Project Report

Data Structure and Algorithms

เสนอ

รศ.ดร.รังสิพรรณ มฤคทัต

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

จัดทำโดย

นางสาว	นิภัสสา	ชัยนิวัฒนา	รหัสนักศึกษา	6513114
นาย	พุฒิพงศ์	โยมะบุตร	รหัสนักศึกษา	6513134
นาย	ปาฏิหาริย์	เขม่นกิจ	รหัสนักศึกษา	6513170
นาย	ภัทรดนัย	สอนสว่าง	รหัสนักศึกษา	6513172

Department of computer Engineering

Faculty of Engineering, Mahidol University

คำนำ

รายงานฉบับนี้เป็นส่วนหนึ่งของวิชา Data Structure and Algorithms (EGCO 221) โดยคณะผู้จัดทำได้จัดทำขึ้นเพื่ออธิบายการทำงานของโปรแกรม Word Ladder ซึ่งเป็นโปรแกรมที่ใช้ในการตรวจสอบหาเส้นทางที่สั้นที่สุดจากคำที่กำหนดไปยังคำที่ต้องการ โดยใช้ Data Structure และ Algorithms มาประยุกต์ใช้ในการแก้ปัญหานี้ โดยในรายงานประกอบไปด้วยคู่มือการใช้งานโปรแกรม การอธิบายการทำงานของ Code และ Algorithms รวมไปถึงข้อจำกัดต่าง ๆ ของโปรแกรม

คณะผู้จัดทำขอขอบพระคุณ รศ.ดร.รังสิพรรณ มฤคทัต ผู้ให้ความรู้ และแนวทางในการศึกษา สุดท้ายนี้ทางคณะผู้จัดทำหวังว่ารายงานฉบับนี้ จะให้ความรู้และประโยชน์ไม่มากก็น้อยแก่ผู้อ่านทุกท่าน หากมีข้อผิดพลาดประการใด ผู้จัดทำจะขอน้อมรับไว้ และขออภัยมา ณ ที่นี้

คณะผู้จัดทำ

สารบัญ

เกี่ยวกับโปรแกรมนี้

โปรแกรมนี้มีใช้ตรวจสอบหาเส้นทางที่สั้นที่สุด และเก็บค่าจากการเปลี่ยนตัวอักษร จากคำศัพท์ที่กำหนด(Source) โดยมีเงื่อนไขการเปลี่ยนคำคือการเปลี่ยนตัวอักษร 1 ตัว(Ladder step) หรือสลับที่ตัวอักษร (Elevator step) ภายในคำนั้น และ มีการเก็บค่า Cost ซึ่งเกิดจากการทำ Ladder step และ Elevator step

โดยการทำ Ladder step 1 ครั้งจะมีการเก็บค่า cost จาก ระยะห่างระหว่างตัวตัวอักษรเก่าและตัวอักษรใหม่ เช่น these -> there มีการเปลี่ยนตัวอักษรจาก s เป็น r ทำให้ค่า cost = 1

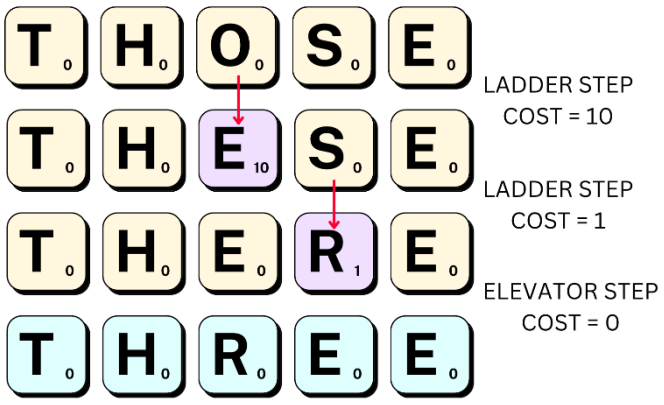
ส่วนการทำ Elevator step 1 ครั้งจะเก็บค่า cost = 0 เนื่องจากเป็นการสลับที่ของตัวอักษร ไม่ได้เกิดจากการเปลี่ยนตัวอักษร

เช่น there -> three มีการสลับที่ตัวอักษรจาก ere เป็น ree ทำให้เก็บค่า cost = 0

จนถึงคำที่ผู้ใช้กำหนดเป็นจุดหมายปลายทาง(Target)

ตัวอย่าง Ladder step และ Elevator step

SOURCE : THOSE
TARGET : THREE



Transformation Cost = 11

คู่มือการใช้งานโปรแกรม

1. เมื่อโปรแกรมเริ่มทำงาน จะให้ user ป้อนชื่อไฟล์เพื่อให้โปรแกรมอ่านคำศัพท์จากไฟล์ โดยไฟล์ทั้งหมดจะมี 2 ไฟล์ คือ **words_250.txt** และ **words_5757.txt**

