การประยุกต์ใช้ Travelling Salesman Problem กับการวางแผนการเดินทาง

นายนพณัฐ ประเสริฐวงษา
นายธิติ ติยะชัยพานิช
นายคณิน วราสินธ์
นายเกษมสันต์ คำแดง
นายศักดิ์สิทธิ์ เวชวิทยาขลัง
นายรัชชนนท์ ทิพทามูล
นายธรรมภณ แฉ่งพาที
นายณัฐนันท์ จารุกรกุล

รายงานนี้เป็นส่วนหนึ่งของรายวิชา CPE112 Programming With Data Structures ตามหลักสูตร
ปริญญาวิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์
มหาวิทยาลัยเทคโนโลยีพระจอมเกล้าธนบุรี
ปีการศึกษา 2566

คำนำ

รายงานฉบับนี้จัดทำขึ้นเพื่อศึกษาการใช้งาน data structure กับสถานการณ์จริง ซึ่งผู้จัดทำได้รับมอบหมายจากคณะอาจารย์ผู้สอนให้นำ ความรู้เรื่อง data structure นำมาประยุกต์ใช้เพื่อสร้างเป็นโครงงานคอมพิวเตอร์โดยมีวัตถุประสงค์เพื่อให้คณะผู้จัดทำฝึกฝนและประยุกต์ใช้ความรู้เรื่อง data structure ในสถานการณ์จริง คณะผู้จัดทำหวังเป็นอย่างยิ่งว่ารายงานฉบับนี้จะเป็นประโยชน์กับผู้ที่สนใจในด้าน data structure หากมีความผิดพลาดประการใดในรายงานฉบับนี้ ทางคณะผู้จัดทำขออภัยไว้ ณ ที่นี้ด้วย

คณะผู้จัดทำ

สารบัญ

สารบัญรูป

รูป หน้า

บทที่ 1 บทนำ

1.1 ที่มาและความสำคัญ

ปัญหา Travelling Salesman Problem (TSP) เป็นปัญหาที่มีความสำคัญในหลายๆ สาขา เช่น โลจิสติกส์ การวางแผนการผลิต และการขนส่ง เนื่องจากเกี่ยวข้องกับการหาวิธีที่สั้นที่สุดในการเดินทางไปยังหลายๆ เมืองหรือจุดหมายโดยไม่ผ่านจุดเดิมซ้ำ (ยกเว้นจุดเริ่มต้น) ความสำคัญของการแก้ปัญหา TSP คือการช่วยลดค่าใช้จ่ายและเวลาในการขนส่ง เพิ่มประสิทธิภาพการผลิต และพัฒนาความรู้ในด้านอัลกอริธึมและการแก้ปัญหาทางคณิตศาสตร์ นักวิจัยและนักพัฒนาซอฟต์แวร์จึงได้คิดค้นอัลกอริธึมหลายแบบเพื่อหาวิธีการแก้ปัญหาที่มีประสิทธิภาพที่สุด

ในบรรดาอัลกอริธึมที่ใช้ในการแก้ปัญหา TSP Brute Force Algorithm เป็นวิธีการที่ตรงไปตรงมาและให้คำตอบที่ถูกต้องที่สุดโดยการทดลองทุกความเป็นไปได้และเลือกวิธีที่ดีที่สุด แม้ว่าจะมีความแม่นยำสูง แต่ใช้เวลาประมวลผลนานมากเมื่อจำนวนเมืองเพิ่มขึ้น เนื่องจากจำนวนความเป็นไปได้เพิ่มขึ้นอย่างทวีคูณ ในทางตรงกันข้าม Nearest Neighbor Algorithm เป็นอัลกอริธึมที่มีความเร็วในการประมวลผลสูงกว่า โดยจะเลือกเดินทางไปยังเมืองที่ใกล้ที่สุดก่อน แม้ว่าจะไม่สามารถรับประกันได้ว่าจะได้เส้นทางที่สั้นที่สุดเสมอแต่ก็เป็นการประนีประนอมระหว่างความเร็วและความแม่นยำ

การศึกษาและเปรียบเทียบอัลกอริธึมทั้งสองแบบนี้ช่วยให้เห็นถึงข้อดีและข้อเสียของแต่ละวิธี และช่วยในการเลือกใช้อัลกอริธึมที่เหมาะสมกับสถานการณ์ที่แตกต่างกัน ข้อมูลที่ได้รับจากการศึกษาและเปรียบเทียบนี้จะเป็นประโยชน์ทั้งในเชิงทฤษฎีและการประยุกต์ใช้ในทางปฏิบัติ

1.2 วัตถุประสงค์

1. เพื่อศึกษาและทำความเข้าใจเกี่ยวกับปัญหา Travelling Salesman Problem (TSP) และการประยุกต์ใช้ในชีวิตจริง
2. เพื่อเปรียบเทียบประสิทธิภาพของ Brute Force Algorithm และ Nearest Neighbor Algorithm ในการแก้ปัญหา TSP

