รายงานประจำวิชา

การออกแบบและวิเคราะห์ขั้นตอนวิธี (Algorithm Design and Analysis)

รหัสวิชา 01418232 หมู่เรียน 870

เรื่อง

Deep First Search Algorithm

กลุ่ม Chill Guys

จัดทำโดย

นาย ปัณณวัฒน์ นิ่งเจริญ	6630250231
นาย พันธุ์ธัช สุวรรณวัฒนะ	6630250281
นาย ปุณณภพ มีฤทธิ์	6630250291
นาย วรินทร์ สายปัญญา	6630250435
นางสาว อัมพุชินิ บุญรักษ์	6630250532

เสนอ

อาจารย์ อานนท์ ผ่องรัศมีเพ็ญ

คณะวิทยาศาสตร ศรีราชา สาขาวิทยาการคอมพิวเตอร์

มหาวิทยาลัยเกษตรศาสตร์ วิทยาเขตศรีราชา

กำหนดปัญหา

ปัญหา

สมมุติว่าเราต้องการวางแผนการเดินทางไปยังหลายจุดหมายที่แตกต่างกัน ซึ่งสถานที่ต่างๆ เหล่านี้มีเส้นทางเชื่อมต่อกัน อาจจะมีการเลือกเส้นทางที่ต้องเดินทางไปเยี่ยมชมหลายแห่งในแต่ละครั้ง ทั้งที่บางเส้นทางอาจยาวและบางเส้นทางอาจสั้นกว่า

ในสถานการณ์เช่นนี้ เราต้องการหาวิธีการเลือกเส้นทางที่ **สามารถเดินไปลึกสุด** ก่อน (คือไปถึงจุดหมายที่เราตั้งใจจะไปให้เร็วที่สุดในเส้นทางนั้น) แล้วค่อยย้อนกลับมาและลองเส้นทางอื่นหากเส้นทางแรกไม่สำเร็จหรือถึงทางตัน ซึ่งตรงกับแนวคิดการใช้ **DFS (Depth-First Search)** ในการค้นหาผลลัพธ์ โดยเลือกเดินทางไปในเส้นทางที่ลึกที่สุดก่อนเสมอ

แก้ปัญหาด้วย algorithm

การใช้ DFS (Depth-First Search) ในการแก้ปัญหาการวางแผนเที่ยวหลายสถานที่แบบลึกก่อน สามารถอธิบายได้ว่ามันเป็นการสำรวจเส้นทางจากจุดเริ่มต้นไปยังจุดหมาย โดยเลือกเส้นทางที่ลึกที่สุดก่อนและพยายามเดินทางไปให้สุดทางจนไม่สามารถไปต่อได้ จากนั้นหากพบทางตันจะย้อนกลับไปยังจุดที่เคยไปแล้วเพื่อสำรวจเส้นทางอื่น ๆ ที่ยังไม่เคยไป จนกว่าจะพบเส้นทางที่สามารถไปถึงจุดหมายได้หรือไม่สามารถหาทางไปได้เลย

ในกรณีของการท่องเที่ยวหลายจุดหมาย, สมมติว่าเราต้องการเดินทางจากจุดเริ่มต้น (เช่น เมือง A) ไปยังจุดหมาย (เช่น เมือง E) และในระหว่างทางนั้นมีหลายเส้นทางที่เชื่อมโยงไปยังเมืองต่าง ๆ เช่น เมือง B, เมือง C, และเมือง D โดยแต่ละเมืองอาจมีเส้นทางที่เชื่อมโยงกับเมืองอื่น ๆ อีกด้วย

เริ่มจากที่เราเลือกจุดเริ่มต้น เช่น เมือง A และจากนั้นเราจะพิจารณาเส้นทางที่สามารถเดินไปต่อได้ ถ้าสมมติว่าเมือง A เชื่อมโยงกับเมือง B และ C, เราจะเลือกเส้นทางไปยังเมือง B ก่อนเพราะตามแนวคิดของ DFS เราจะเลือกเส้นทางที่ลึกที่สุดก่อน (ในกรณีนี้หมายถึงการไปเมือง B ก่อน) เมื่อไปถึงเมือง B แล้ว เราจะตรวจสอบว่ามีเส้นทางที่ยังไม่เคยไปหรือไม่ ซึ่งอาจเป็นเมือง D ในกรณีนี้