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
```

(รูปสำหรับข้อที่ 1)

2. ในกรณีที่ user ใส่ชื่อไฟล์ถูกโปรมแกรมพา user มาที่ Main menu โดยในหน้านี้จะมีตัวเลือกให้ user พิมพ์ตัวอักษรดังต่อไปนี้คือ s, l, q, c
   โดยความหมายของตัวอักษรแต่ละตัว มีดังนี้
   - ตัวอักษร “s” ใช้สำหรับในการค้นหาและนับว่ามีคำศัพท์ที่ขึ้นต้นด้วย คำหรือตัวอักษรที่ user พิมพ์ลงไป
   - ตัวอักษร “l” ใช้สำหรับหาเส้นทางที่สั้นที่สุดจาก คำศัพท์ที่กำหนด(source) และ คำศัพท์ที่เป็นจุดหมายปลายทาง(Target) โดยทั้งสองคำจะให้ user เป็นคนพิมพ์ลงไป
   - ตัวอักษร “q” ใช้สำหรับปิดโปรแกรมหรือออกจากโปรแกรม
   - ตัวอักษร “c” ใช้สำหรับหาว่าคำศัพท์ที่ user พิมพ์ลงไปนั้น มีคำไหนที่สามารถทำ Ladder step 1 ครั้ง หรือ Elevator step 1 ครั้ง
   -

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
```

(รูปสำหรับข้อที่ 2)

2.1 กรณีที่ user เลือกพิมพ์ตัวอักษร “s” โปรแกรมจะให้ user พิมพ์คำหรือตัวอักษร

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
s

Search =
z

=== Available words ===
zones    zeros    zebra    zowie    zooms    zebus    zetas    zonal    zilch    zombi
zappy    zloty    zingy    zonks    zests    zings    zesty    zooks    zooey    zippy
zincs    zayin    zoned    zeals
Count = 24

=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
s

Search =
cor

=== Available words ===
coral    cords    cores    corps    corny    corks    corky    corns    cornu    corms
cored    cordy    corgi    corer
Count = 14

=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
```

(รูปสำหรับข้อที่ 2.1)

2.2 กรณีที่ user เลือกพิมพ์ตัวอักษร “l” โปรแกรมจะให้ user พิมพ์คำศัพท์ 2 คำ

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
l

Enter word_1 =
corns
Enter word_2 =
swing

corns -> corps (ladder   = +2)
corps -> crops (Elevator = +0)
crops -> drops (ladder   = +1)
drops -> drips (ladder   = +6)
drips -> grips (ladder   = +3)
grips -> grins (ladder   = +2)
grins -> rings (Elevator = +0)
rings -> tings (ladder   = +2)
tings -> sting (Elevator = +0)
sting -> swing (ladder   = +3)

Transformation Cost = 19

=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
```

(รูปสำหรับข้อที่ 2.2)

2.3 กรณีที่ user เลือกพิมพ์ตัวอักษร “q”

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
q


Process finished with exit code 0
```

(รูปสำหรับข้อที่ 2.3)

2.4 กรณีที่ user เลือกพิมพ์ตัวอักษร “c” โปรแกรมจะให้ user พิมพ์คำศัพท์ หรือ ตัวอักษร “b” เพื่อกลับมาที่ Main menu

```
Enter word  file = words_5757.txt
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
c

=========== Search Neighbors =========
Enter word (b = back to main menu):
crops
Neighbors of crops: [cross, drops, crows, chops, corps, props, crocs, coops, clops, craps]

Enter word (b = back to main menu):
sting
Neighbors of sting: [swing, stung, sling, stint, stink, suing, tings]

Enter word (b = back to main menu):
b
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
```

(รูปสำหรับข้อที่ 2.4)

Graph data structure

```
15        private Graph<String, DefaultWeightedEdge> graph;
          2 usages
```

```
24            graph = new SimpleWeightedGraph<>(DefaultWeightedEdge.class);
```

```
46        for (String word1 : words) {
47            for (String word2 : words) {
48                if (!word1.equals(word2)) {
49                    if (isNeighbor(word1, word2)) {
50                        double weight = getTransformationCost(word1, word2);
51                        graph.addEdge(word1, word2);
52                        graph.setEdgeWeight(graph.getEdge(word1, word2), weight);
53                    } else if (isElevator(word1, word2)) {
54                        graph.addEdge(word1, word2);
55                        graph.setEdgeWeight(graph.getEdge(word1, word2),  v: 0.0);
56                    }
```

- Graph ที่ได้เลือกใช้ในโปรแกรมนี้คือ Undirected Weighted Graph เนื่องจากในโปรแกรมมีการกำหนด Weight ระหว่างโหนด ซึ่งคือระยะห่างของแต่ละตัวอักษร โดยในโปรแกรมใช้ Class SimpleWeightedGraph จากไลบรารี JGraphT เพื่อสร้าง Object ของ Graph ที่ให้ชื่อว่า graph ในบรรทัดที่ 24 โดยกำหนดให้ Vertex เป็น String ซึ่งในที่นี้คือ words ที่ได้จากการอ่านค่าจากไฟล์ ส่วน Edge กำหนดเป็น DefaultWeightedEdge เพื่อกำหนดค่าเริ่มต้นที่เป็น 0,0 จากนั้นใช้ method getTransformationCost() ในบรรทัดที่ 50 เพื่อกำหนดค่า Weight ของกราฟซึ่งคือระยะห่างของแต่ละตัวอักษรที่ใช้ในการแปลงคำ

