Project Report

เรื่อง

Light out puzzle

เสนอ

รศ.ดร.รังสิพรรณ มฤคทัต

จัดทำโดย

นายศิรลักษณ์	ฑีฆะ	6213133
นายวีรวิชญ์	วงศ์ฉัตรชลีกุล	6213166
นายกรวิชญ์	วิเศษสุวรรณ	6213192

รายงานนี้เป็นส่วนหนึ่งของวิชา Data Structure and Algorithms

ภาคเรียนที่ 2 ปีการศึกษา 2563

คำนำ

รายงาน เรื่อง “ Light out puzzle ” เป็นส่วนหนึ่งของวิชา EGCO 221 Data Structure and Algorithms ในภาคเรียนที่ 2 ปีการศึกษา 2563 จัดทำขึ้น โดยมีจุดประสงค์เพื่อนำเสนอผลการปฏิบัติตามกระบวนการเขียนรายงานตามที่ได้ศึกษา อันเป็นการฝึกฝนทักษะกระบวนการเขียนโปรแกรมคอมพิวเตอร์และกระบวนการศึกษาค้นคว้าหาความรู้อย่างเป็นระบบ อีกทั้งเพื่อเสริมสร้างความรักในการศึกษาค้นคว้าหาความรู้ด้วยตนเอง อันเป็นหัวใจสำคัญที่จะเอื้อให้เกิดประโยชน์ในการศึกษาระดับสูงต่อไป

ประโยชน์ที่คณะผู้จัดทำได้รับจากการเขียนรายงานเรื่องนี้ คือ ได้ฝึกฝนกระบวนการทำงานกลุ่ม การวางแผนงาน การปรับปรุงแก้ไขงานการศึกษาค้นคว้า การฝึกทักษะในการเขียนโปรแกรมคอมพิวเตอร์ เป็นต้น นอกจากนี้ประโยชน์ที่ผู้ศึกษาจะได้รับจากรายงานนี้คือ ความรู้ความเข้าใจเกี่ยวกับ ลักษณะองค์ประกอบ และการใช้งานของภาษา JAVA ในการนำเอามาเขียนโปรแกรมคอมพิวเตอร์ อันจะเป็นการจุดประกายทางความคิดให้แก่ผู้ศึกษาที่จะนำความรู้เกี่ยวกับการเขียนเขียนโปรแกรมคอมพิวเตอร์ด้วยภาษา JAVA ไปปรับประยุกต์ใช้ในชีวิตประจำวันให้เกิดประโยชน์สูงสุดต่อไป

ผู้จัดทำหวังเป็นอย่างยิ่งกว่า รายงานเรื่องนี้จะเอื้อประโยชน์ในการศึกษาค้นคว้าแก่ผู้ที่สนใจไม่มากก็น้อย หากมีข้อบกพร่องประการใด ทางผู้จัดทำขอน้อมรับไว้เพื่อปรับปรุงแก้ไขในโอกาสต่อไป

คณะผู้จัดทำ

3 พฤษภาคม 2564

สารบัญ

เรื่อง **หน้า**

คู่มือการใช้งานโปรแกรม

Main Menu

1. ให้ user เลือกตัวเลือกที่ต้องการใช้งานจาก 3 ตัวเลือก

ตัวอย่างการเลือกใช้งานแต่ละตัวเลือก

ตัวเลือกที่ 1 : แก้ปัญหา puzzle

```
--------------------< com.mycompany:lightoutpuzzle >--------------------
Building lightoutpuzzle 1.0-SNAPSHOT
--------------------------------[ jar ]---------------------------------

--- exec-maven-plugin:3.0.0:exec (default-cli) @ lightoutpuzzle ---
Welcome to Lightout puzzle solve program
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity
1
Number of row for square grid =
```

ตัวเลือกที่ 2 : ข้อมูลเกี่ยวกับ Light out puzzle

```
Welcome to Lightout puzzle solve program
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity
2

=================================================================
                      Lights out puzzle game
  Lights Out is a puzzle game. consisting of a grid of lights
  that are either on or off Pressing any light will toggle
  it and its adjacent lights. The goal of the game is to switch
  all the lights off.

     you can play it on http://daattali.com/shiny/lightsout/
=================================================================
```

ตัวเลือกที่ 3 : ออกจากโปรแกรม

```
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity

please input choice on menu
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity
3
Exit program thx to play
------------------------------------------------------------------------
BUILD SUCCESS
------------------------------------------------------------------------
```

เมื่อ user เลือกตัวเลือกที่ 1 Solve puzzle จาก Main Menu

```
Welcome to Lightout puzzle solve program
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity
1
Number of row for square grid =
3
Initial states (9 bits, left to right, line by line)
110010001
```

2. กรอกขนาดของตารางที่ต้องการ (Number of row from square grid) โดยตารางมีลักษณะเป็นสี่เหลี่ยมจัตุรัส จากตัวอย่าง 3 หมายถึง ตารางขนาด 3 × 3 โดยค่า input ต้องเป็น positive Integer 3 - 5

3. กรอกค่าเริ่มต้นที่ต้องการ จากโปรแกรมให้มีค่าเป็น 9 bit จากซ้ายไปขวา (เลขฐาน 2) โดยค่าที่กรอกเป็นได้เพียงแค่ เลข 0 กับ 1 เพียงเท่านั้น

```
Bit string  = 110010001, Decimal ID = 275
      | col 0 | col 1 | col 2
row 0 |   1   |   1   |   0
row 1 |   0   |   1   |   0
row 2 |   0   |   0   |   1

Please wait a minute some states have more time to find solution...
3 moves to turn off all light

Start
      | col 0 | col 1 | col 2
row 0 |   1   |   1   |   0
row 1 |   0   |   1   |   0
row 2 |   0   |   0   |   1

>>> move 1 : Row 0 Colum 0
      | col 0 | col 1 | col 2
row 0 |   0   |   0   |   0
row 1 |   1   |   1   |   0
row 2 |   0   |   0   |   1

>>> move 2 : Row 2 Colum 0
      | col 0 | col 1 | col 2
row 0 |   0   |   0   |   0
row 1 |   0   |   1   |   0
row 2 |   1   |   1   |   1

>>> move 3 : Row 2 Colum 1
      | col 0 | col 1 | col 2
row 0 |   0   |   0   |   0
row 1 |   0   |   0   |   0
row 2 |   0   |   0   |   0
Puzzle finish
back to start
```

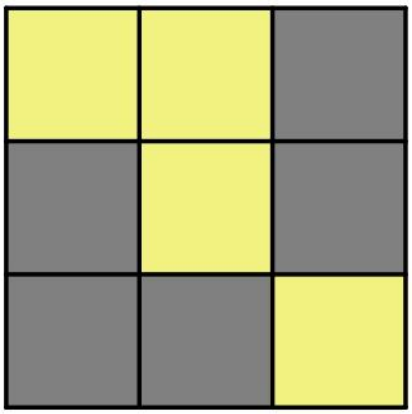
4. หน้าจอจะมีการแสดงตารางเริ่มต้น และวิธีการแก้ puzzle ออกมา

เลข 1 จะมีสถานะเป็น เปิดไฟ (สีเหลือง)

เลข 2 จะมีสถานะเป็น ปิดไฟ (สีเทา)

ตารางเริ่มต้น

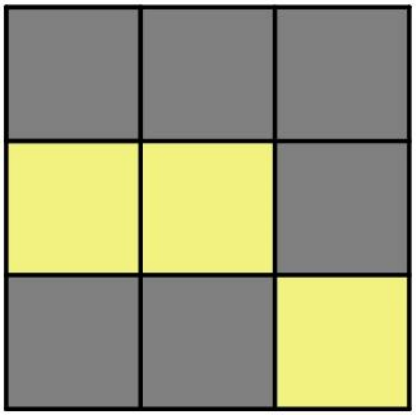
```
Bit string  = 110010001, Decimal ID = 275
      | col 0 | col 1 | col 2
row 0 |   1   |   1   |   0
row 1 |   0   |   1   |   0
row 2 |   0   |   0   |   1
```



Move 1 : Row 0 Column 1

```
Start
      | col 0 | col 1 | col 2
row 0 |   1   |   1   |   0
row 1 |   0   |   1   |   0
row 2 |   0   |   0   |   1

>>> move 1 : Row 0 Colum 0
```