หากจากเมือง B ไปยังเมือง D เราจะเดินทางต่อไปจนไม่สามารถไปต่อได้ เช่น หากเมือง D ไม่มีเส้นทางต่อไปถึงเมือง E, ระบบจะย้อนกลับมายังเมือง B และลองเส้นทางอื่นที่ยังไม่เคยไป ซึ่งในกรณีนี้คือเมือง C หรือเมือง E ถ้าเลือกเส้นทางจาก B ไป E และพบว่ามีเส้นทางตรงไปยังเมือง E ก็จะถือว่าพบเส้นทางไปถึงจุดหมายแล้ว

แต่หากเส้นทางที่เราพยายามเลือกไม่มีเส้นทางไปถึงเมือง E เลย ก็จะต้องย้อนกลับไปหลาย ๆ ครั้งจนกว่าเราจะตรวจสอบทุกเส้นทางที่มีในกราฟจนหมด หากพบเส้นทางที่ไปถึงเมือง E ได้ เราก็จะแสดงเส้นทางที่ค้นพบออกมา เช่น A ⟶ B ⟶ D ⟶ E แต่ถ้าหากไม่พบเส้นทางเลย ก็จะได้ผลลัพธ์ว่าไม่สามารถไปถึงจุดหมายได้

หลักการของ DFS คือการสำรวจเส้นทางลึกสุดก่อน โดยไม่สนใจเส้นทางที่ตื้นกว่า ตราบใดที่ยังไม่พบทางตันหรือจุดหมาย จึงจะย้อนกลับไปสำรวจเส้นทางใหม่ที่ยังไม่ได้ไป การใช้ DFS จึงเหมาะกับการหาทางลึก ๆ ในกราฟที่มีเส้นทางซับซ้อนหรือมีหลายทางเลือก ที่อาจต้องใช้เวลาในการย้อนกลับหลายครั้งเพื่อหาทางที่ดีที่สุด

สุดท้าย การเดินทางที่ใช้ DFS จะคำนึงถึงการตรวจสอบทุกเส้นทางจนสุดความสามารถก่อนที่จะสรุปว่าพบเส้นทางหรือไม่ ซึ่งเป็นลักษณะของการหาทางในกราฟที่ต้องการผลลัพธ์ที่สามารถเดินทางไปถึงจุดหมายได้จริงหรือไม่

โค้ดของ Algorithm

main.py

src > main.py

```python
import graphs as all_graphs


# function สำหรับการเช็ค vertices ใน graph ว่าถ้ามีจะ return True ไม่มีจะ return False
def has_vertices(graph, start, goal):
    # เก็บเป็น set เพื่อให้ไม่มี vertices ที่มีค่าซ้ำกัน
    vertices = set()

    # วนลูปรอบแรกของ graph ให้ดึง key และ value ใน dict ออกมา
    for key, values in graph.items():
        # เพิ่มค่า key ลงใน all_vertices
        vertices.add(key)
        # วนลูปรอบที่สองของ values เพราะ value ที่ได้มาตอนแรกใน dict เป็น list
        # ต้องวน loop อีกรอบเพื่อได้ค่า element
        for value in values:
            # เพิ่มค่า value
            vertices.add(value)

    # เขียน condition return ออกมาว่าถ้า start และ goal อยู่ใน all_vertices
    # ให้ return True ไม่อยู่ให้ return False
    return start in vertices and goal in vertices