1.3 ขอบเขตโครงงาน

ทางคณะผู้จัดทำได้จำกัดขอบเขตของโครงงานนี้ไว้เพียงสำหรับการวิเคราะห์เส้นทางที่สั้นที่สุดระหว่าง 14 จังหวัดในประเทศไทย ได้แก่ กรุงเทพมหานคร อยุธยา ลพบุรี ประจวบคีรีขันธ์ กระบี่ โคราช เชียงใหม่ เชียงราย แม่ฮ่องสอน สุโขทัย ชลบุรี จันทบุรี กาญจนบุรี และราชบุรี ผ่านการโดยสาร

ทางรถยนต์เท่านั้น เพื่อความสะดวกสบายแก่การคำนวน การเก็บและรวบรวมข้อมูล และการประมวลผลข้อมูลโดยรวม

1.4 ผลที่คาดว่าได้รับ

1. ความเข้าใจที่ลึกซึ้งยิ่งขึ้นเกี่ยวกับ TSP: ผู้จัดทำจะมีความเข้าใจที่ลึกซึ้งเกี่ยวกับปัญหา Travelling Salesman Problem และการประยุกต์ใช้ในสาขาต่างๆ

2. การพัฒนาอัลกอริธึมที่มีประสิทธิภาพ: ผู้จัดทำจะได้เรียนรู้และพัฒนา Brute Force Algorithm และ Nearest Neighbor Algorithm สำหรับการแก้ปัญหา TSP และทดสอบประสิทธิภาพของอัลกอริธึมเหล่านี้

3. การเปรียบเทียบและวิเคราะห์อัลกอริธึม: การเปรียบเทียบและวิเคราะห์ข้อดีและข้อเสียของ Brute Force Algorithm และ Nearest Neighbor Algorithm จะช่วยให้สามารถเลือกใช้อัลกอริธึมที่เหมาะสมกับปัญหาต่างๆได้

4. แนวทางสำหรับการประยุกต์ใช้ในชีวิตจริง: ข้อมูลและผลการทดสอบที่ได้รับจะสามารถนำไปประยุกต์ใช้ในการแก้ปัญหาที่เกี่ยวข้องกับการเดินทางและการขนส่งในชีวิตจริงได้

5. การพัฒนาทักษะการเขียนโปรแกรมและการแก้ปัญหา: ผู้จัดทำจะได้พัฒนาทักษะในการเขียนโปรแกรม การวิเคราะห์ปัญหา และการแก้ปัญหาด้วยอัลกอริธึมที่ซับซ้อน

1.5 การแบ่งหน้าที่การทำงาน

ในกลุ่มของคณะผู้จัดทำ สมาชิกแต่ละคนมีตำแหน่งและหน้าที่ ดังนี้

ชื่อของสมาชิก	ตำแหน่ง/หน้าที่
นายนพณัฐ ประเสริฐวงษา	ผู้ศึกษาและเขียน pipeline code ของ Traveling Saleman Problem / ผู้ประสานงานกลุ่ม
นายธิติ ติยะชัยพานิช	ผู้ต้นคิดไอเดีย/ผู้จัดทำสไลด์นำเสนอ/ผู้นำเสนอ
นายคณิน วราสินธ์	ผู้ดูแลภาพรวมของโปรแกรม / ผู้เขียนโปรแกรมสนับสนุน / ผู้นำเสนอ
นายเกษมสันต์ คำแดง	ผู้เขียนโปรแกรม/ผู้ทดสอบโปรแกรม / ผู้นำเสนอ

นายศักดิ์สิทธิ์ เวชวิทยาขลัง	ผู้เขียนโปรแกรมสนับสนุน/ผู้ทดสอบโปรแกรม / ผู้นำเสนอ
นายรัชชนนท์ ทิพทามูล	ผู้จัดทำสไลด์นำเสนอ/การจัดทำรายงาน / ผู้นำเสนอ
นายธรรมภณ แฉ่งพาที	ผู้จัดทำสไลด์นำเสนอ/การจัดทำรายงาน / ผู้นำเสนอ
นายณัฐนันท์ จารุกรกุล	ผู้จัดทำสไลด์นำเสนอ/ผู้นำเสนอ

บทที่ 2 ทฤษฎีที่เกี่ยวข้อง

2.1 กราฟ (Graph)

2.1.1 กราฟไม่มีทิศทาง (Undirected Graph)

เป็นกราฟที่ไม่กำหนดทิศทางของเส้นเชื่อมระหว่างจุดยอด

2.1.2 กราฟมีทิศทาง (Directed Graph)

เป็นกราฟที่กำหนดทิศทางของเส้นเชื่อมระหว่างจุดยอด

2.1.3 กราฟที่มีน้ำหนัก (Weighted Graph)

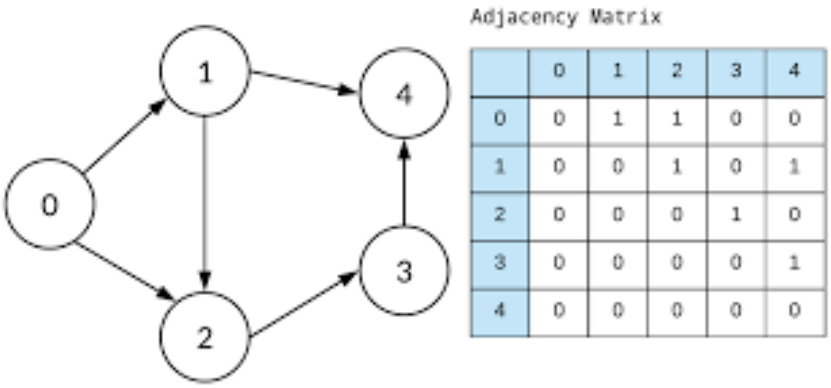
เป็นกราฟที่กำหนดค่าขนาดของเส้นเชื่อมระหว่างจุดยอด