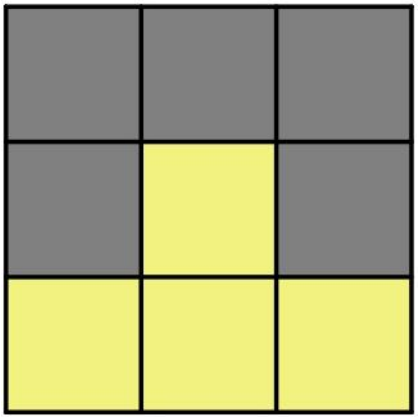
จะได้ตารางใหม่ออกมา



Move 2 : Row 2 Column 0

```
      | col 0 | col 1 | col 2
row 0 |   0   |   0   |   0
row 1 |   1   |   1   |   0
row 2 |   0   |   0   |   1

>>> move 2 : Row 2 Colum 0
```

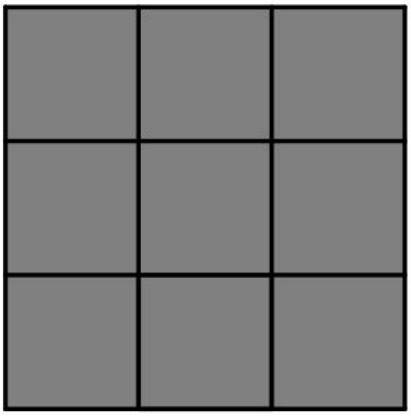
จะได้ตารางใหม่ออกมา



Move 3 : Row 2 Column 1

```
>>> move 3 : Row 2 Colum 1
      | col 0 | col 1 | col 2
row 0 |   0   |   0   |   0
row 1 |   0   |   0   |   0
row 2 |   0   |   0   |   0
Puzzle finish
back to start
```

จะได้ตารางใหม่ออกมา

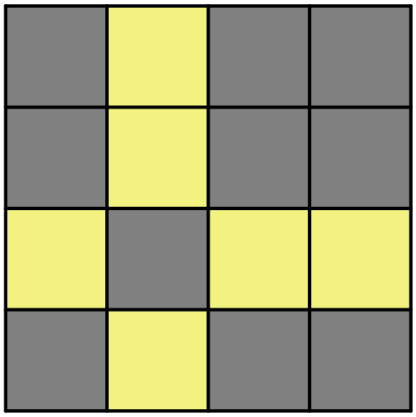


```
Welcome to Lightout puzzle solve program
1. Solve puzzle
2. Information about light out puzzle
3. Exit program
choose your Activity
```

5. เมื่อทำการทำงานในหัวข้อที่ 1 เสร็จสิ้น จะกลับสู่เมนูหลัก และหากต้องการออกจากโปรแกรมให้พิมพ์ตัวเลือกที่ 3

นิยาม / ศัพท์เฉพาะ

- State หมายถึง รูปแบบ ของ puzzle เช่น



Data Structure

โครงสร้างข้อมูล และตัวแปรประเภทต่างๆ ที่สำคัญของโปรแกรม

1. Class Grid

```
public class Grid {

    private int[][] state;
    private int value;
    private int size;
    private int[] str;

    Grid(int[][] a) {...7 lines }

    Grid(int v, int s) {...7 lines }

    Grid() {...4 lines }

    public void ValueToMatrix() {...12 lines }

    public void StateValue() {...14 lines }

    public void Printstate() {...16 lines }

    public int GetValue() {...3 lines }

    public int[][] GetState() {...3 lines }
```

Class Grid เป็น object ที่เปรียบเสมือน ตารางของแต่ละ state โดยจะเก็บค่าของ state ในรูปแบบเลขฐาน 10 (int Decimal_ID), ขนาดของตาราง (int size), state ในรูปแบบของArray 2 มิติ (int [ ][ ]state) และ state ในรูปแบบข้อความ (int str[ ])โดยมีจุดประสงค์เพื่อจัดการข้อมูลต่างๆ ของ state ในแต่ละ state

โดย Constructor ทั้ง 3 แบบ แตกต่างกันจากวิธีสร้าง object ดังนี้

1. Grid(int [ ][ ] a)

   Constructor ที่อาศัยการสร้าง object จาก Array 2D (State ในรูปแบบ Matrix)

2. Grid(int v, int s)

   Constructor ที่อาศัยการสร้าง object จาก ค่า Decimal_ID

3. Grid()

   Constructor ที่ไม่ต้องใช้ข้อมูลใดๆ ในการสร้าง object เพราะ เป็น Final Grid สำหรับการทำ BFS ในแต่ละชั้นๆ

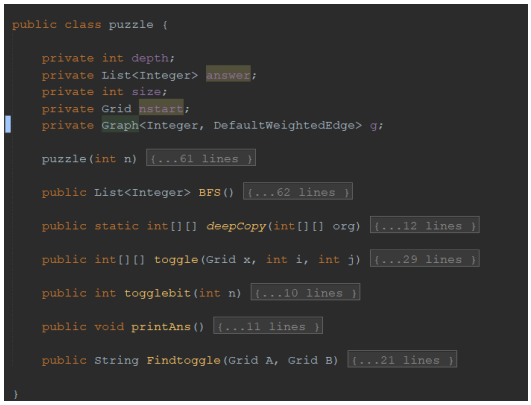
Method ต่าง ๆ

1. ValueToMatrix() สำหรับ แปลงค่า Decimal_ID ไปเป็น Array 2D
2. StateValue() สำหรับ แปลงค่า Array 2D ไปเป็น Decimal_ID
3. Printstate() สำหรับแสดงผลของ state ในรูปแบบ Matrix
4. GetValue(),GetState() getter

2. Class Mygraph

Class Mygraph เป็น object ที่เปรียบเสมือน Graph โดยจะมีการใช้ Graph API JGrapht โดยมี ตัวแปร Graph ประเภท undirected weight โดย Vertex มีค่าเป็น Integer และมี Method สำหรับหา shortest path ซึ่งใช้ DijkstraShortestPath Algorithm ในการค้นหา

3. Class puzzle



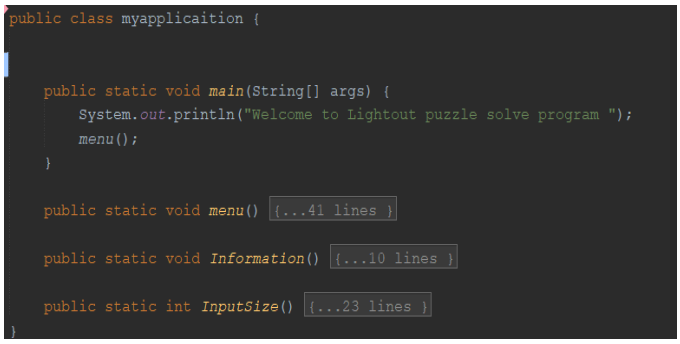
Class puzzle_ เป็น คลาสสำหรับแก้ปัญหา โดย depth จะเป็นระยะทาง จาก state ปัญหาของ user ไปหา คำตอบ , List<Integer> answer จะเก็บ node ของ solution ที่ได้จากการทำ DijkstraShortestPath, Grid nstart สำหรับ ตารางของปัญหาที่ได้รับจาก user และ Graph<Integer,DefaultWeightedEdge> สำหรับเก็บกราฟที่ได้จากการทำ BFS

โดย constructor puzzle(int n) จะรับค่าขนาดของตารางมาเพื่อสร้าง object รับค่า state เริ่มต้นจาก user ในรูปแบบของเลขฐาน 2 จากนั้นจึงนำไปใช้ BFS เพื่อสร้าง Graph จากนั้น จะเรียกใช้ method FindShortestPath ของ **class Mygraph** เพื่อค้นหา solution ของ state

Method ต่าง ๆ

1. BFS() สำหรับการทำ BFS เพื่อสร้าง Graph สำหรับ DijkstraShortestPath
2. deepCopy() สำหรับ การทำ Deep copy
3. toggle() สำหรับการทำ toggle เพื่อค้นหา state ใหม่
4. togglebit() สำหรับการสลับค่า 1 และ 0
5. printAns() สำหรับแสดงผล วิธีการแก้ปัญหา
6. Findtoggle(Grid A, Grid B) สำหรับค้นหา ว่าต้อง toggle Grid A ที่ตำแหน่งใด ถึงจะไปเป็น Grid B ได้

4. Class myapplicaition



Class myapplicaition__เป็นเมนคลาสสำหรับจัดการหน้า menu ต่างๆ

5. Array 2 มิติ ที่เก็บตัวแปรประเภท Integer

Array 2 มิตินี้เปรียบเสมือน ตารางขนาด N*N โดยตำแหน่งที่ ไฟเปิด จะเก็บค่า 1 และ ไฟปิดจะเก็บค่า 0

```
int[ ][ ] state =  new int [size][size]
```