```

```python
# Algorithm: Deep-Firth-Search
def dfs(graph, start, goal, path=None):
    # เช็คว่าถ้าค่า path เป็น None ให้ทำเงื่อนไขต่อไปนี้ (สำหรับรัน algorithm นี้ในครั้งแรก)
    if path is None:
        # กำหนดค่า path ให้เป็น list      You, 50 seconds ago • Uncommitted changes
        path = []
        # เช็ค condition นี้ว่าถ้า vertices ที่รับค่ามาจาก paramters ไม่มีอยู่ใน graph ให้ throw exception นี้ออกไป
        if not has_vertices(graph, start, goal):
            raise Exception(f"ไม่มี vertices {start} หรือ {goal} ที่อยู่ใน graph!")

    # เพิ่มค่า start ลงใน path
    path.append(start)

    # ถ้าพบเส้นทางเป้าหมายแล้วให้ return ค่า path ออกมา
    if start == goal:
        return path
```

```python
    # เช็คว่าถ้าค่า path เป็น None ให้ทำเงื่อนไขต่อไปนี้ (สำหรับรัน algorithm นี้ในครั้งแรก)
    if path is None:
        # กำหนดค่า path ให้เป็น list      You, 50 seconds ago • Uncommitted changes
        path = []
        # เช็ค condition นี้ว่าถ้า vertices ที่รับค่ามาจาก paramters ไม่มีอยู่ใน graph ให้ throw exception นี้ออกไป
        if not has_vertices(graph, start, goal):
            raise Exception(f"ไม่มี vertices {start} หรือ {goal} ที่อยู่ใน graph!")

    # เพิ่มค่า start ลงใน path
    path.append(start)

    # ถ้าพบเส้นทางเป้าหมายแล้วให้ return ค่า path ออกมา
    if start == goal:
        return path

    # วน loop หาเส้นทางของ goal เมื่อวน loop ค่าของตัวแปร neighbor จะเป็นค่า value ของ dict
    for neighbor in graph[start]:
        # ถ้าค่า neighbor ไม่อยู่ใน path ให้เรียกใช้ dfs ตัวมันเองอีกครั้ง(recursive)
        # และ หลีกเลี่ยงปัญหาเกิดการเก็บค่าเดิมของ path ที่เราเคยเดินมาแล้ว
        if neighbor not in path:
            # ส่ง arguments ไปใหม่เปลี่ยนจาก start → neightbor ที่เหลือเหมือนเดิม
            result = dfs(graph, neighbor, goal, path.copy())
            # เช็คถ้ามีผลลัพธ์(มี elemnet ใน list) ให้ return ผลลัพธ์นั้นออกมา
            if result:
                return result  # Return the first found path
```

```python
# กำหนดค่า start และ goal
# start คือจุดเริ่มต้นใน graph
start = "A"
# goal คือจุดเป้าหมายที่ต้องไปให้ถึงใน graph
goal = "E"

# เขียน try catch เพื่อดักจับอาจจะมีการเกิด exception จาก dfs นี้ไว้
try:
    # เรียกใช้ algorithm: dfs เพื่อหาเส้นทาง
    path = dfs(all_graphs.graph3, start, goal)
    # แสดงผลลัพธ์
    print(f"เส้นทางจาก {start} ไปยัง {goal} คือ")
    print(" → ".join(path))
except Exception as e:
    # แสดงข้อความของ exception
    print(e.__str__())
```

graphs.py

src > graphs.py > ...

You, 28 minutes ago | 1 author (You)

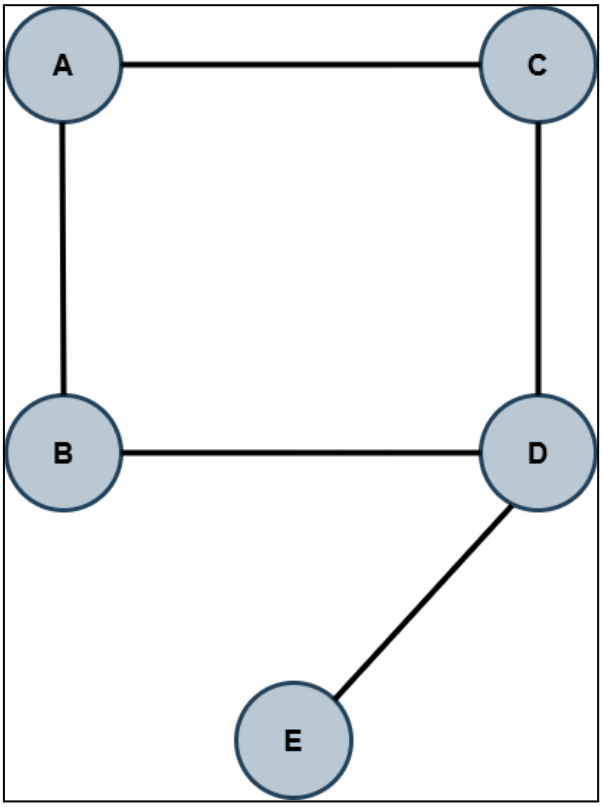
```python
# เส้นทางของ graph
# key คือ vertex
# value คือ vertex ที่เชื่อมต่อกันด้วยเส้น edge กับ vertex ของ key (neighbor)

# ? ทุก graphs ที่อยู่ใน module นี้เป็น Undirected Graphs ทั้งหมด
# กราฟอันที่ 1
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D"],
    "D": ["B", "C", "E"],
    "E": ["D"],
}
```

```python
# กราฟอันที่ 2
graph2 = {
    "A": ["B", "G"],
    "B": ["A", "G", "C"],
    "C": ["B", "E"],
    "E": ["C", "G", "J", "K"],
    "G": ["B", "E", "J"],
    "J": ["G", "E", "D", "F"],
    "D": ["J", "F", "K"],
    "F": ["J", "D", "K"],
    "K": ["E", "D", "F"],
}
```