```
136           DijkstraShortestPath<String, DefaultWeightedEdge> dijkstra = new DijkstraShortestPath<>(graph);
137           GraphPath<String, DefaultWeightedEdge> path = dijkstra.getPath(startWord, endWord); //เก็บเส้นทางไว้ใน GraphPath
138
```

- ใช้ Dijkstra Algorithm ในการใช้หาระยะทางที่สั้นที่สุด โดยในโปรแกรมทำการสร้าง Object ของ DijkstraShortestPath ในบรรทัดที่ 136 และสร้าง Object GraphPath ในบรรทัดที่ 137 เพื่อเก็บ Path ที่ได้จากการใช้ Dijkstra algorithm

- โดยเงื่อนไขที่ใช้เช็คในการเพิ่ม Edge ระหว่างโหนด คือเช็คว่าโหนด ทั้ง 2 สามารถแปลงเป็นอีกตัวนึงผ่านวิธี Ladder step หรือ Elevator step ได้ไหม โดย Ladder step คือการแปลงเป็นอีกคำโดยการเปลี่ยนตัวอักษร 1 และ Elevator step คือการสลับที่ตัวอักษรภายในคำนั้น

- การเช็คว่าเป็น Ladder step จะใช้ method isNeighbor() ในการเช็คว่าเข้าเงื่อนไขของการเป็น Ladder step ไหม แล้วถ้าเป็นจะกำหนดค่า weight ของกราฟด้วย method getTransformationCost() โดย weight ที่กำหนดจะเป็นระยะห่างของตัวอักษรที่ต่างกัน

```java
public boolean isNeighbor(String a, String b) {
    if (a.length() != b.length()) return false;
    int differ = 0;
    for (int i = 0; i < a.length(); i++) {
        if (a.charAt(i) != b.charAt(i)) differ++;
        if (differ > 1) return false;
    }
    return differ == 1;
}
```

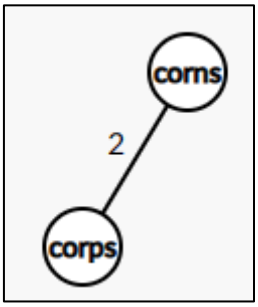
```java
private double getTransformationCost(String a, String b) {
    double cost =0;
    for (int i = 0; i < a.length(); i++) {
        if (a.charAt(i) != b.charAt(i)) {
            cost = a.charAt(i) - b.charAt(i);
            if(cost<0) cost=cost*(-1);
            break;
        }
    }
    return cost;
}
```

- การเช็คว่าเป็น Elevator step จะใช้ method isNeighbor() ในการเช็ค โดยถ้าเข้าเงื่อนไขของการเป็น Elevator step จะกำหนดให้ weight ของกราฟเป็น 0

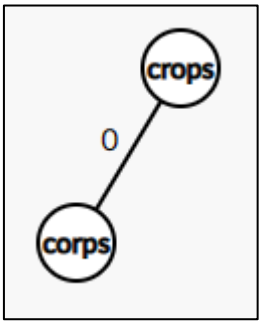
```java
public boolean isElevator(String a, String b) {
    if (a.length() != b.length()) return false;
    char[] charArrayA = a.toCharArray();
    char[] charArrayB = b.toCharArray();
    Arrays.sort(charArrayA);
    Arrays.sort(charArrayB);
    return Arrays.equals(charArrayA, charArrayB);
}
```

ตัวอย่างการเชื่อม edge

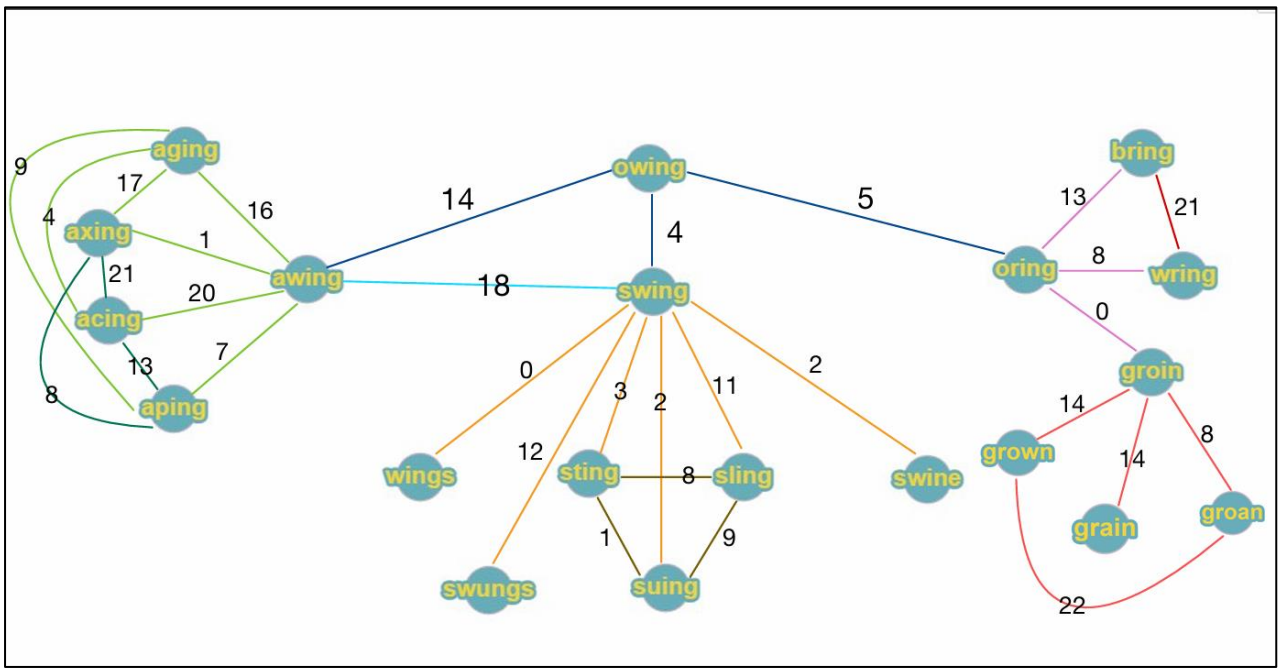
- corns และ corps เป็น Ladder step โดยตัว n และ ตัว p มีระยะห่าง 2 ตัวอักษรจึงเชื่อม Edge ระหว่างกันแล้วกำหนดให้ weight เป็น 2



- corps และ crops เป็น Elevator step จึงเชื่อม Edge แล้วกำหนดให้ weight เป็น 0



ตัวอย่างกราฟที่ได้จากโปรแกรม



Other data structures

Class main_Project2