1	1	0
0	0	1
1	1	0

แสดงตารางที่เก็บค่า ไฟเปิด/ปิด เป็น 1 และ 0

6. ArrayList

ถูกใช้ในขั้นตอนการทำ BFS (breadth first search) สำหรับเก็บค่า int Decimal_ID ที่เคยถูกเข้าถึงไปแล้ว เพื่อไม่ให้ ไปค้นหาที่ state ซ้ำ ตามหลักการของ BFS

```
ArrayList<Integer> checkpath = new ArrayList<Integer>();
```

ทุกครั้งที่มีการเริ่มต้นค้นหา state ใหม่ จะใช้ ArrayList เพื่อเช็คว่า มีค่าอยู่ใน ArrayList หรือไม่ ด้วย Method .**contains()** หากไม่มีค่าอยู่ก็ให้ทำการใช้ Method **.Add()** เพื่อทำการเพิ่มค่าใหม่ลงไป

7. ArrayDeque

ใช้ ArrayDeque เก็บตัวแปรประเภท Grid โดย Method สำคัญที่ใช้ได้แก่

- **pollFirst()** สำหรับดึงค่าแรกใน ArrayDeque ออกมาใช้ เพื่อเริ่มการทำ BFS ค้นหา state ใกล้เคียง

- **add()** เพิ่ม Grid ของ state ใหม่ที่ค้นหามา ลงไปยังคิวท้ายสุด ของ ArrayDeque เพื่อนำมาใช้สำหรับค้นหา state ใหม่ๆ

เหตุผลที่เลือกใช้ ArrayDeque เพราะ เราต้องการทำ Queue ดึงลำดับ ในการทำ BFS เป็น First in/ First out ดึงข้อมูลตัวแรกสุดของ Queue และ Add ข้อมูลที่ค้นพบล่าสุด ไปอยู่ในท้าย Queue

8. List

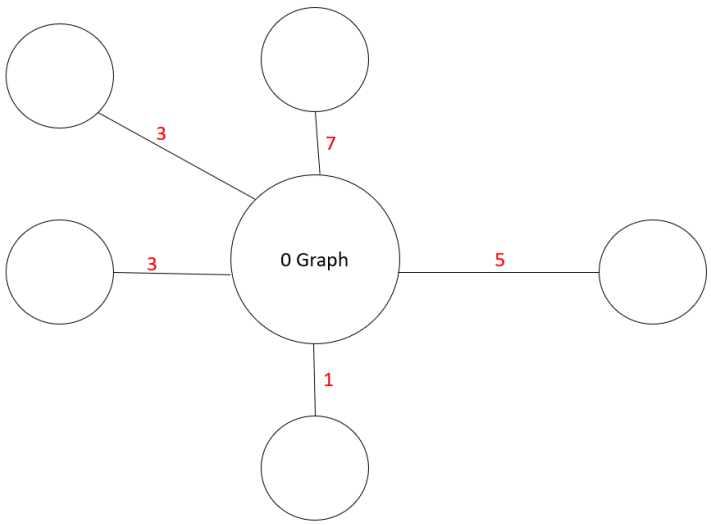
- เก็บตัวแปรประเภท int decimal_ID ที่ได้มาจาก DijkstraShortestPath เพื่อนำมาแสดง Solution ของ state เริ่มต้น

9. Graph

- เก็บกราฟประเภท Simple weight (Undirected weight Graph) โดย มี Vertex เป็น int decimal_ID ของ state แต่ละ state ที่ได้จาก BFS และ ทุกๆ Edge มีค่า weight เท่าๆ กัน โดยในโปรแกรมจะให้ ค่า weight เป็น 1 เสมอ

Brute Force Logic

การแก้ light out n*n แบบ Brute force โดยเริ่มจากสร้างจุดเริ่มต้น เป็น matrix ขนาด n×n ที่ทุกค่าเป็น 0 จากนั้นให้ไล่สร้างวิธีการกดไฟขึ้นมาเป็น matrix จาก matrix 0 ซึ่งใน1ปุ่มมีเลือก กดหรือไม่กด แทนตัวเลขด้วย 1 , 0 ตามลำดับ ทำให้มีทั้งหมด $2^{n^2}-1$ matrix จากนั้นให้สร้าง states ของ puzzle ขึ้นมาจากวิธีการคลิกของแต่ละ matrix ที่เราได้ไล่สร้างไว้ตอนแรก แล้วทำเป็น node graph โดยให้มีการเก็บข้อมูลเป็น states ของ puzzle วิธีการกดไฟเป็น matrix และระยะทางสู่จุดเริ่มต้นที่เป็น matrix 0 เป็น weight เกิดจากการนับจำนวนเลข 1 ในวิธีการกดไฟ



หน้าตาของกราฟแบบคร่าวๆ

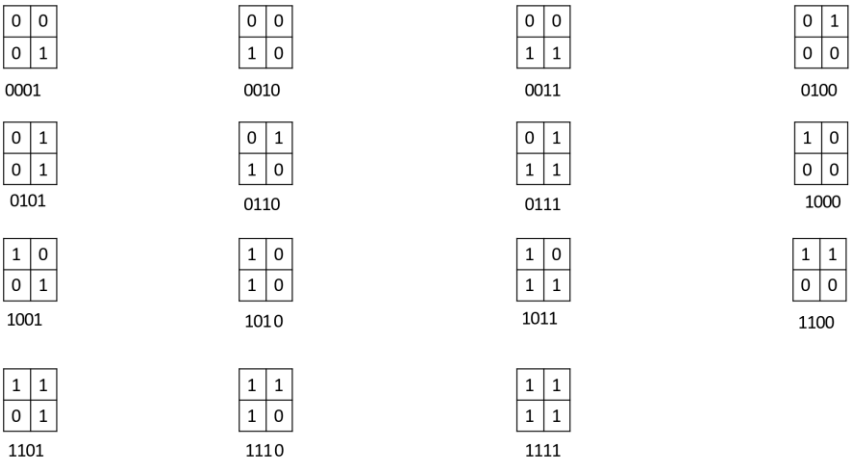
ในการแก้ puzzle จะให้ user ระบุขนาดรายละเอียดของ puzzle เข้ามาแล้วโปรแกรมจึงสร้างกราฟด้วยวิธีการข้างต้น จากนั้นโปรแกรมจะไล่หา node ที่มีรายละเอียดของ puzzle ที่เหมือนกันกับ input ของ user จากนั้นโปรแกรมก้จะบอกวิธีการแก้ puzzle ถ้ามีหลายวิธีโปรแกรมจะแสดงวิธีที่ใช้จำนวนคลิกน้อยที่สุดก่อนเสมอ ในกรณีที่ไม่มี node ใดตรงกับ input โปรแกรมจะบอกว่า input puzzle นั้นไม่มีวิธีการแก้