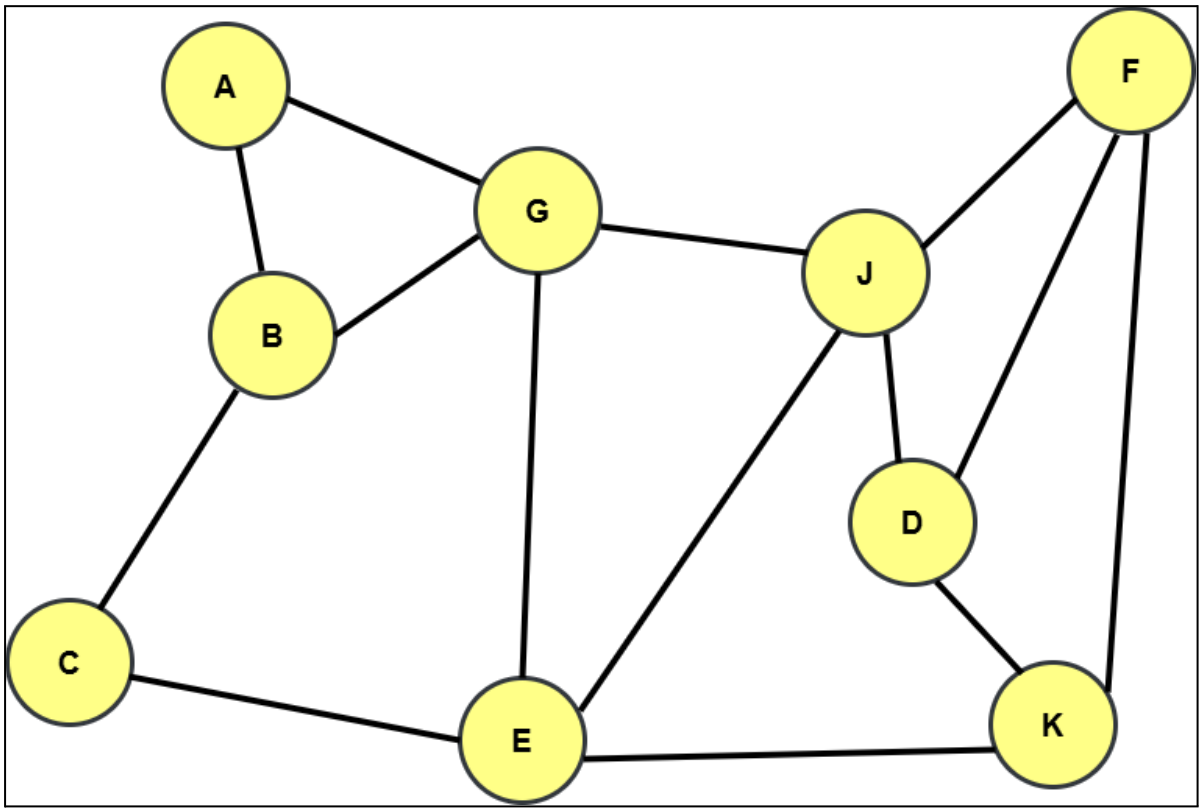
```python
# กราฟอันที่ 3
graph3 = {
    "A": ["B", "C"],
    "B": ["A", "D", "E"],
    "D": ["B", "H", "I"],
    "H": ["D"],
    "I": ["D"],
    "E": ["B"],
    "C": ["F", "G"],
    "F": ["C"],
    "G": ["J", "C"],
    "J": ["K", "G"],
    "K": ["J"],
}
```