2.2 ตัวแทนเชิงกราฟ (Graph Representation)

2.2.1 เมตริกซ์ประชิด (Adjacency Matrix)

การแสดงกราฟด้วยการใช้เมตริกซ์ โดยสามารถแสดงได้ทุกรูปแบบ ซึ่งมีคุณสมบัติดังนี้

1. เข้าใจได้ง่าย
2. รองรับจำนวนจุดยอดที่คงที่
3. สามารถใช้เมตริกซ์ได้มากกว่าวิธีอื่น เนื่องจากมีจำนวน cell ภายในเมตริกซ์สำหรับทุกๆ จุดยอด



	0	1	2	3	4
0	0	1	1	0	0
1	0	0	1	0	1
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

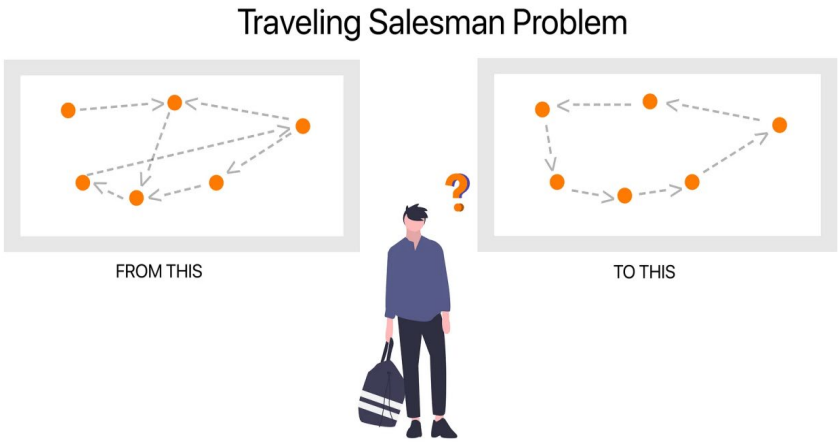
ภาพที่ 2.1 - ตัวอย่างการแสดงกราฟประเภทต่างๆ โดยการใช้เมตริกซ์

2.2.2 รายการประชิด (Adjacency List)

รายการประชิด เป็นลิสต์ที่ถูกใช้ในการเก็บข้อมูลของจุดยอดที่อยู่ติดกับจุดยอดแต่ละตัว

2.3 Travelling Salesman Problem

ปัญหาการเดินทางของพนักงานขาย (Travelling Salesman Problem: TSP) เป็นปัญหาที่สำคัญในวงการคณิตศาสตร์และคอมพิวเตอร์ ในปัญหานี้เราสมมติการเป็นพนักงานขายที่จะต้องเดินทางไปขายของในที่ต่างๆ โดยใช้ระยะการเดินทางที่น้อยที่สุดและจะต้องเดินทางกลับมาที่จุดเริ่มต้น โดยปัญหานี้มีทางแก้ไขหลายวิธีแต่ในโครงงานนี้เราจะมุ่งเน้นอยู่ 2 อัลกอริทึมหลัก Brute Force Algorithm และ Nearest Neighbor Algorithm



ภาพที่ 2.2 - ตัวอย่างปัญหาการเดินทางของพนักงานขาย

2.3.1 Brute Force Algorithm

Brute Force Algorithm ตามชื่อของอัลกอริทึมนี้ เป็นการแก้ปัญหาโดยใช้กำลังโดยในบรรดาวิธีการแก้ปัญหาเป็นวิธีการที่ง่ายที่สุดแต่ใช้แรงคำนวนของคอมพิวเตอร์มากที่สุด เราจะทำการค้นหาเส้นทางที่สั้นที่สุดด้วยการคำนวณทุกความเป็นไปได้และเลือกใช้เส้นทางที่มีระยะน้อยที่สุด โดยการแก้ปัญหาด้วยวิธีนี้มีข้อดีและข้อเสีย ดังนี้

ข้อดี

1. ความง่าย ทั้งในด้านการออกแบบและแนวความคิด
2. ผลลัพธ์ที่ได้แม่นยำ เนื่องจากเป็นผลที่ดีที่สุดจากทุกความเป็นไปได้

ข้อเสีย

1. ประสิทธิภาพต่ำ ใช้แรงคำนวณและเวลามาก โดย Time Complexity ของ Brute Force Algorithm ในการแก้ปัญหา TSP เป็น O(n!) (n เป็นจำนวนของ node ในกราฟ)

2. ไม่เหมาะสำหรับปัญหาที่ซับซ้อน ประสิทธิภาพในการแก้ปัญหาเมื่อเทียบกับวิธีอื่นในปัญหาที่ไม่ซับซ้อนไม่มีความแตกต่างมากนัก แต่เมื่อความซับซ้อนเพิ่มขึ้น ประสิทธิภาพของ Brute Force Algorithm จะลดลงตามมา