```java
public class main_Project2 {
    public static void main(String[] args) {
        System.out.println("Enter word file = ");
        String filename;
        Scanner scanner = new Scanner(System.in);
        filename = scanner.nextLine();
        word_ladder wl  = new word_ladder(filename);
        wl.menu();
    }
}
```

คลาส **main_Project2** เป็น main class ซึ่งจะทำหน้าที่รับ input **ชื่อไฟล์ของคำศัพท์** ซึ่งมีทั้งหมด 2 ไฟล์ ได้แก่ words_250.txt และ words_5757.txt จากนั้นสร้าง object ของ word_ladder คือ wl และ constructor ของ class word_ladder จะเรียก method readAndBuildGraph จากนั้นจะเรียก method menu() ใน class word ladder เพื่อเข้าสู่ตัวโปรแกรมต่อไป

Class word_ladder

```
public class word_ladder {
    6 usages
    private List<String> words;
    14 usages
    private Graph<String, DefaultWeightedEdge> graph;
    2 usages
    private boolean running = true;
    1 usage
    public word_ladder(String filename) { readAndBuildGraph(filename); }
    1 usage
    private void readAndBuildGraph(String filename) {...}
    public void menu() {...}
    public void search(String searchString) {...}
    1 usage
    public void findWordLadder(String startWord, String endWord) {...}
    2 usages
    public boolean isNeighbor(String a, String b) {...}
    2 usages
    public boolean isElevator(String a, String b) {...}
    1 usage
    private double getTransformationCost(String a, String b) {...}
    1 usage
    public void searchNeighbor(){...}
    1 usage
    public void showNeighbors(String word_nei) {...}
}
```

ใน **class word_ladder** จะประกอบไปด้วย method ต่างๆ ได้แก่

1. **word_ladder()** เป็น constructor ที่เรียกใช้ method readAndBuildGraph()

2. **readAndBuildGraph()** เป็น method ที่อ่านไฟล์และสร้างกราฟ มีการใช้ ArrayList สำหรับเก็บค่า word นั่นคือคำศัพท์ที่โปรแกรมอ่านมาจากไฟล์และนำมาสร้างกราฟต่อ

3. **menu()** เป็น method ที่ไว้สำหรับแสดงหน้าเมนูและรับค่า input จาก User ได้แก่ “s” = search, “l” = ladder, “c” = searchNeighbor, “q” = quit

“s” คือ ค้นหาคำขึ้นต้นด้วยคำที่ต้องการค้นหา โดยเมื่อเลือกเมนู s แล้ว โปรแกรมจะให้ User ป้อนคำหรือคำขึ้นต้นของคำศัพท์ และเรียกใช้ method search() โดยจะส่งค่า word ที่ User ป้อนเข้ามา

“l” คือ การตรวจสอบและค้นหาเส้นทางที่สั้นที่สุด โดยเมื่อเลือกเมนู l แล้ว โปรแกรมจะให้ User ป้อนคำศัพท์เพื่อใช้เป็นตัวเริ่มต้น (source) และคำศัพท์เพื่อใช้เป็นจุดปลายทาง (target) จากนั้นจะเรียกใช้ method findWordLadder() โดยจะส่งค่าคำศัพท์ทั้ง2คำเพื่อตรวจสอบและค้นหาต่อไป

“c” จะเรียกใช้ method searchNeighbor() ซึ่งมีไว้หาโหนดเพื่อนบ้านที่อยู่ติดกัน นั่นคือ จะตรวจสอบเส้นทางที่เป็นไปได้ทั้งหมดของโหนดที่ User ต้องการ โดยจะกล่าวในส่วนของ searchNeighbor() ต่อไป

“q” จะเรียกใช้เมื่อต้องการจบโปรแกรม

4. **search()** เป็น method ที่ไว้สำหรับตรวจสอบคำศัพท์ที่ User ป้อนเข้ามาว่าตรงกับ คำศัพท์คำใดบ้างในไฟล์ และจะถูกจัดเก็บไว้ใน ArrayList ที่ถูกสร้างขึ้นภายใน method นี้ และจะแสดงคำศัพท์ทั้งหมดที่ตรงกับเงื่อนไขที่ User ป้อนเข้ามา และแสดงจำนวนของคำศัพท์ทั้งหมด

5. **findWordLadder()** เป็น method ตรวจสอบและค้นหาเส้นทางที่สั้นที่สุด

6. **isNeighbor()** เป็น method ที่ใช้เช็คว่า word 2 คำนี้เป็น Neighbor กันไหม โดยในที่นี้การที่จะเป็น Neighbor กัน word ทั้งสองต้องต่างกันแค่ 1 ตัวอักษร

7. **isElevator()** เป็น method ที่ใช้เช็คว่า word 2 คำนี้สามารถแปลงเป็นอีกตัว โดยการสลับที่ได้ไหม

8. **getTransformationCost()** เป็น method ที่ใช้คำนวณค่า Cost และจะ Return ค่าเป็น Cost ที่ใช้ในการแปลง word

9. **searchNeighbor()** เป็น method ที่รับคำศัพท์ที่ User ป้อนเข้ามา โดย method นี้จะวนลูปไปเรื่อยๆ จนกว่าจะได้รับที่ตรงกับเงื่อนไข (input = b or B) แล้วเรียก method showNeighbors() เพื่อส่งคำศัพท์ไปตรวจสอบต่อไป แล้วกลับไปยังส่วนของเมนู menu()

10. **showNeighbors()** เป็น method สำหรับตรวจสอบและค้นหาโหนดที่อยู่ติดกัน โดยใช้ ArrayList เพื่อตรวจสอบและเปรียบเทียบคำ**ศัพท์**ที่ตรงกับเงื่อนไขของ isNeighbor และ isElevator และสร้าง ArrayList เพื่อเก็บข้อมูลของคำศัพท์ที่ตรงกับเงื่อนไข พร้อมทั้งแสดงผลลัพธ์ออกมาในรูปแบบของ list

มีการใช้ **ArrayList** เพื่อทำให้ง่ายต่อการจัดเก็บข้อมูล และสะดวกในการเรียกใช้ข้อมูลภายในโครงสร้างเดียวกัน ได้แก่

```
20      private void readAndBuildGraph(String filename) {
21          String path = "src/main/java/Project2_6513134/";
22          words = new ArrayList<>();
```

words = new ArrayList<>(); ใน readAndBuildGraph() เป็นการนำข้อมูลประเภท List มาสร้าง ArrayList โดยมี object words ไว้สำหรับเก็บคำ**ศัพท์**ทั้งหมดที่อ่านจากไฟล์

```
100     public void search(String searchString) {
101         List<String> temp = new ArrayList<>();
```

List<String> temp = new ArrayList<>(); ใน search() เป็นการเก็บข้อมูลประเภท ArrayList จาก list ของ words โดย object temp สำหรับเก็บคำศัพท์ที่ตรงกับเงื่อนไขภายใน method