Showcase 2x2 light out puzzle

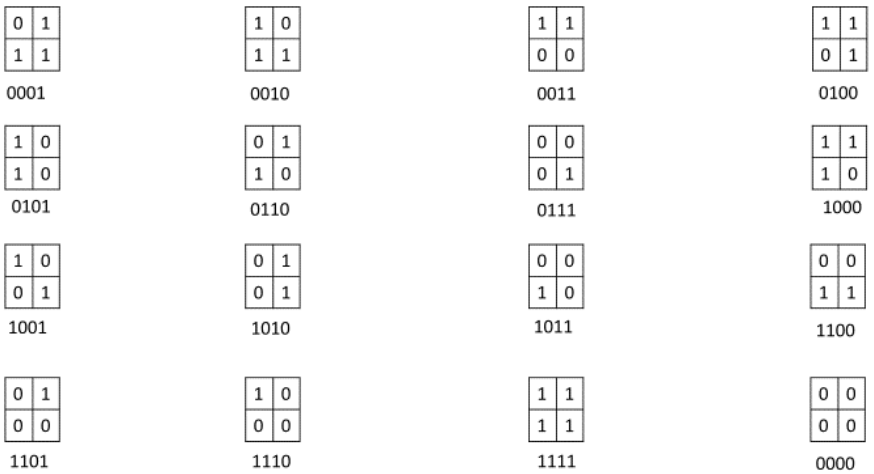
เริ่มต้นด้วยการสร้างเมทริกซ์ 0

0	0
0	0

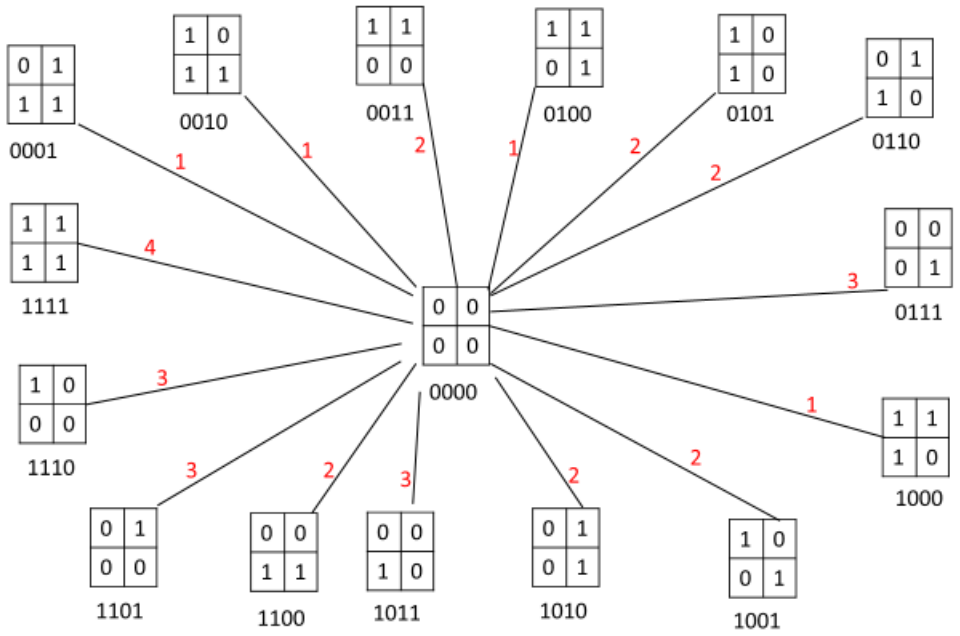
สร้างทุกรูปแบบการคลิกที่เป็นไปได้ (กรณีตารางขนาด 2×2 จะมีคลิกที่เป็นไปได้ = $2^4 - 1 = 15$ แบบ)



สร้าง states puzzle โดยใช้วิธีการกดจากแต่ละ matrix ข้างบน



สร้างกราฟจาก states



จากวิธีการแก้ปัญหานี้ asymptotic runtime เป็น $O(2^{n^2})$

Algorithm

BFS (breadth-first search)

BFS (breadth-first search) เป็นการท่องกราฟรูปแบบหนึ่ง โดยให้ค้นหาแบบแนวกว้างไปตามระดับจากจุดเริ่มต้น จนพบคำตอบ หากค้นพบรูปแบบที่ซ้ำ ก็จะไม่ไปค้นหาบริเวณนั้นอีก

โดย ในที่นี้ได้นำมาแก้ปัญหาของ light out puzzle เริ่มจาก state เริ่มต้นที่เรารับค่ามา ทำการ toggle ทุกจุดจาก state เริ่มต้น จากนั้น เพิ่ม state เข้าแถวคิว เพื่อรอทำการค้นหาในอีกระดับนึง หากเป็นค่าที่เคยค้นพบแล้ว ก็จะไม่ถูกเพิ่มลงคิว และเมื่อค้นหาจบครบในระดับนี้แล้ว ก็ให้ทำการค้นหาในระดับต่อไปจนกว่าจะเจอคำตอบ แล้วนำ state ที่ได้จากการ toggle ทั้งหมด ไปสร้าง Graph โดยจะมี Runtime อยู่ที่ O(|V| + |E|) โดย V, E เป็นจำนวน Vertex และ Edge ทั้งหมด

ตัวอย่างโค้ดในส่วนของ **BFS**

```
public Graph<Integer, DefaultWeightedEdge>  BFS() {
    Grid presentGrid;
    int Decimal_value;
    ArrayDeque<Grid> Q = new ArrayDeque<>();    // สร้าง Queqe สำหรับ
    ArrayList<Integer> checkpath = new ArrayList<Integer>(); // สร้าง ArrayList สำหรับเช็ค Grid ที่ซ้ำ
    Grid FinalGrid = new Grid();
    Q.add(nstart);   // เพิ่ม Grid เริ่มต้นลงไปใน Queqe
    Q.add(FinalGrid); // เพิ่ม Grid สุดท้ายเพื่อเช็คในแต่ละระดับ

    System.out.println("Please wait a minute some states have more time to find solution...");
    while (!Q.isEmpty()) {  // Runtime O(|V|)
        presentGrid = Q.pollFirst();   // ดึงค่าแรกจาก Queqe
        Decimal_value= presentGrid.GetValue(); // เปลี่ยนเป็นค่า ฐาน 10 (Decimal)
        if (depth > (size * size)) {
            System.out.println("This state is no solution");
            System.exit(0);
        } else if (Decimal_value == -1) {
            depth++;
            Q.add(presentGrid);
        } else if (Decimal_value == 0) {

            System.out.println(depth + " moves to turn off all light");
            if (depth == 0) {
                System.out.println("Initial states has turn off all light");
                System.exit(0);
            }
            System.out.println("\nStart");
            break;
        } else {
```

```
        } else {
            for (int i = 0; i < size; i++) {   // Runtime O(|E|)

                for (int j = 0; j < size; j++) {
                    int[][] st = toggle(presentGrid, i, j);  //toggle state ใหม่ , ค้นหาเส้นทางใหม่
                    Grid nextGrid = new Grid(st);
                    int temp = nextGrid .GetValue();
                    if (!checkpath.contains(temp)) { //state ที่ไม่เคยผ่าน
                        checkpath.add(temp);  //เพิ่มค่าที่เคยผ่านมา
                        Graphs.addEdgeWithVertices(g, Decimal_value, temp, 1); // เพิ่มไปในกราฟ
                        Q.add(nextGrid);  // เพิ่มลงที่คิว
                    }
                }
            }

        }

    }
```

Dijkstra's algorithm / DijkstraShortestPath

Dijkstra's algorithm / DijkstraShortestPath เป็น Algorithm สำหรับค้นหาเส้นทางที่สั้นที่สุด จาก จุดเริ่มต้น / ไปยังจุดเป้าหมาย โดย Graph นั้นจะต้องเป็น non – negative cycle weight

โดยในโปรแกรมนี้ เราใช้ JGrapht เพื่อสร้าง Object Algorithm สำหรับ DijkstraShortestPath เหตุผลที่เราเลือกใช้ Algorithm นี้ เพราะ Graph ที่ได้จากการทำ BFS นั้นเป็น Undirected Graph ที่ weight ทุกๆ Edge มีค่าเป็น 1 เสมอ จึงไม่มี negative cycle weight จึงเป็นเหตุผลที่เลือกใช้ **Dijkstra's algorithm** โดยจะมี Runtimeอยู่ที่ $O(|V|^2+|V|)$ เนื่องจากกราฟมีลักษณะเป็น spare graph จึงสรุปได้ว่า Runtime ของ DijkstraShortestPath จะขึ้นอยู่กับจำนวน Vertex ที่ได้จาก BFS

ตัวอย่างโค้ดในส่วนของ DijkstraShortestPath

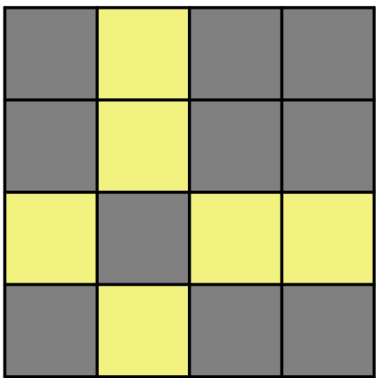
```
public List<Integer> DijkstraShortestPath(int start ,int Goal)
{

    if (G.containsVertex(start) && G.containsVertex(Goal))
    {
        ShortestPathAlgorithm<Integer, DefaultWeightedEdge> shpath = null;


        try
        {
            shpath = new DijkstraShortestPath<>(G);  //สร้าง obj Algorithm ของ Dijkstra
            shpath.getPath(start, Goal);
        }
        catch(IllegalArgumentException e) { System.out.println(e); }


        if (shpath.getPath(start, Goal) != null)
        {

            List<Integer> allNodes = shpath.getPath(start, Goal).getVertexList();
                                                //ดึงค่าทั้งหมดที่ได้จาก Dijkstra
            return allNodes;
        }
        else
            System.out.printf("\nPath from %s to %s doesn't exist\n", start, Goal);
    }
    return null;
}
    public Graph<Integer, DefaultWeightedEdge> getG(){
    return G;
}
```

วิธีนี้เป็นวิธีการแก้ปัญหาที่ได้จำนวน move ที่น้อยที่สุดหรือไม่ ?