2.3.2 Nearest Neighbor Algorithm

Nearest Neighbor Algorithm เป็นอีกหนึ่งวิธีที่สามารถใช้แก้ปัญหา TSP โดยโปรแกรมจะทำการค้นหาจุดยอดหรือเมืองที่อยู่ใกล้ที่สุด (มีเส้นทางระหว่างกันน้อยที่สุด) และให้เดินทางตามเส้นทางนั้นไปเรื่อยๆ จนกลับมาที่จุดเริ่มต้น โดยการแก้ปัญหาด้วยวิธีนี้มีข้อดีและข้อเสีย ดังนี้

ข้อดี

1. ความง่าย ทั้งในด้านการออกแบบและแนวความคิด
2. สามารถให้ผลลัพธ์ได้อย่างรวดเร็วและมีประสิทธิภาพสูง

ข้อเสีย

1. ผลลัพธ์ที่ได้อาจมีความไม่แม่นยำ และอาจจะให้ผลลัพธ์ออกมาเป็นเส้นทางที่ยาวที่สุดได้ตามคุณภาพของข้อมูล
2. ไม่เหมาะกับกราฟที่มีขนาดใหญ่และการจัดระเบียบที่ซับซ้อน เนื่องจากประสิทธิภาพและความเร็วของการประมวลผลจะลดลงตามขนาดและความซับซ้อนของปัญหา
3. ใช้หน่วยความจำมาก หากใช้กับกราฟที่มีขนาดใหญ่

บทที่ 3 วิธีการดำเนินงาน

3.1 อุปกรณ์ที่ใช้ในการทำงาน

3.1.1 เครื่องมือ

- เครื่องคอมพิวเตอร์ (Windows 10/Windows 11)

3.1.2 ซอฟแวร์

- Replit (Online)
- Visual Studio Code

3.2 ขั้นตอนการดำเนินงาน

3.2.1 เลือกหัวข้อและวางแผนการดำเนินงาน

มีการประชุมเพื่อมองหาปัญหาเพื่อนำมาดำเนินโครงงาน โดยจะเป็นการประชุมแบบเสนอ ไอเดียจากทุกๆ ฝ่ายแล้วมีการแย้งและสนับสนุนกัน หลังจากได้ประเด็นปัญหาแต่ละฝ่ายแยกย้ายกัน ไปหาข้อมูลในส่วนที่ได้รับมอมหมายพร้อมกับการพิมพ์ข้อมูลนั้นลงในเอกสารส่วนรวม เมื่อมีข้อมูล ครบถ้วนจึงแบ่งหน้าที่การทำงาน เพื่อเริ่มดำเนินงาน

3.2.2 ออกแบบรูปแบบและระบบการทำงานของโปรแกรม

เมื่อเริ่มโปรแกรม ตัวโปรแกรมจะทำการแสดงผลจังหวัด ที่มีข้อมูลบันทึกอยู่ในฐานข้อมูลทั้งหมดจากนั้นโปรแกรมจะให้ผู้ใช้ทำการเลือกว่า ผู้ใช้ต้องการเพิ่มข้อมูลเส้นทางของจังหวัดใหม่ ประมวลผลเส้นทางที่สั้นที่สุด หรือหยุดโปรแกรม ดังภาพด้านล่าง

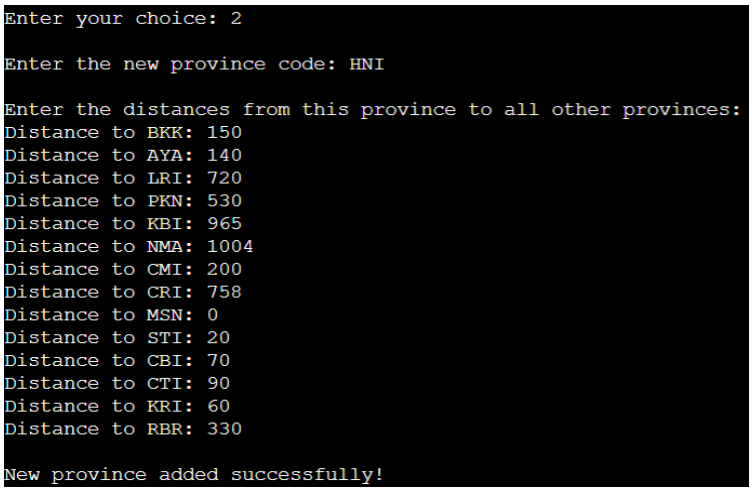
```
==========================
     Province Travel
==========================

Available Provinces: BKK AYA LRI PKN KBI NMA CMI CRI MSN STI CBI CTI KRI RBR

Choose an option:
1. Calculate TSP
2. Add new province
3. Exit
Enter your choice: 1
```