รูปภาพกราฟ

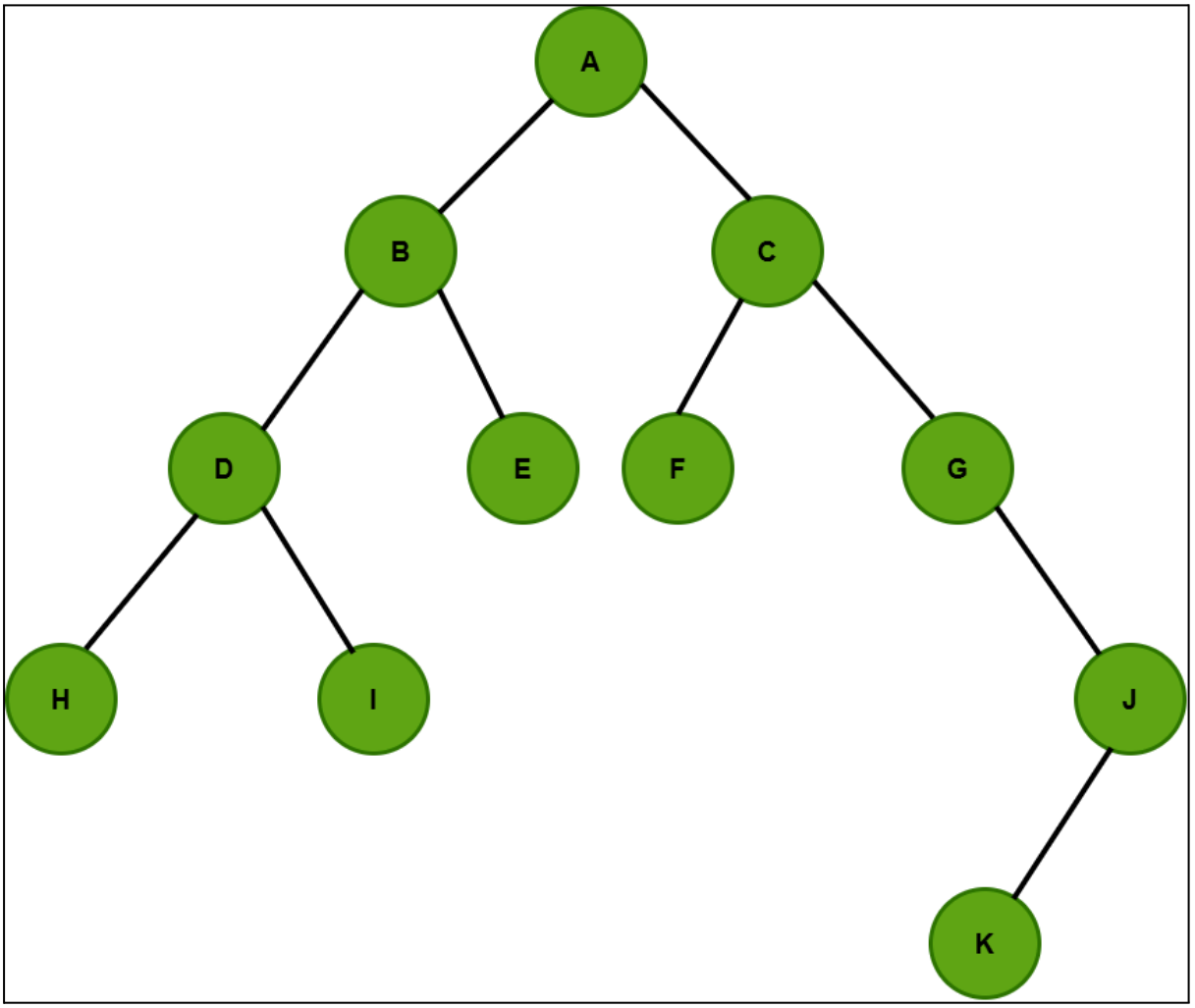
กราฟอันที่ 1



กราฟอันที่ 2



กราฟอันที่ 3



ขั้นตอนวิธีการทำงานของ Algorithm

อธิบายการทำงานของโค้ดทีละขั้นตอนในการหาเส้นทางจากจุด A ไปยังจุด E ด้วย DFS

โค้ดที่ให้มาใช้ Depth-First Search (DFS) เพื่อหาเส้นทางจากจุดเริ่มต้น (A) ไปยังจุดหมายปลายทาง (E) ในกราฟที่กำหนด

Step 1: กำหนดกราฟ

เรามีกราฟที่ใช้โครงสร้าง dictionary โดยมี key เป็นจุด (node) และ value เป็น list ของจุดที่เชื่อมต่อ (neighbors)

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D"],
    "D": ["B", "C", "E"],
    "E": ["D"],
}
```

Step 2: ตรวจสอบว่าจุดเริ่มต้นและจุดหมายมีอยู่ในกราฟหรือไม่

```
if not has_vertices(graph, start, goal):
    raise Exception(f"ไม่มี vertices {start} หรือ {goal} ที่อยู่ใน graph!")
```

ฟังก์ชัน `has_vertices(graph, start, goal)` จะทำงานโดย:

1. สร้าง set ที่เก็บ ทุก node ที่มีอยู่ในกราฟ
2. ตรวจสอบว่า ทั้งจุดเริ่มต้น (A) และจุดหมายปลายทาง (E) มีอยู่ในกราฟหรือไม่
3. ถ้ามีอยู่ในกราฟ ⟶ ทำงานต่อ

   ถ้าไม่มีอยู่ในกราฟ ⟶ แจ้งข้อผิดพลาด (raise Exception)

Step 3: เริ่มการค้นหาด้วย DFS