เนื่องจากการใช้ Dijkstra's algorithm / DijkstraShortestPath เป็นการค้นหา เส้นทางที่สั้นที่สุดจากจุดเริ่มต้น(state ของปัญหา) ไปยังปลายทางอยู่แล้ว ดังนั้น วิธีนี้จึงยืนยันได้ว่า วิธีนี้ เป็น วิธีแก้ปัญหาที่สั้นที่สุด

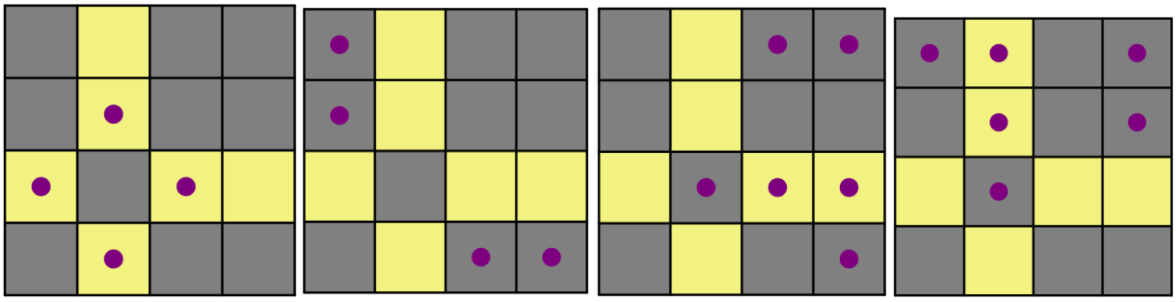
No solution

กรณีที่ไม่มีคำตอบเกิดจากการที่ 1 puzzle มีหลาย solution ทำให้ บาง states ไม่สามารถเกิดขึ้นจริงได้ยกตัวอย่างเช่น

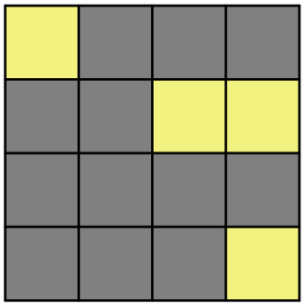


ให้สีเทาเป็นไฟปิด สีเหลืองเป็นไฟเปิด

Puzzle นี้พบว่ามีมากกว่า 1 solution โดยสีเทาเป็นไฟปิด สีเหลืองเป็นไฟเปิด จุดสีม่วงเป็นจุดทั้งหมดที่ต้องกดเพื่อแก้ puzzle



กรณีตัวอย่างนี้พบว่ามีอย่างน้อย 4 solution จึงทำให้บาง state ที่ user input เข้ามาไม่พบ solution เช่น

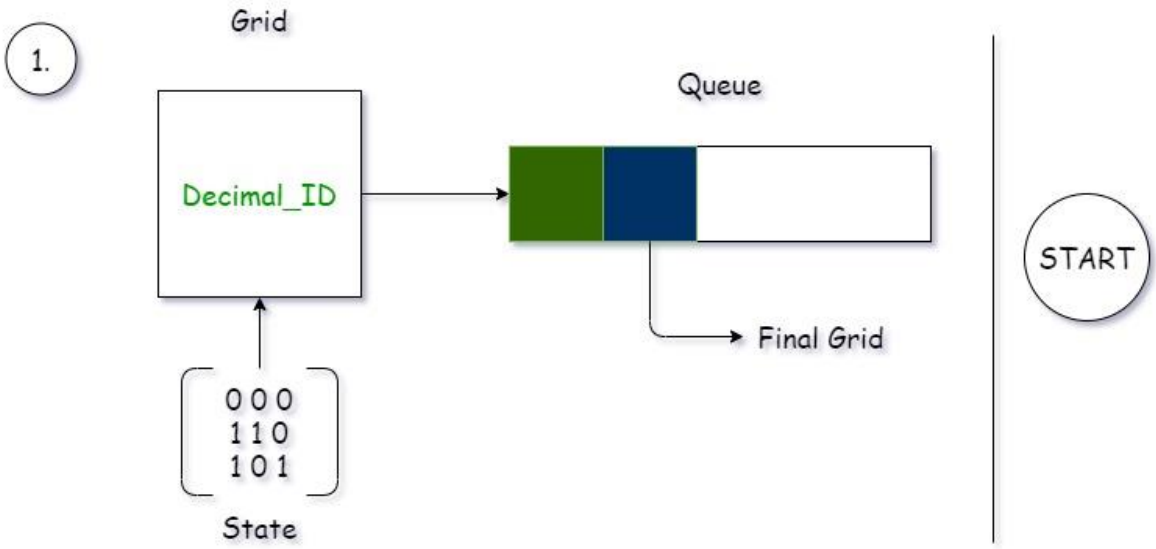


Puzzle นี้จากวิธี Brute force พบว่าในการคลิกทุกรูปแบบไม่พบ state ที่เหมือน puzzle จึงสรุปว่าไม่มี solution

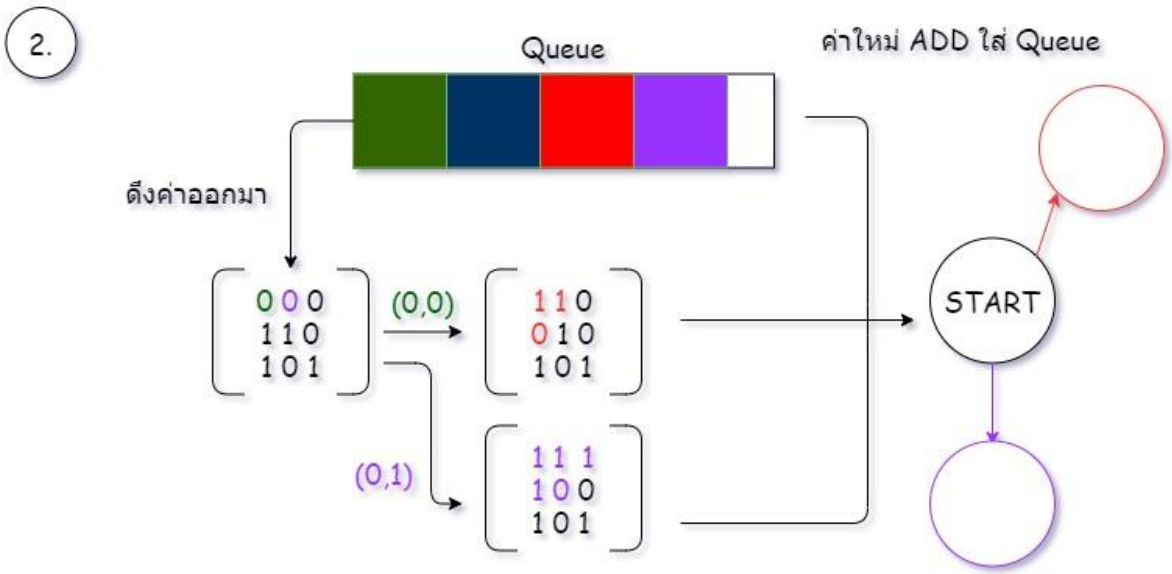
Example case : 000110101

Step ในส่วนของ BFS

1. เริ่มต้นทำ BFS โดยให้ initial state เก็บในรูปแบบ Grid จากนั้น นำลงไปใส่ใน ArrayDeque (Queue)