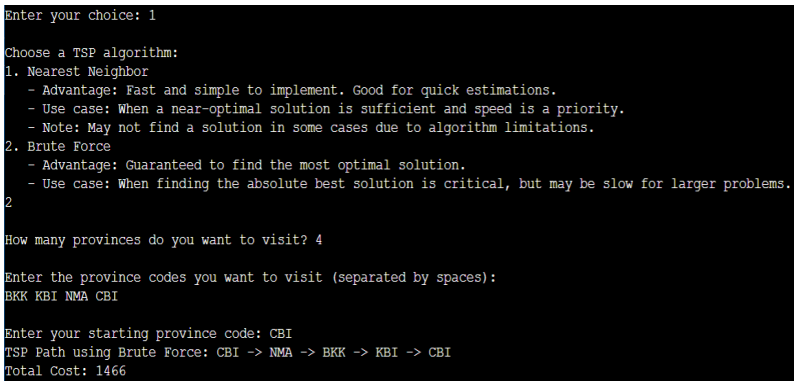
ภาพที่ 3.1 - หน้าจอแสดงผลเริ่มต้น

หากผู้ใช้เลือกต้องการเพิ่มข้อมูลเส้นทางของจังหวัดใหม่ ระบบจะทำการให้ผู้ใช้ใส่ข้อมูลจำพวกชื่อของจังหวัด (ในระบบ 3 ตัวอักษร) และข้อมูลของเส้นทางระหว่างจังหวัดที่มีข้อมูลอยู่แล้วและจังหวัดใหม่แล้วระบบจะทำการให้ผู้ใช้ทำการเลือกการปฏิบัติการอีกครั้ง ดังภาพด้านล่าง



ภาพที่ 3.2 - ตัวอย่างการเพิ่มข้อมูลเส้นทางของจังหวัดใหม่

หากผู้ใช้เลือกต้องการประมวลผลเส้นทางที่สั้นที่สุด ระบบจะให้ผู้ใช้เลือกวิธีที่ระบบใช้การวิเคราะห์ข้อมูลระหว่าง Brute Force Algorithm และ Nearest Neighbor Algorithm จากนั้นระบบจะให้ผู้ใช้ทำการใส่ข้อมูลต่างๆ ได้แก่ จำนวนของจังหวัดที่ผู้ใช้ต้องการไปชื่อของจังหวัด ที่ผู้ใช้ต้องการไป (ในระบบ 3 ตัวอักษร) และชื่อของจังหวัดเริ่มต้นจากนั้นระบบจะทำการประมวลผลเส้นทางโดยใช้วิธีที่ ผู้ใช้ได้เลือกไว้ก่อนแสดงผลออกมาเป็นเส้นทางการเดินทางระหว่างจังหวัดเป็นลำดับและระยะทางทั้งหมดที่คำนวนออกมาได้ ดังภาพด้านล่าง



ภาพที่ 3.3 - ตัวอย่างการประมวลผลเส้นทางที่สั้นที่สุดแบบ Brute Force Algorithm

หากผู้ใช้เลือกต้องการหยุดโปรแกรมระบบก็จะปิดตัวเอง เป็นการสิ้นสุดโปรแกรมโดยปริยาย

3.2.3 รวบรวมข้อมูลไว้ใช้บรรจุลงในฐานข้อมูล

เราอ้างอิงและรวบรวมข้อมูลที่บรรจุลงในฐานข้อมูลเพื่อนำมาใช้ในการวิเคราะห์และคำนวนเส้นทางระหว่างจังหวัด 2 จังหวัดจาก Google Maps ซึ่งข้อมูลที่เราใช้ทั้งหมดจะถูกจัดเก็บในรูปของไฟล์ .txt

3.2.4 เขียนโปรแกรม

หลังจากการออกแบบและรวบรวมข้อมูลไว้ในฐานข้อมูล จะดำเนินการเขียนโปรแกรมใน Replit และ Visual Studio Code โดยมีขั้นตอนดังนี้

3.2.4.1 นิยามโครงสร้างข้อมูล

จะเริ่มการเขียนโปรแกรมโดยการใช้ struct ในการนิยาม struct node เพื่อเป็นตัวแทนของ node ใน adjacency list และใช้ struct ในการนิยาม struct Graph ซึ่งประกอบไปด้วยจำนวน vertices ชื่อของจังหวัด adjacency matrix และ adjacency list เพื่อเป็นตัวแทนของกราฟ

```
// Structure to represent a node in the adjacency list
struct node {
    int vertex;
    struct node *next;
};

// Structure to represent the graph
struct Graph {
    int numVertices;
    char provinceCodes[MAX_PROVINCES][MAX_CODE_LENGTH];
    int adjMatrix[MAX_PROVINCES][MAX_PROVINCES];
    struct node **adjLists;
};
```

ภาพที่ 3.4 - ภาพโปรแกรมในการนิยามโครงสร้างข้อมูล

3.2.4.2 การสร้างฟังก์ชันช่วย

ถัดมาจะเป็นการส้รางฟังก์ชันช่วยเพื่อที่จะควบคุมและจัดการกระบวนการต่างๆ ของกราฟ โดยมีฟังก์ชันต่างๆ ดังนี้

1. ฟังก์ชัน createNode( ) : สำหรับการสร้างจุดยอดในรายการประชิด

```c
struct node *createNode(int v) {
    struct node *newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}
```

ภาพที่ 3.5 - ภาพโปรแกรมฟังก์ชัน createNode( )