```
206     public void showNeighbors(String word_nei) {
207
208         List<String> neighbors = new ArrayList<>();
```

List<String> neighbors = new ArrayList<>(); ใน showNeighbors() เป็นการสร้าง ArrayList ที่มี object คือ neighbors เก็บคำศัพท์ที่ตรงกับเงื่อนไขของการเป็น Neighbor node โดย Neighbor node ในที่นี้หมายถึงโหนดที่ติดต่อกันในกราฟ โดยมีเส้นเชื่อม (edge) ติดกัน ซึ่งในที่นี้โหนดจะเป็นคำศัพท์ภายใน word ladder และโหนดที่ติดต่อกันโดยตรงคือคำศัพท์ที่สามารถแปลงกันได้ด้วยการเปลี่ยนตัวอักษรเพียงตัวเดียวหรือโหนดที่เป็นคำศัพท์ที่สามารถจัดเรียงตัวอักษรใหม่ให้เป็นคำศัพท์อีกคำหนึ่งได้

Graph algorithm

Graph algorithm ที่เลือกใช้คือ Dijkstra’s algorithm ซึ่งเป็นอัลกอริทึมการค้นหาเส้นทางสั้นที่สุดในกราฟที่มีน้ำหนักบนเส้นเชื่อมระหว่างจุด (weighted graph) โดยอัลกอริทึมนี้จะหาเส้นทางที่มีค่าน้ำหนักรวมต่ำที่สุดจากจุดเริ่มต้นไปยังจุดสิ้นสุด เนื่องจาก ระยะทางระหว่างโหนด (weight)จำเป็นต้องเป็นจำนวนเต็มบวกและยังเป็นอัลกอริทึมที่ใช้ในการหาเส้นทางที่สั้นที่สุดระหว่างสองโหนด

โดยหลักการทำงานของ Algorithm นี้คือ

```java
private void readAndBuildGraph(String filename) {
    String path = "src/main/java/Project2_6513134/";
    words = new ArrayList<>();
    graph = new SimpleDirectedWeightedGraph<>(DefaultWeightedEdge.class);
    boolean openfilesuccess = false;

    while(!openfilesuccess){
        try {
            File file = new File( pathname: path + filename);
            Scanner scan = new Scanner(file);
            openfilesuccess = true;
            while (scan.hasNext()) {
                String word = scan.nextLine().trim();
                words.add(word);
                graph.addVertex(word);
            }
            scan.close();
        } catch (FileNotFoundException e) {
            System.out.println("File not found: " + e.getMessage());
            System.out.println("Please Input New Filename = ");
            Scanner scanner = new Scanner(System.in);
            filename = scanner.nextLine();
        }
    }

    for (String word1 : words) {
        for (String word2 : words) {
            if (!word1.equals(word2)) {
                if (isNeighbor(word1, word2)) {
                    double weight = getTransformationCost(word1, word2);
                    graph.addEdge(word1, word2);
                    graph.setEdgeWeight(graph.getEdge(word1, word2), weight);
                } else if (isElevator(word1, word2)) {
                    graph.addEdge(word1, word2);
                    graph.setEdgeWeight(graph.getEdge(word1, word2), v: 0.0);
                }
            }
        }
    }

}
```

```java
public void findWordLadder(String startWord, String endWord) {
    try {
        if (!graph.containsVertex(startWord)) {
            System.out.println("Start word '" + startWord + "' is not in the graph.\n");
            return;
        }
        if (!graph.containsVertex(endWord)) {
            System.out.println("End word '" + endWord + "' is not in the graph.\n");
            return;
        }
        DijkstraShortestPath<String, DefaultWeightedEdge> dijkstra = new DijkstraShortestPath<>(graph);
        GraphPath<String, DefaultWeightedEdge> path = dijkstra.getPath(startWord, endWord);

        if (path != null) {
            List<DefaultWeightedEdge> edgeList = path.getEdgeList();

            for (DefaultWeightedEdge edge : edgeList) {
                String source = graph.getEdgeSource(edge);
                String target = graph.getEdgeTarget(edge);
                int weight = (int) graph.getEdgeWeight(edge);
                if (weight == 0) {
                    System.out.println(source + " -> " + target + " (Elevator = +" + weight + ")");
                } else {
                    System.out.println(source + " -> " + target + " (ladder   = +" + weight + ")");
                }

            }
            System.out.println("\n" + "Transformation Cost = " + (int) path.getWeight() + "\n");
        } else {
            System.out.println("Can not transform '" + startWord + "' into '" + endWord + "'.\n");
        }
    } catch (IllegalArgumentException e) {
        System.out.println("Error: " + e.getMessage());
    }
}
```

```java
private double getTransformationCost(String a, String b) {
    double cost =0;
    for (int i = 0; i < a.length(); i++) {
        if (a.charAt(i) != b.charAt(i)) {
            cost = a.charAt(i) - b.charAt(i);
            if(cost<0) cost=cost*(-1);
            break;
        }
    }
    return cost;
}
```