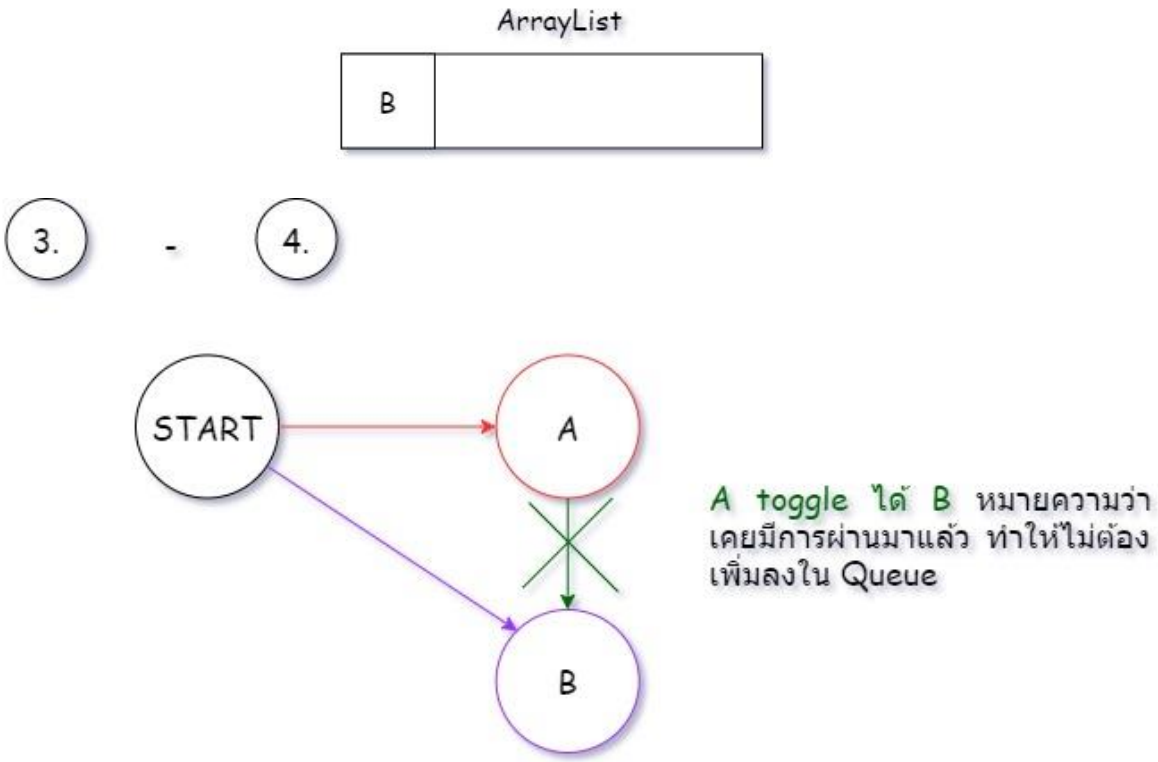


2. ดึงค่าแรกจาก Queue ทำการ toggle state ทีละตำแหน่ง (0,0) , (0,1) , (0,2) . . .(2,2)

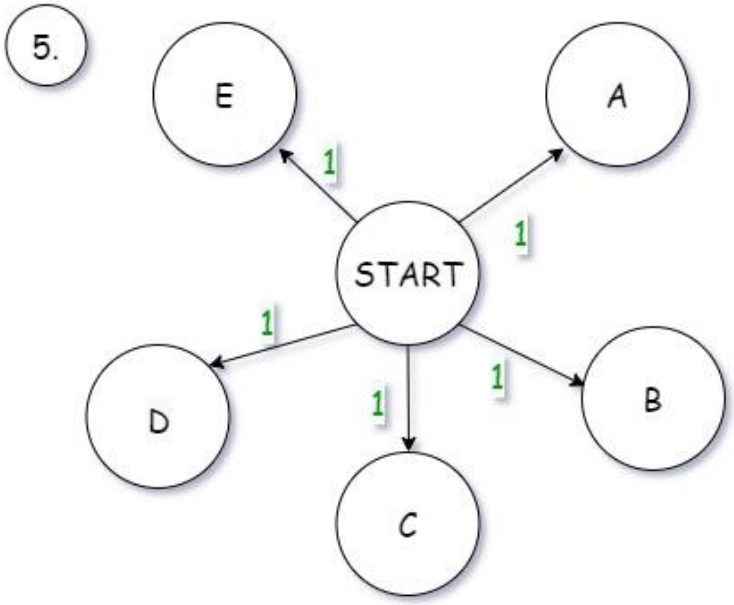


3. หลังจากที่เราทำการ toggle เสร็จในแต่ละรอบ เช็คว่า state ใหม่ที่ toggle มานั้น เป็น state ที่เคยผ่านมาแล้วหรือไม่ โดยใช้ ArrayList check path ในการจำค่าของ state ใหม่ๆ

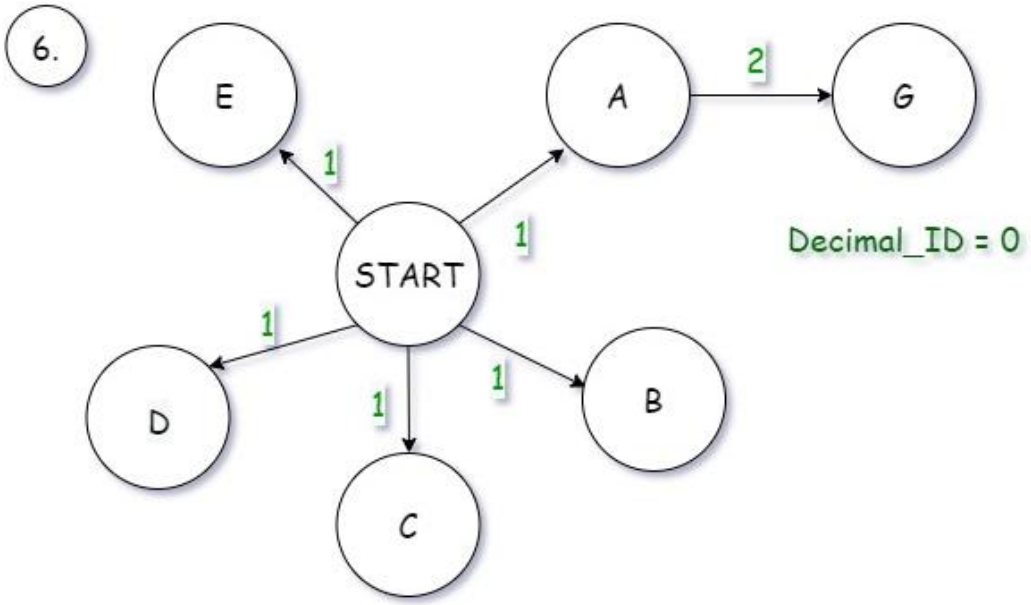
4. หากเป็น state ที่ไม่ซ้ำ ให้เพิ่มลงใน Queue และ สร้าง Graph สำหรับเส้นทางใหม่ แต่หากเป็น state เก่า ที่เป็นเส้นทางที่ซ้ำ ให้เราเพิกเฉย ไม่ต้องเพิ่มลง Queue



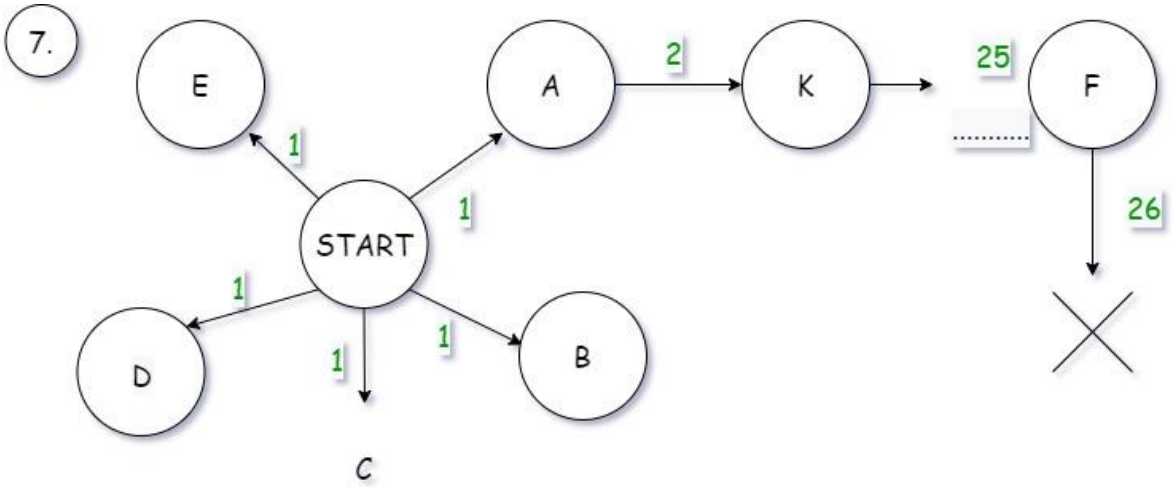
5. เมื่อถึงจุดสิ้นสุดในระดับแล้ว ให้นับระดับความลึกเพิ่มขึ้น 1 แล้ว ทำซ้ำวนข้อ 2 – 4