2. ฟังก์ชัน addEdge( ) : สำหรับการเพิ่มเวส้นเชื่อมในรายการประชิด

```c
void addEdge(struct Graph *graph, int s, int d, int cost) {
    struct node *newNode = createNode(d);
    newNode->next = graph->adjLists[s];
    graph->adjLists[s] = newNode;
}
```

ภาพที่ 3.6 - ภาพโปรแกรมฟังก์ชัน addEdge( )

3. ฟังก์ชัน readProvinceData( ) : สำหรับการดึงข้อมูลของกราฟจากไฟล์ .txt แล้วเริ่มกระบวนการสร้างกราฟ

```c
int readProvinceData(struct Graph *graph, const char *filename) {
    FILE *file = fopen(filename, "r");
    if (file == NULL) {
        perror("Error opening file");
        return 0;
    }

    fscanf(file, "%d", &graph->numVertices);

    graph->adjLists = malloc(graph->numVertices * sizeof(struct node *));
    for (int i = 0; i < graph->numVertices; i++) {
        graph->adjLists[i] = NULL;
    }

    for (int i = 0; i < graph->numVertices; i++) {
        fscanf(file, "%s", graph->provinceCodes[i]);
    }

    for (int i = 0; i < graph->numVertices; i++) {
        for (int j = 0; j < graph->numVertices; j++) {
            fscanf(file, "%d", &graph->adjMatrix[i][j]);
            if (graph->adjMatrix[i][j] != 0) {
                addEdge(graph, i, j, graph->adjMatrix[i][j]);
            }
        }
    }

    fclose(file);
    return 1;
}
```

ภาพที่ 3.7 - ภาพโปรแกรมฟังก์ชัน readProvinceData( )

4. ฟังก์ชัน findProvinceIndex( ) : สำหรับการหา index ของจังหวัดนั้นๆ ในรายการชื่อจังหวัด

```
int findProvinceIndex(struct Graph *graph, char *provinceCode) {
    for (int i = 0; i < graph->numVertices; i++) {
        if (strcmp(graph->provinceCodes[i], provinceCode) == 0) {
            return i;
        }
    }
    return -1;
}
```

ภาพที่ 3.8 - ภาพโปรแกรมฟังก์ชัน findProvinceIndex( )

5. ฟังก์ชัน findNearestNeighbor( ) : สำหรับหาจุดยอดที่อยู่ใกล้ที่สุดที่ยังไม่ได้ไปถึงเพื่อดำเนินต่อใน Nearest Neighbor Algorithm

```
int findNearestNeighbor(struct Graph *graph, int currentVertex, int *visited) {
    int minDistance = INT_MAX;
    int nearestNeighbor = -1;

    for (int i = 0; i < graph->numVertices; i++) {
        if (graph->adjMatrix[currentVertex][i] != 0 && !visited[i] &&
            graph->adjMatrix[currentVertex][i] < minDistance) {
            minDistance = graph->adjMatrix[currentVertex][i];
            nearestNeighbor = i;
        }
    }

    return nearestNeighbor;
}
```

ภาพที่ 3.9 - ภาพโปรแกรมฟังก์ชัน findNearestNeighbor( )

3.2.4.3 การเขียนโปรแกรมเพื่อแก้ปัญหา TSP

เมื่อมีฟังก์ชันพื้นฐานที่สำคัญครบแล้ว ต่อไปจะเป็นการเขียนโปรแกรมโดยมีเป้าหมายเพื่อแก้ปัญหา TSP ด้วยวิธี Brute Force Alogorithm และ Nearest Neighbor Alogorithm

1. ฟังก์ชัน permute( ) : สำหรับการสร้างเส้นทางการเดินทางระหว่างจังหวัดที่ผู้ใช้ต้องการไปทั้งหมดผ่านการเรียงสับเปลี่ยนและคำนวณระยะทางทั้งหมดที่ใช้

ในเส้นทาง เพื่อค้นหาเส้นทางที่สั้นที่สุด เป็นจำนวน n! ครั้ง ใช้ในกระบวนการทำ TSP แบบ Brute Force Algorithm

```
// Generate permutations of an array recursively
void permute(int *arr, int start, int end, struct Graph *graph, int startVertex,
long long *minCost, int *bestPath) {
    if (start == end) {
        long long currentCost = calculateCost(graph, arr, end);

        if (currentCost < *minCost && arr[0] == startVertex) {
            *minCost = currentCost;
            for (int i = 0; i < end + 1; i++) {
                bestPath[i] = arr[i];
            }
        }
    } else {
        for (int i = start; i <= end; i++) {
            swap(&arr[start], &arr[i]);
            permute(arr, start + 1, end, graph, startVertex, minCost, bestPath);
            swap(&arr[start], &arr[i]);
        }
    }
}
```

ภาพที่ 3.10 - ภาพโปรแกรมฟังก์ชัน permute( )