```
path = dfs(all_graphs.graph, start, goal)
```

เราจะเริ่ม DFS จาก A เพื่อค้นหา E โดยกระบวนการเป็นดังนี้:

Step 4: อธิบายการทำงานของ DFS ทีละขั้นตอน

เริ่มต้นที่ A

```
path.append(start)  # path = ["A"]
```

- A ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า A == E หรือไม่ ⟶ **ไม่ใช่** (ทำงานต่อ)
- สำรวจเพื่อนบ้านของ A คือ **["B", "C"]**
- ลองไปทางแรกก่อน ⟶ B
- ไปที่ B

```
path.append("B")  # path = ["A", "B"]
```

- B ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า B == E หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ B คือ **["A", "D"]**
- A เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ D

ไปที่ D

```
path.append("D")  # path = ["A", "B", "D"]
```

- D ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า D == E หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ D คือ **["B", "C", "E"]**
- B เคยมาแล้ว ⟶ **ข้าม**
- C เป็นอีกตัวเลือก แต่ E เป็นเป้าหมาย ⟶ เลือกไปที่ E ก่อน

ไปที่ E

```
path.append("E")  # path = ["A", "B", "D", "E"]
```

- E ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า E == E ⟶ **เป็นจริง!**
- DFS จบลง และคืนค่าเส้นทางที่พบ

Step 5: แสดงผลลัพธ์

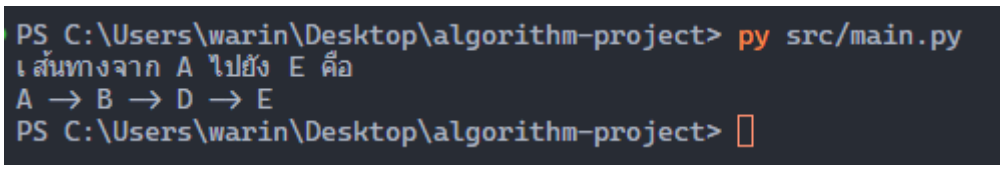
```
print(f"เส้นทางจาก {start} ไปยัง {goal} คือ")
print(" -> ".join(path))
```

สรุปการทำงานของ DFS

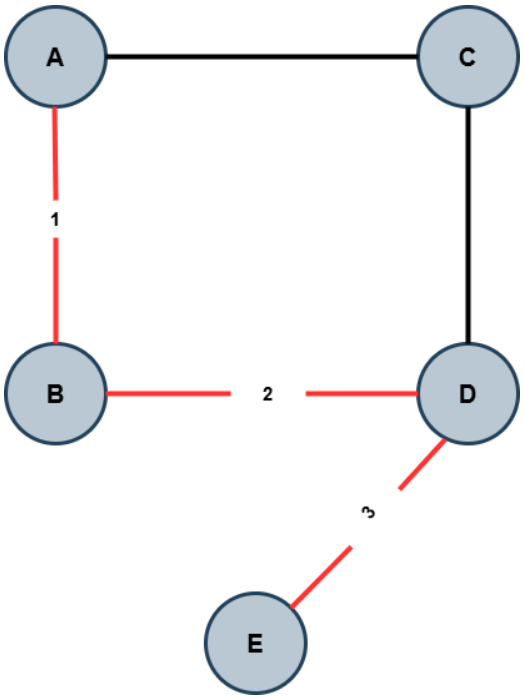
1. เริ่มต้นที่ A
2. ไปที่ B
3. ไปที่ D
4. ไปที่ E (**เป้าหมาย**)
5. เจอเป้าหมาย ⟶ **คืนค่าเส้นทางที่พบ**

ภาพผลลัพธ์ที่ได้คือ:

เส้นทางจาก A ไปยัง E คือ



กราฟ 1



อธิบายการทำงานของ DFS ทีละขั้นตอน (จาก A ไป K ใน graph2)

```
graph2 = {
    "A": ["B", "G"],
    "B": ["A", "G", "C"],
    "C": ["B", "E"],
    "E": ["C", "G", "J", "K"],
    "G": ["B", "E", "J"],
    "J": ["G", "E", "D", "F"],
    "D": ["J", "F", "K"],
    "F": ["J", "D", "K"],
    "K": ["E", "D", "F"],
}
```

Step 1: ตรวจสอบว่าจุดเริ่มต้นและจุดหมายอยู่ในกราฟ