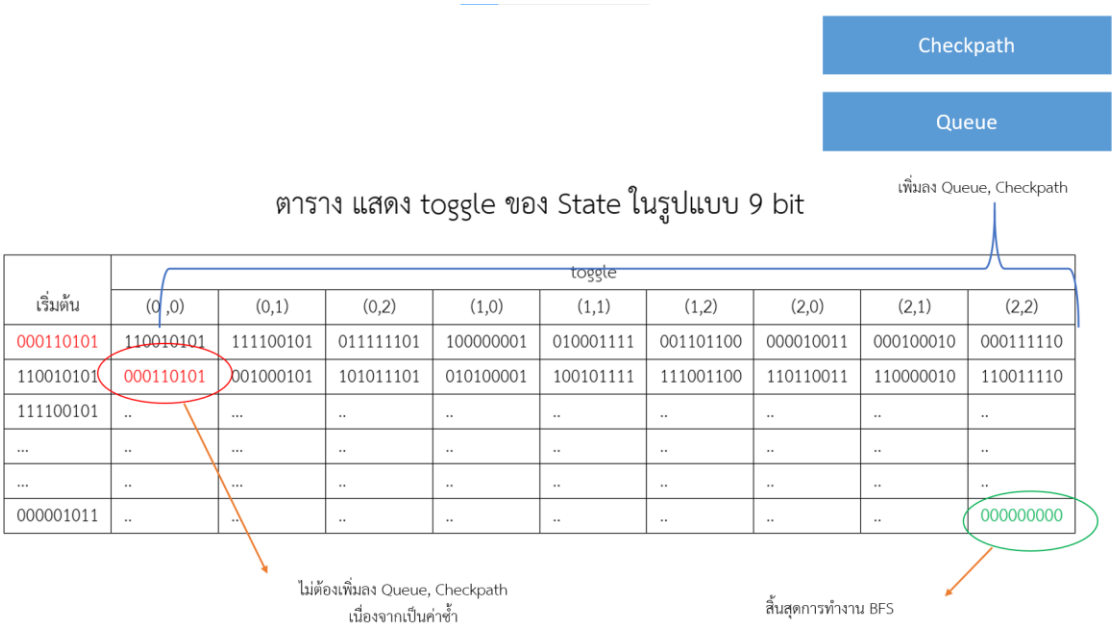


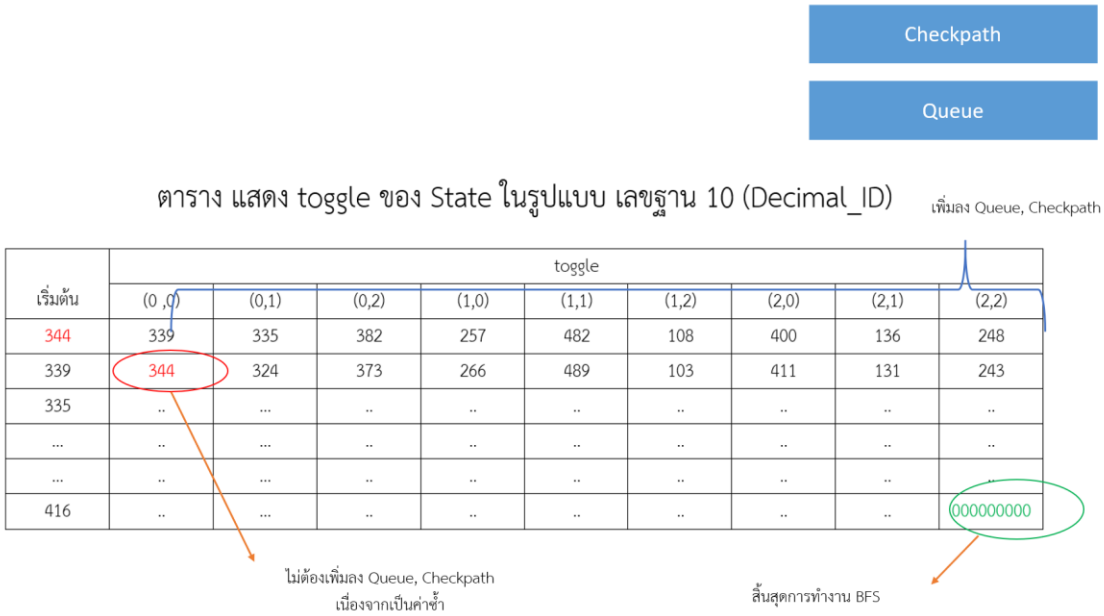
6. หากพบเจอ state ที่มีค่า Decimal_ID = 0 แสดงว่าเจอเป้าหมายแล้ว ให้จบการทำงานของ BFS แล้วทำ DijkstraShortestPath ต่อจาก Graph ที่เราสร้างไว้



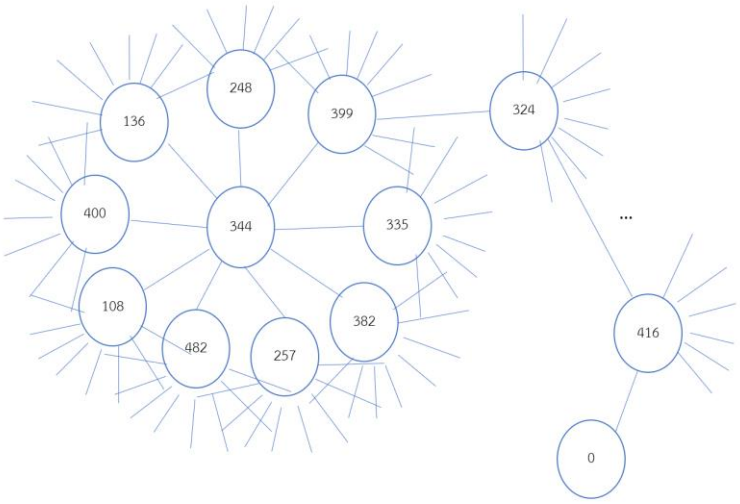
7. หากระดับความลึก มากกว่า N×N แสดงว่า state ของปัญหานี้ไม่มี Solution เนื่องจากไม่มีทางที่ BFS จะทำระดับความลึกมากกว่า N×N ได้




Checkpath
Queue
ตาราง แสดง toggle ของ State ในรูปแบบ 9 bit
เพิ่มลง Queue, Checkpath
toggle
เริ่มต้น
(0 ,0)
(0,1)
(0,2)
(1,0)
(1,1)
(1,2)
(2,0)
(2,1)
(2,2)
000110101 110010101 111100101 011111101 100000001 010001111 001101100 000010011 000100010 000111110
110010101 000110101 001000101 101011101 010100001 100101111 111001100 110110011 110000010 110011110
111100101 .. ..
... .
... .
000001011 000000000
ไม่ต้องเพิ่มลง Queue, Checkpath
เนื่องจากเป็นค่าซ้ำ
สิ้นสุดการทำงาน BFS


Checkpath
Queue
ตาราง แสดง toggle ของ State ในรูปแบบ เลขฐาน 10 (Decimal_ID)
เพิ่มลง Queue, Checkpath
toggle
เริ่มต้น
(0 ,0)
(0,1)
(0,2)
(1,0)
(1,1)
(1,2)
(2,0)
(2,1)
(2,2)
344 339 335 382 257 482 108 400 136 248
339 344 324 373 266 489 103 411 131 243
335 .. ..
... .
... .
416 000000000
ไม่ต้องเพิ่มลง Queue, Checkpath
เนื่องจากเป็นค่าซ้ำ
สิ้นสุดการทำงาน BFS

ตัวอย่าง กราฟที่สร้างได้ จาก BFS



Step ในส่วนของ Dijkstra shortest path

1. ในส่วนของกราฟ ค่าของ Node เป็น Decimal_ID จึงให้ Node เริ่มต้น = S เป็น 344 และ Node target เป็น 0

2. ให้ Dsv เป็นเส้นทางที่สั้นที่สุดของ Node S กับ Node V รอบๆ โดย Dsv เริ่มต้นเป็น inf (ยกเว้น Dss - > ระหว่างทางระหว่าง S ไป S = 0)

3. ค้นหา เส้นทางที่สั้นที่สุดระหว่าง Node S กับ Node รอบๆ หากเจอเส้นทางที่สั้นกว่า ให้ update ค่า Dsv ใหม่

4. ทำซ้ำ ไปเรื่อยๆ จนกว่า ทุกๆ node ของ กราฟจะถูก Visited

Dijkstra's Shortest path

Node	Visited	Distance	Pre_Node
344	0	0	0
339	0	inf	0
335	0	inf	0
382	0	inf	0
..	0	inf	0
416	0	inf	0
Goal	0	inf	0

เริ่มที่ start node 344 ปรับค่า Distance ใหม่ หาก Distance ใหม่ < Distance เก่า ให้อัพเดตค่าไปเรื่อยๆ

Node	Visited	Distance	Pre_Node
344	0	0	0
339	0	1	344
335	0	1	344
382	0	1	344
257	0	1	344
..	0	inf	0
324	0	inf	0
373	0	inf	0
266	0	inf	0
..	0	inf	0
..	0	inf	0
416	0	inf	0
Goal	0	inf	0