```java
public boolean isNeighbor(String a, String b) {
    if (a.length() != b.length()) return false;
    int differ = 0;
    for (int i = 0; i < a.length(); i++) {
        if (a.charAt(i) != b.charAt(i)) differ++;
        if (differ > 1) return false;
    }
    return differ == 1;
}
```

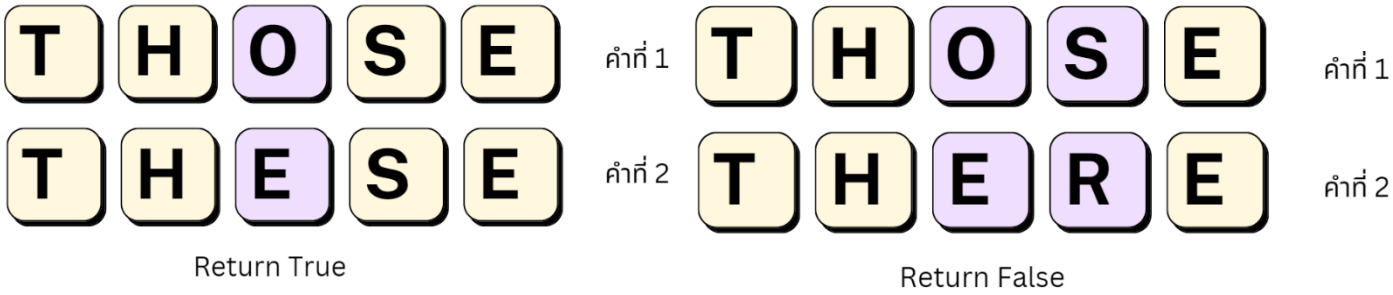
```java
public boolean isElevator(String a, String b) {
    if (a.length() != b.length()) return false;
    char[] charArrayA = a.toCharArray();
    char[] charArrayB = b.toCharArray();
    Arrays.sort(charArrayA);
    Arrays.sort(charArrayB);
    return Arrays.equals(charArrayA, charArrayB);
}
```

เงื่อนไขที่อัลกอริทึมจะเลือก ladder หรือ elevator ระหว่างคำ 2 คำ?

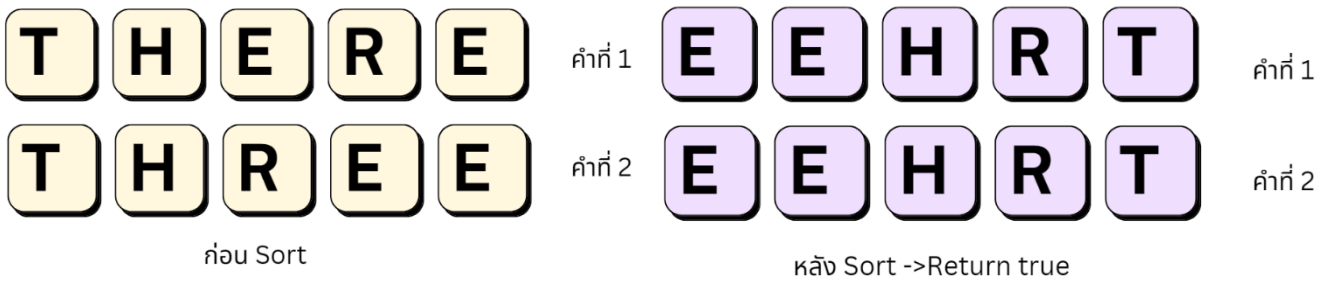
เมื่อเริ่มโปรแกรมจะอ่านคำจากไฟล์มาวน Loop เปรียบเทียบกันทุกคำ โดยจะใช้ฟังก์ชัน isNeighbor() กับ isElevator ในการเปรียบเทียบระหว่างคำ 2 คำนั้น ว่าจะเป็น ladder หรือ elevator โดยจะใช้

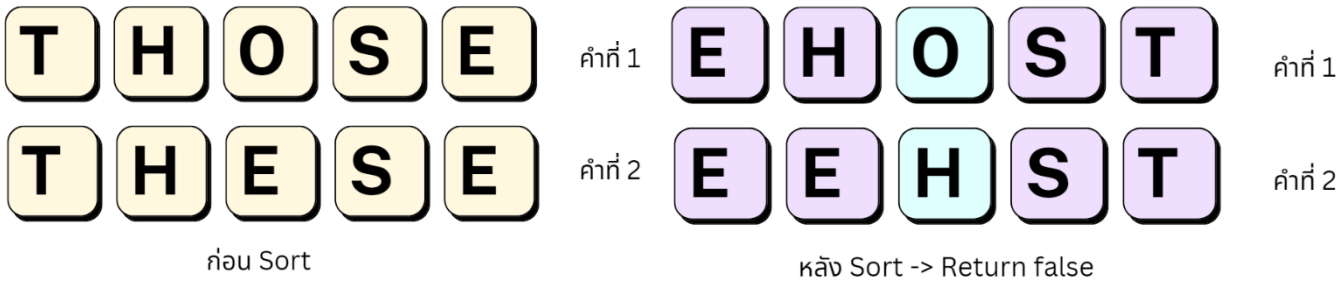