2. ฟังก์ชัน TSP_BruteForce( ) : เมื่อฟังก์ชันนี้ถูกเรียก โปรแกรมจะการเรียกฟังก์ชัน permute( ) ให้รับข้อมูลของจังหวัดที่ต้องการไปและจังหวัดเริ่มต้น และคืนรายการของเส้นทางที่สั้นที่สุดและระยะทางรวม เพื่อการแสดงค่า

```
// Brute Force Algorithm for TSP
void TSP_BruteForce(struct Graph *graph, int startVertex, int *included) {
    int numIncluded = 0;
    for (int i = 0; i < graph->numVertices; i++) {
        if (included[i]) {
            numIncluded++;
        }
    }

    int S[numIncluded];
    int j = 0;
    for (int i = 0; i < graph->numVertices; i++) {
        if (included[i]) {
            S[j++] = i;
        }
    }

    long long minCost = LLONG_MAX;
    int bestPath[numIncluded];

    permute(S, 0, numIncluded - 1, graph, startVertex, &minCost, bestPath);

    printf("TSP Path using Brute Force: ");
    for (int i = 0; i < numIncluded; i++) {
        printf("%s -> ", graph->provinceCodes[bestPath[i]]);
    }
    printf("%s\n", graph->provinceCodes[startVertex]);
    printf("Total Cost: %lld\n", minCost);
}
```

ภาพที่ 3.11 - ภาพโปรแกรมฟังก์ชัน TSP_BruteForce( )

3. ฟังก์ชัน TSP_NearestNeighbor( ) : เมื่อฟังก์ชันนี้ถูกเรียกโปรแกรมจะทำการรับข้อมูลของจังหวัดที่ต้องการไปและจังหวัดเริ่มต้นและเรียกฟังก์ชัน findNearestNeighbor( ) เพื่อค้นหาจังหวัดที่มีระยะทางจากจังหวัดปัจจุบัน น้อยสุด แล้วเดินทางตามเส้นทางนั้น โดยเพิ่มจังหวัดปัจจุบันลงในรายการเส้นทาง บวกระยะทางที่ใช้ลงในระยะทางรวม และให้จังหวัดกลายเป็นจังหวัดปัจจุบัน วนรอบซ้ำๆไปเรื่อยๆจนกว่าจะเดินทางจนครบทุกจังหวัดที่ต้องการไป แล้วจึงเดินทางกลับจังหวัดเริ่มต้นและบวกระยะทางระหว่างจังหวัดเริ่มต้นกับจังหวัดปลายลงในระยะทางรวม ก่อนคืนค่าออกมาเป็นรายการเส้นทางที่สมบูรณ์และระยะทางรวม เพื่อการแสดงค่า

```c
// Nearest Neighbor Algorithm for TSP (with backtracking)
void TSP_NearestNeighbor(struct Graph *graph, int startVertex, int *included) {
    int visited[MAX_PROVINCES] = {0};
    int path[MAX_PROVINCES];
    int currentVertex = startVertex;
    long long totalCost = 0;
    int provincesToVisit = 0;

    for (int i = 0; i < graph->numVertices; i++) {
        if (included[i]) {
            provincesToVisit++;
        }
    }

    visited[startVertex] = 1;
    path[0] = startVertex;
    int count = 1;
```

ภาพที่ 3.12 - ภาพโปรแกรมฟังก์ชัน TSP_NearestNeighbor( ) ส่วนแรก

```c
    while (count < provincesToVisit) {
        int nearest = findNearestNeighbor(graph, currentVertex, visited, included);

        if (nearest != -1) {
            visited[nearest] = 1;
            path[count] = nearest;
            totalCost += graph->adjMatrix[currentVertex][nearest];
            currentVertex = nearest;
            count++;
        } else {
            // Backtracking
            if (count > 1) {
                count--;
                visited[currentVertex] = 0;
                currentVertex = path[count - 1];
            } else {
                printf("Error: Could not find a valid path for the selected provinces.\n");
                printf("This might be due to limitations of the Nearest Neighbor algorithm.\n");
                return;
            }
        }
    }

    totalCost += graph->adjMatrix[currentVertex][startVertex];
    path[provincesToVisit] = startVertex;

    printf("TSP Path using Nearest Neighbor: ");
    for (int i = 0; i <= provincesToVisit; i++) {
        printf("%s", graph->provinceCodes[path[i]]);
        if (i < provincesToVisit) {
            printf(" -> ");
        }
    }
    printf("\nTotal Distance: %lld\n", totalCost);
}
```

