost : "+sumCost+" ===\n");
    System.out.println("================================================================");
}
```

Graph Algorithm

Dijkstra คืออัลกอริทึมที่ใช้ในการหาเส้นทางสั้นที่สุดระหว่างจุดสองจุดในกราฟ (graph) ที่มี weight เส้นเชื่อมระหว่าง Node แต่ละคู่ โดยมีการพิจารณาค่า weight เพื่อหาเส้นทางที่มีค่าน้อยที่สุดหรือระยะทางสั้นที่สุด

การทำงานของอัลกอริทึม Dijkstra มีขั้นตอนดังนี้:

1. เริ่มต้นที่จุดเริ่มต้นของกราฟ
2. กำหนดค่าระยะทางเริ่มต้นของจุดทุกจุดในกราฟเป็นอนันต์ ยกเว้นจุดเริ่มต้นเป็นศูนย์
3. ทำซ้ำไปเรื่อยๆ จนกว่าจะเคยไปทุก Node ในกราฟ
    a. เลือก Node ที่มีระยะทางที่น้อยที่สุดและยังไม่เคยไปเพื่อเป็น Node ปัจจุบัน
    b. ปรับระยะทางสำหรับ Node ที่เชื่อมต่อกับ Node ปัจจุบัน โดยใช้ weight ของเส้นทางที่ไปข้างยัง Node ข้างหน้าและค่าระยะทางปัจจุบัน
4. เมื่อไปทุก Node เรียบร้อยแล้วอัลกอริทึมจะสิ้นสุด

เหตุผลเลือกใช้ Dijkstra:

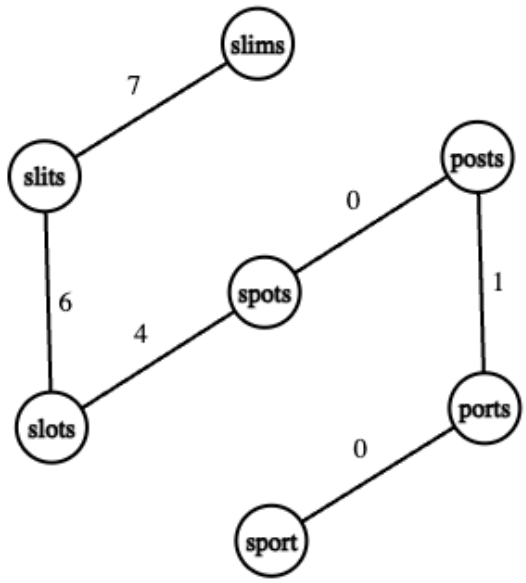
1. ความแม่นยำและประสิทธิภาพ Dijkstra: สามารถหาเส้นทางที่สั้นที่สุดระหว่างจุดสองจุดในกราฟได้อย่างแม่นยำ และมีประสิทธิภาพที่ดีในการทำงาน โดยมักจะมีเวลาการทำงานในระดับ O(V^2) หรือ O(E log V) โดยที่ V คือจำนวนของ Node ในกราฟและ E คือจำนวนเส้นทางในกราฟ
2. ใช้ได้กับกราฟที่มี positive weight: Dijkstra สามารถใช้กับกราฟที่มี weight และทางข้อมูลที่เราทำการ input เข้าไปในตัวโปรแกรม เป็น positive weight ทั้งหมด

```
ShortestPathAlgorithm<String, DefaultWeightedEdge> shpath = new DijkstraShortestPath<>(wordGraph);
```

Example find ladder from SPORT to SLIMS

```
========== solution from [ sport ] to [ slims ] ==========

start => sport
[ sport     ->     ports ]  ====  ( elevator + 0)
[ ports     ->     posts ]  ====  ( ladder   + 1)
[ posts     ->     spots ]  ====  ( elevator + 0)
[ spots     ->     slots ]  ====  ( ladder   + 4)
[ slots     ->     slits ]  ====  ( ladder   + 6)
[ slits     ->     slims ]  ====  ( ladder   + 7)
end => slims

=== Transformation cost : 18 ===

======================================================================
```

- sport -> ports edge มีค่าเป็น 0 กำหนดให้เมื่อ edge เท่ากับ 0 อยู่ใน elevator step (มีตัวอักษรเหมือนกันทั้งหมด แต่อยู่ในตำแหน่งที่ไม่เหมือนกัน)
- ports -> posts edge มีค่าเป็น 1 เมื่อค่า edge มากกว่า 0 อยู่ใน ladder step (ตัวอักษรต่างกัน 1 ตัวอักษรในตำแหน่งเดียวกัน)



-

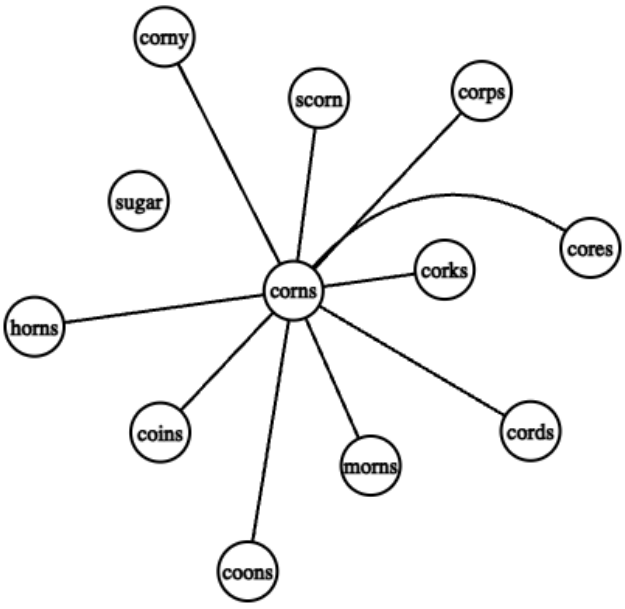
Example find ladder between CORNS and SUGAR

```
Enter 5-letter word1 :
corns
Enter 5-letter word2 :
sugar
No solution way from [ corns ] to [ sugar ]
```

ไม่มีคำตอบเนื่องจาก Dijkstra ไม่สามารถหา shortest path ได้ เพราะ การทำงานของ Dijkstra algorithm เป็นการค้นหา shortest path จากจุดเริ่มต้นไปยังทุกๆ จุดในกราฟ โดยทำการอัพเดตค่า shortest path ที่เคยเจอมาแล้ว และเลือกจุดที่มีค่า shortest path น้อยที่สุดในขณะนั้นเป็นจุดถัดไปที่จะตรวจสอบ จนกว่าจะได้ shortest path ไปยังทุกๆ จุด

การที่ Dijkstra algorithm ไม่สามารถหา shortest path ไปยัง Node Sugar นั้นเกิดจากข้อจำกัดของวิธีการทำงานของอัลกอริทึมนี้ เนื่องจาก Node Sugar ไม่มี edge ที่เชื่อมโยงไปยัง Node อื่น ๆ ในกราฟ

ตัวอย่างกราฟ



Limitation

(ข้อจำกัดโปรแกรม)

- ยังไม่พบข้อจำกัดของโปรแกรม

Referrence

(แหล่งอ้างอิง)

- Slide Chapter 8 เรื่อง Undirected graph
- Slide Chapter 9 เรื่อง Unweighted Graph, Weighted Graph – Dijkstra
- Slide Chapter 10 เรื่องเครื่องมือการใช้งานต่างๆของ org.jgrapht