if-else if โดยจะเริ่มทำงานที่ isNeighbor() หากเป็นจริงจะเป็น ladder หากเป็นเท็จจะไปทำงานที่ isElevator() หากเป็นจริงจะเป็น elevator หากทั้ง 2 ฟังก์ชันเป็นเท็จจะไม่มีการเพิ่มเส้นเชื่อมระหว่างคำ 2 คำนั้น

1.isNeighbor() จะรับพารามิเตอร์ 2 ตัว คือคำ 2 คำ โดยจะวนลูปผ่านตัวอักษรแต่ละตัวของ คำแรกและคำที่สอง เพื่อตรวจสอบว่ามีตัวอักษรที่แตกต่างกันกี่ตัว หากมีตัวอักษรแตกต่างกัน 1 ตัวฟังก์ชันจะคืนค่าจริงกลับ แต่ถ้ามีตัวอักษรแตกต่างกันมากกว่า 1 ตัวขึ้นไปฟังก์ชันจะคืนค่าเท็จกลับไป



2.isElevator() จะรับพารามิเตอร์ 2 ตัว คือคำ 2 คำ เริ่มต้นฟังก์ชันจะเรียงตัวอักษรจาก A ไป Z ของคำที่ 1 และ คำที่ 2 แล้วนำมาเทียบกันว่าตัวอักษรทุกตัวเหมือนกันหรือไม่ ถ้าเหมือนกันทุกตัวฟังก์ชันจะคืนค่าจริงกลับไป แต่ถ้าไม่เหมือนกันทุกตัวอักษรฟังก์ชันจะคืนค่าเท็จกลับไป





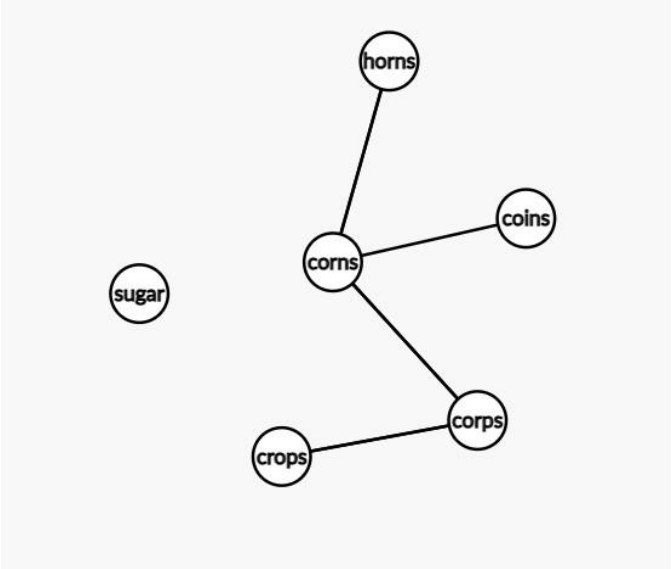
ทำไมถึงไม่สามารถเปลี่ยนคำ ‘corns’ เป็น ‘sugar’ ?

เนื่องจากโหนด sugar ไม่มีเส้นทางเชื่อมกับโหนดใดๆเลย สามารถพิสูจน์ได้จากการเรียกใช้ฟังก์ชัน searchNeighbor() ซึ่งจะเรียกใช้วนลูปเช็คทุกโหนด sugar กับทุกโหนดในกราฟและเรียกใช้ฟังก์ชัน isNeighbor() และ isElevetor() เพื่อเช็คว่าสามารถสร้างเส้นทางเชื่อมโหนด sugar กับโหนดนั้นได้ไหม