ภาพที่ 3.13 - ภาพโปรแกรมฟังก์ชัน TSP_NearestNeighbor( ) ส่วนที่สอง

3.2.4.4 การเพิ่มความสะดวกสบายให้ผู้ใช้งาน

เพื่อให้ผู้ใช้งานเข้าถึงโปรแกรมได้ง่ายขึ้น
จึงเพิ่มตัวเลือกการทำงานเป็นการเพิ่มจังหวัดใหม่เข้าไปในฐานข้อมูลได้โดยตรง
ซึ่งได้ผลลัพธ์เป็นฟังก์ชัน addNewProvince( )

```
int addNewProvince(const char *filename, const char *provinceCode, int *distances,
                   int numExistingProvinces, struct Graph *graph) {
    // Check if the province code already exists
    for (int i = 0; i < numExistingProvinces; i++) {
        if (strcmp(graph->provinceCodes[i], provinceCode) == 0) {
            printf("Province code %s already exists.\n", provinceCode);
            return 0; // Indicate failure
        }
    }

    FILE *file = fopen(filename, "r+");
    if (file == NULL) {
        perror("Error opening file");
        return 0;
    }

    fprintf(file, "%d\n", numExistingProvinces + 1);

    for (int i = 0; i < numExistingProvinces; i++) {
        fprintf(file, "%s ", graph->provinceCodes[i]);
    }
    fprintf(file, "%s\n", provinceCode);

    for (int i = 0; i < numExistingProvinces; i++) {
        for (int j = 0; j < numExistingProvinces; j++) {
            fprintf(file, "%d ", graph->adjMatrix[i][j]);
        }
        fprintf(file, "%d ", distances[i]);
        fprintf(file, "\n");
    }

    for (int i = 0; i < numExistingProvinces; i++) {
        fprintf(file, "%d ", distances[i]);
    }
    fprintf(file, "0\n");

    fclose(file);
    return 1;
}
```

ภาพที่ 3.14 - ภาพโปรแกรมฟังก์ชัน addNewProvince( )

3.2.5 ทดสอบการทำงานของโปรแกรมและปรับปรุงแก้ไขข้อบกพร่องในระบบ

ผู้ที่มีหน้าที่ในการทดสอบการทำงานของโปรแกรมจะต้องทำการตรวจสอบว่าระบบในโปรแกรมทำงานเป็นปกติและในแบบที่ผู้พัฒนาต้องการหรือไม่
และแก้ไขหรือรายงานข้อบกพร่องที่พบในระหว่างการตรวจสอบให้กับผู้พัฒนา

บทที่ 4 ผลการดำเนินงาน

จากการดำเนินงานในการประยุกต์ใช้ Travelling Salesman Problem กับการวางแผนการเดินทาง ได้มีการทดลองใช้ทั้ง Brute Force Algorithm และ Nearest Neighbor Algorithm กับการเดินทางระหว่าง 14 จังหวัดในประเทศไทย โดยผลการดำเนินงาน สามารถสรุปได้ดังนี้:

4.1 การใช้ Brute Force Algorithm

จากการทดลองใช้อัลกอริทึม Brute Force ในการคำนวณเส้นทางที่สั้นที่สุดระหว่าง 14 จังหวัด พบว่าอัลกอริทึมนี้สามารถหาคำตอบที่แม่นยำที่สุดได้ แต่ใช้เวลาในการประมวลผลนาน เนื่องจากความซับซ้อนที่เป็น O(n!) ทำให้เมื่อจำนวนเมืองเพิ่มขึ้น เวลาในการประมวลผลจะเพิ่มขึ้นอย่างทวีคูณ

4.2 การใช้ Nearest Neighbor Algorithm

จากการทดลองใช้อัลกอริทึม Nearest Neighbor ในการคำนวณเส้นทาง พบว่าอัลกอริทึมนี้สามารถให้คำตอบได้อย่างรวดเร็วและมีประสิทธิภาพ แต่ผลลัพธ์ที่ได้อาจไม่แม่นยำเสมอไป บางครั้งอาจได้เส้นทางที่ยาวกว่าที่ควรจะเป็น แต่ยังคงเป็นวิธีที่ดีในการประนีประนอมระหว่างความเร็วและความแม่นยำ

4.3 การเปรียบเทียบประสิทธิภาพ

จากการเปรียบเทียบประสิทธิภาพของทั้งสองอัลกอริทึม พบว่า Brute Force Algorithm มีความแม่นยำสูงสุด แต่ใช้เวลาในการประมวลผลนาน ขณะที่ Nearest Neighbor Algorithm สามารถให้ผลลัพธ์ได้รวดเร็ว แต่ความแม่นยำอาจลดลงตามความซับซ้อนของปัญหา

4.4 ข้อดีและข้อเสียของแต่ละอัลกอริทึม

Brute Force Algorithm

- ข้อดี: ความแม่นยำสูง
- ข้อเสีย: ใช้เวลาประมวลผลนาน ไม่เหมาะกับปัญหาที่มีจำนวนเมืองมาก

Nearest Neighbor Algorithm

- ข้อดี: ประมวลผลได้รวดเร็ว
- ข้อเสีย: ความแม่นยำไม่สูงเมื่อเทียบกับ Brute Force Algorithm

บทที่ 5 สรุปผล

จากการดำเนินงานในการประยุกต์ใช้ Travelling Salesman Problem กับการวางแผนการเดินทาง พบว่าอัลกอริทึม Brute Force และ Nearest Neighbor มีข้อดีและข้อเสียที่แตกต่างกัน การเลือกใช้อัลกอริทึมขึ้นอยู่กับความต้องการของปัญหา หากต้องการความแม่นยำสูง ควรเลือกใช้ Brute Force Algorithm แต่หากต้องการความรวดเร็วในการประมวลผล ควรเลือกใช้ Nearest Neighbor Algorithm ข้อมูลและผลการทดสอบที่ได้รับสามารถนำไปประยุกต์ใช้ในการแก้ปัญหาที่เกี่ยวข้องกับการเดินทางและการขนส่งในชีวิตจริงได้ และยังเป็นแนวทางในการพัฒนาอัลกอริทึมที่มีประสิทธิภาพในอนาคต