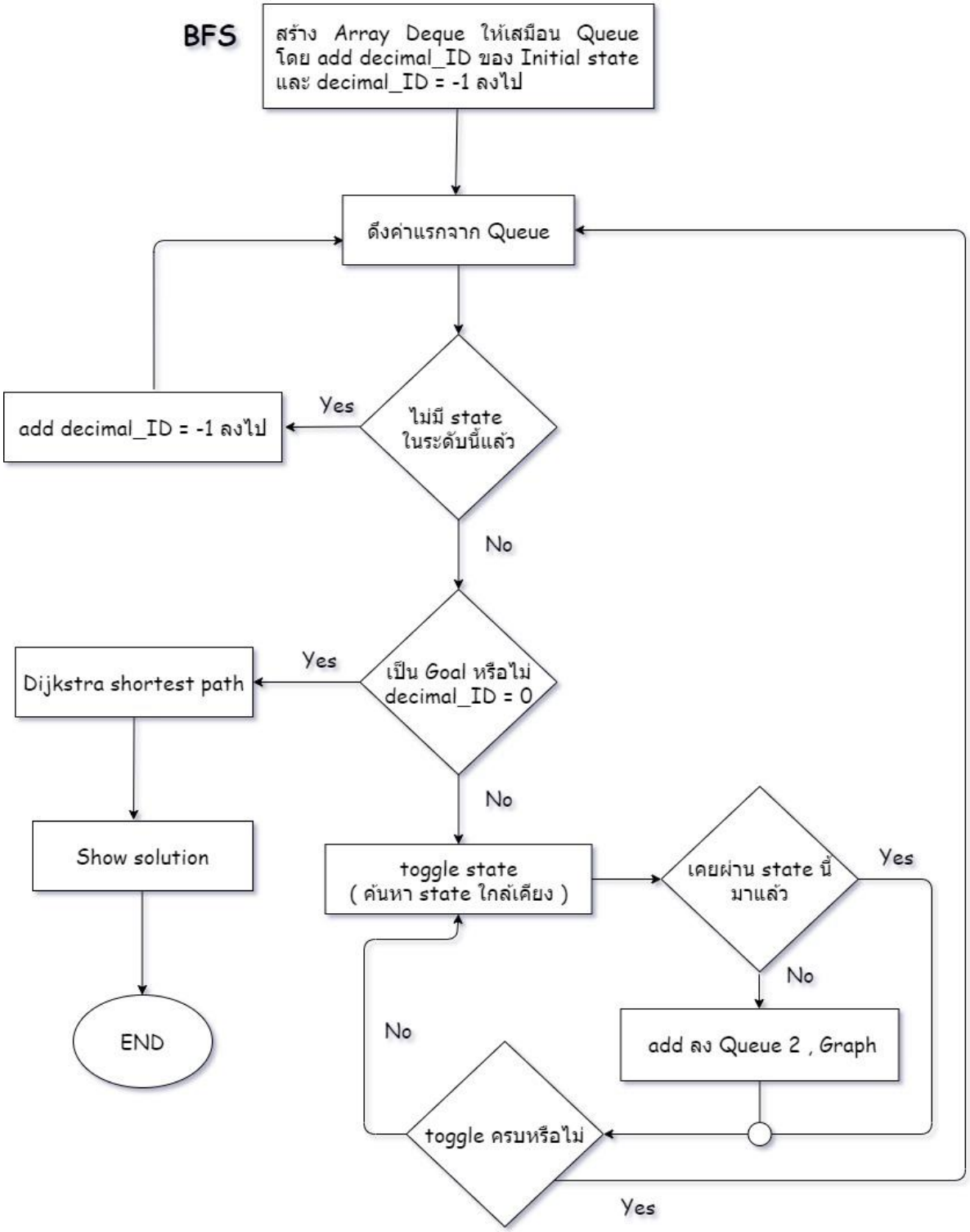
Node ที่เคยผ่านมาแล้ว ให้Mark ค่าว่า Visited ไปแล้ว และเริ่มทำ Node ต่อไป

Node	Visited	Distance	Pre_Node
344	1	0	0
339	0	1	344
335	0	1	344
382	0	1	344
257	0	1	344
..	0	inf	0
324	0	1	339
373	0	1	339
266	0	1	339
..	0	inf	0
..	0	inf	0
416	0	inf	0
Goal	0	inf	0

ทำจนครบทุก Node ในกราฟ จึงเสร็จวิธีการของ Dijkstra’s Shortest path

Node	Visited	Distance	Pre_Node
344	1	0	0
339	1	1	344
335	1	1	344
382	1	1	344
257	1	1	344
..	0	-	-
324	1	1	339
373	1	1	339
266	1	1	339
..	1	-	-
..	1	-	-
416	1	4	112
Goal	1	5	416

Flowchart


BFS
สร้าง Array Deque ให้เสมือน Queue โดย add decimal_ID ของ Initial state และ decimal_ID = -1 ลงไป
ดึงค่าแรกจาก Queue
ไม่มี state ในระดับนี้แล้ว
Yes
add decimal_ID = -1 ลงไป
No
เป็น Goal หรือไม่ decimal_ID = 0
Yes
Dijkstra shortest path
Show solution
END
No
toggle state (ค้นหา state ใกล้เคียง)
เคยผ่าน state นี้มาแล้ว
Yes
No
add ลง Queue 2 , Graph
toggle ครบหรือไม่
No
Yes

Compare runtime

วิธีการปัญหาแบบ BFS + Dijkstra ของกลุ่มเรานั้นใช้วิธีการสร้าง Node ไปจนหาวิธีการแก้ปัญหาเจอแล้วจนหยุด โดย Runtime ของ Dijkstra จะขึ้นอยู่กับ จำนวน Node จาก BFS ในกรณีที่ดีที่สุดคือ 1 ครั้ง ในกรณีที่แย่ที่สุดคือ 2^{n^2} ครั้งในปัญหาขนาด n^2 ส่วนวิธี Brute force นั้นทุกปัญหาสร้าง Node ขึ้นมา 2^{n^2} ครั้งเพราะวิธี Brute force ต้องสร้างทุก Node ที่เป็นไปได้ก่อนแล้วจึงหาวิธีการแก้ที่ตรงกับปัญหาที่ใส่เข้ามา

สรุป: ในกรณีส่วนใหญ่ BFS + Dijkstra ใช้ runtime น้อยกว่า Brute force (2^{n^2}) แต่ในกรณีที่ต้องสร้างทุก Node วิธีการแก้ปัญหาของทั้ง 2 หลักการนี้จะใช้ runtime เท่ากัน O(2^{n^2})

ข้อจำกัดของโปรแกรม

1. ตั้งแต่ 5 ขึ้นไป จำนวน state ที่ต้องการค้นหาจะมีเยอะมากตาม $2^{n \times n}$ ดังนั้น จึงอาจต้องรอโปรแกรมค้นหาวิธีแก้ปัญหาที่นานมาก

2. เนื่องจากการใช้ int เก็บค่าของ state ดังนั้น ยิ่งจำนวนตารางมาก ทำให้ int ไม่สามารถรองรับค่าของ state นั้นได้ (เกิน 2^{31}) จึงเป็นเหตุผลที่ไม่สามารถทำการค้นหา solution ต้งแต่ ตาราง 6×6 เป็นต้นไป

3. บางปัญหาอาจไม่มี solution โปรแกรมจึงไม่สามารถค้นหาวิธีแก้ปัญหานั้นได้

บรรณานุกรม

What is the algorithm for solving lights out puzzle in minimum number of moves in Java?. 2554. (ออนไลน์), สืบค้นเมื่อ 15 เมษายน 2564 จาก

https://www.quora.com/What-is-the-algorithm-for-solving-lights-out-puzzle-in-minimum-number-of-moves-in-Java

Lights Out - Lights On algorithm. 2558. (ออนไลน์), สืบค้นเมื่อ 15 มีนาคม 2564 จาก

https://www.reddit.com/r/algorithms/comments/2lx85u/lights_out_lights_on_algorithm/

Python , 8 Puzzle และ A*. 2563. (ออนไลน์), สืบค้นเมื่อ 20 มีนาคม 2564 จาก

https://pawutjingjit.medium.com/python-8-puzzle-%E0%B9%81%E0%B8%A5%E0%B8%B0-a-5d3535c0b153

Lights Out Best-First Search/A* Algorithm. 2563. (ออนไลน์), สืบค้นเมื่อ 20 มีนาคม 2564 จาก

https://stackoverflow.com/questions/60157747/lights-out-best-first-search-a-algorithm

daattali/lightsout. 2559. (ออนไลน์), สืบค้นเมื่อ 20 มีนาคม 2564 จาก

https://github.com/daattali/lightsout

Light out puzzle. 2559. (ออนไลน์), สืบค้นเมื่อ 17 มีนาคม 2564 จาก

https://www.geogebra.org/m/JexnDJpt