```
=========== Main menu ===========
Enter (s = search, l = ladder, q = quit, c = searchNeighbor):
c

=========== Search Neighbors =========
Enter word (b = back to main menu):
sugar

Neighbors of sugar: []
```



ตัวอย่างบางส่วนของกราฟ

Limitation

1. ในส่วน method ของ searchNeighbor เพื่อหาโหนดที่เชื่อมกันนั้น ถ้า input เป็นคำศัพท์ที่ไม่ได้อยู่ในไฟล์ที่โปรแกรมใช้ แต่โปรแกรมจะสมมุติว่ามีโหนดนั้นแล้วจะหาโหนดที่เชื่อมกันให้

ตัวอย่าง file : words_5757.txt

input = swing (มีคำว่า swing ในไฟล์ words_5757.txt)

```
Enter word (b = back to main menu):
swing

Neighbors of swing: [wings, swung, sting, owing, sling, swine, suing, awing]
```

input = siwng (ไม่มีมีคำว่า siwng ในไฟล์ words_5757.txt)

```
Enter word (b = back to main menu):
Siwng

Neighbors of siwng: [wings, swing]
```

2. ไม่สามารถเพิ่มโหนด (คำศัพท์) ในระหว่างที่โปรแกรมกำลังทำงาน หากต้องการเพิ่มโหนดในระหว่างที่กำลังรันโปรแกรมจะต้องออกจากโปรแกรมก่อน แล้วไปเพิ่มคำศัพท์ในไฟล์ที่เราต้องการ แล้วจึงรีนใหม่เพื่อให้โปรแกรมสามารถใช้คำล่าสุดที่เราเพิ่มไปได้

บรรณานุกรม

1. https://graphonline.ru/en/
2. https://csacademy.com/app/graph_editor/