```
if not has_vertices(graph2, start, goal):
    raise Exception(f"ไม่มี vertices {start} หรือ {goal} ที่อยู่ใน graph!")
```

ผลลัพธ์:

- A และ K มีอยู่ในกราฟ ⟶ ทำงานต่อได้

Step 2: เริ่มการค้นหาด้วย DFS

```
path = dfs(graph2, start, goal)
```

เริ่มต้นที่ A และใช้ DFS เพื่อหา K

Step 3: ค้นหาเส้นทางจาก A ไป K ทีละขั้นตอน

เริ่มต้นที่ A

path.append("A") # path = ["A"]

- A ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า A == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ A คือ **["B", "G"]**
- ไปที่ตัวเลือกแรก ⟶ B

ไปที่ B

path.append("B") # path = ["A", "B"]

- B ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า B == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ B คือ **["A", "G", "C"]**
- A เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ G

ไปที่ G

path.append("G") # path = ["A", "B", "G"]

- G ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า G == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ G คือ **["B", "E", "J"]**
- B เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ E

ไปที่ E

```
path.append("E")  # path = ["A", "B", "G", "E"]
```

- E ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า E == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ E คือ **["C", "G", "J", "K"]**
- C และ G เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ J

ไปที่ J

```
path.append("J")  # path = ["A", "B", "G", "E", "J"]
```

- J ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า J == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ J คือ **["G", "E", "D", "F"]**
- G และ E เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ D

ไปที่ D

```
path.append("D")  # path = ["A", "B", "G", "E", "J", "D"]
```

- D ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า D == K หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ D คือ **["J", "F", "K"]**
- J เคยมาแล้ว ⟶ **ข้าม**
- ไปที่ K **(จุดหมาย!)**

ไปที่ K (เป้าหมาย)

```
path.append("K")  # path = ["A", "B", "G", "E", "J", "D", "K"]
```

- K ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า K == K ⟶ **เป็นจริง!**
- DFS จบลง และคืนค่าเส้นทางที่พบ

Step 4: แสดงผลลัพธ์

```
print(f"เส้นทางจาก {start} ไปยัง {goal} คือ")
print(" -> ".join(path))
```

ผลลัพธ์ที่ได้

เส้นทางจาก A ไปยัง K คือ

A -> B -> G -> E -> J -> D -> K

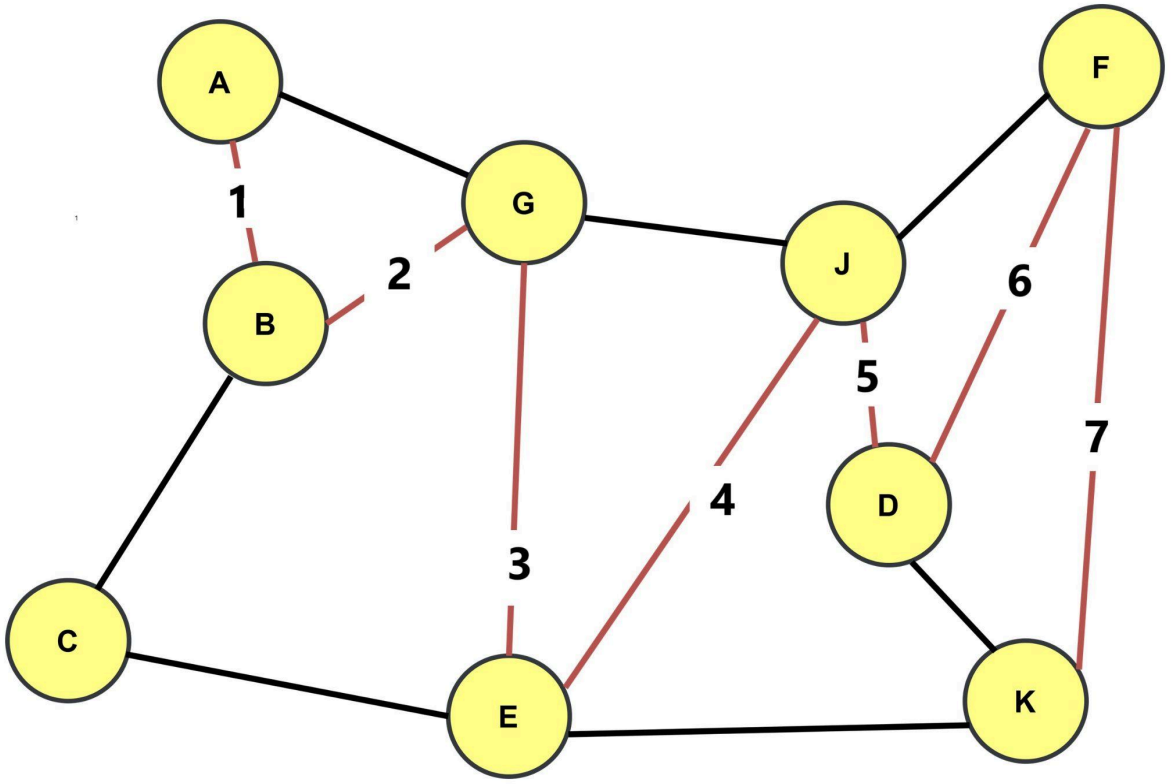
สรุปการทำงานของ DFS

1. เริ่มต้นที่ A
2. ไปที่ B
3. ไปที่ G
4. ไปที่ E
5. ไปที่ J
6. ไปที่ D
7. ไปที่ K (เป้าหมาย)

ภาพผลลัพธ์ที่ได้คือ:

```
PS C:\Users\warin\Desktop\algorithm-project> py src/main.py
เส้นทางจาก A ไปยัง K คือ
A → B → G → E → J → D → F → K
PS C:\Users\warin\Desktop\algorithm-project>
```

กราฟ 2



อธิบายการทำงานของ DFS ทีละขั้นตอน (จาก A ไป I ใน graph3)

graph3 = {

```
    "A": ["B", "C"],
    "B": ["A", "D", "E"],
    "D": ["B", "H", "I"],
    "H": ["D"],
    "I": ["D"],
    "E": ["B"],
    "C": ["F", "G"],
    "F": ["C"],
    "G": ["J", "C"],
    "J": ["K", "G"],
    "K": ["J"],
}
```

Step 1: ตรวจสอบว่าจุดเริ่มต้นและจุดหมายอยู่ในกราฟ

```
if not has_vertices(graph3, start, goal):
    raise Exception(f"ไม่มี vertices {start} หรือ {goal} ที่อยู่ใน graph!")
```

ผลลัพธ์:

- A และ I มีอยู่ในกราฟ ⟶ **ทำงานต่อได้**

Step 2: เริ่มการค้นหาด้วย DFS

```
path = dfs(graph3, start, goal)
```

เริ่มต้นที่ A และใช้ DFS เพื่อหา I

Step 3: ค้นหาเส้นทางจาก A ไป I ทีละขั้นตอน

เริ่มต้นที่ A

```
path.append("A")  # path = ["A"]
```

- A ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า A == I หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ A คือ **["B", "C"]**
- ไปที่ตัวเลือกแรก ⟶ B

ไปที่ B

```
path.append("B")  # path = ["A", "B"]
```

- B ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า B == I หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ B คือ **["A", "D", "E"]**
- A เคยมาแล้ว ⟶ ข้าม
- ไปที่ D

ไปที่ D

```
path.append("D")  # path = ["A", "B", "D"]
```

- D ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า D == I หรือไม่ ⟶ **ไม่ใช่**
- สำรวจเพื่อนบ้านของ D คือ **["B", "H", "I"]**
- B เคยมาแล้ว ⟶ ข้าม
- H ยังไม่ได้ไป แต่ I เป็นเป้าหมาย ⟶ ไปที่ I ก่อน

ไปที่ I (เป้าหมาย)

```
path.append("I")  # path = ["A", "B", "D", "I"]
```

- `I` ถูกเพิ่มลงในเส้นทาง
- ตรวจสอบว่า `I == I` ⟶ **เป็นจริง!**
- **DFS จบลง และคืนค่าเส้นทางที่พบ**

Step 4: แสดงผลลัพธ์

```
print(f"เส้นทางจาก {start} ไปยัง {goal} คือ")
print(" -> ".join(path))
```

ผลลัพธ์ที่ได้

เส้นทางจาก A ไปยัง I คือ

A -> B -> D -> I

สรุปการทำงานของ DFS

1. เริ่มต้นที่ A
2. ไปที่ B
3. ไปที่ D
4. ไปที่ I (เป้าหมาย)

ภาพผลลัพธ์ที่ได้คือ:

```
PS C:\Users\warin\Desktop\algorithm-project> py src/main.py
เส้นทางจาก A ไปยัง I คือ
A → B → D → I
PS C:\Users\warin\Desktop\algorithm-project>
```

กราฟ 3