บทที่ 3
การแก้ปัญหาโดยวิธีค้นหา
(Problem Solving by Searching)

ในบทนี้จะได้ศึกษา Goal-based agent ชนิดหนึ่งเรียกว่า Problem-solving agent หรือเอเยนต์แก้ปัญหา เอเยนต์ชนิดนี้ตัดสินใจว่าจะทำอะไรโดยค้นหาชุดของการกระทำที่จะนำไปสู่สถานะที่ต้องการ บทนี้จะเริ่มจากการกำหนดให้แน่ชัดว่าอะไรคือองค์ประกอบของปัญหา พร้อมทั้งยกตัวอย่างให้เห็นชัดเจน แล้วจึงกล่าวถึงอัลกอริทึมในการค้นหาทั่วไป สำหรับอัลกอริทึมที่ใช้ในบทนี้จะเป็นแบบที่ไม่ใช้ข้อมูลในการแก้ปัญหา ให้แต่นิยามของปัญหาเท่านั้น (Uninformed search algorithms)

3.1 เอเยนต์ที่แก้ปัญหา (Problem-solving agent)

จากที่กล่าวมาแล้วว่าเอเยนต์ฉลาด (Intelligent agent) ต้องพยายามทำให้เกิดเกณฑ์การวัดสมรรถนะสูงที่สุด วิธีหนึ่งที่ทำได้ง่ายคือให้เอเยนต์สร้างเป้าหมาย (Goal) ที่ต้องการให้บรรลุขึ้นมา และทำให้เป็นไปตามเป้าหมายนั้นให้มากที่สุด เนื่องจากการเข้าใกล้เป้าหมายมากขึ้นเท่าใด ก็ยิ่งทำให้เอเยนต์มีเกณฑ์การวัดสมรรถนะมากขึ้นเท่านั้นเพราะเท่ากับเข้าใกล้เกณฑ์ที่ใช้วัดความสำเร็จนั่นเอง

ดังนั้น การแก้ปัญหาของเอเยนต์ จึงสามารถกำหนดขึ้นเป็นขั้นตอนได้ดังนี้

3.1.1 สร้างเป้าหมาย (Goal formulation) เป้าหมายนี้สร้างขึ้นตามสถานการณ์จริงในขณะนั้น และต้องสอดคล้องกับการใช้เกณฑ์การวัดสมรรถนะของเอเยนต์ด้วย

ตัวอย่างเช่น สมมุติให้เอเยนต์กำลังพักผ่อนสุดสัปดาห์อยู่ที่เมืองอารัด ประเทศโรมาเนีย เกณฑ์การวัดสมรรถนะของเอเยนต์มีหลายปัจจัย ได้แก่

1. ผิวเป็นสีแทนจากการพักผ่อนอาบแดด
2. ใช้ชีวิตตามแบบฉบับชาวโรมาเนียให้เต็มที่
3. เที่ยวชมวิวทิวทัศน์
4. ท่องราตรีให้สนุก
5. ไม่มีอาการเมาค้าง
6. อื่น ๆ

ปัญหาในการตัดสินใจจะซับซ้อนเพราะแต่ละอย่างมีข้อดีข้อเสียซึ่งกันและกัน (เรียกว่า Trade off หมายถึงว่าถ้าทำบางเรื่องได้ดี ก็ต้องยอมเสียไปบางเรื่อง) และต้องศึกษาคู่มือนำเที่ยวโดยละเอียด ถ้าในสถานการณ์ปัจจุบันสมมุติว่าเอเยนต์มีตั๋วเครื่องบินชนิดคืนเงินไม่ได้เที่ยวบินออกจากเมืองบูคาเรสท์ในวันรุ่งขึ้น กรณีเช่นนี้เอเยนต์ต้องสร้างเป้าหมายให้กับตัวเองว่าจะต้องเดินทางไปเมืองบูคาเรสท์ให้ได้ การกระทำใด ๆ ที่ทำให้ไปถึงเมืองบูคาเรสท์ไม่ทันเวลาจะต้องถูกยกเลิก ไม่ต้องนำมาพิจารณาอีกต่อไป ปัญหาการตัดสินใจของเอเยนต์จะง่ายขึ้นมาก เพราะเป้าหมาย (Goal) เป็นตัวรวบรวมพฤติกรรม โดยจำกัดวัตถุประสงค์ (Objective) ของเอเยนต์ให้แคบเข้า

เราสามารถมองว่าเป้าหมายคือเซตของสถานะ (State) ในโลก (โลกในที่นี้มีขอบเขตแค่เรื่องที่เอเยนต์กำลังสนใจศึกษาอยู่เท่านั้น ไม่ใช่โลกทั้งโลกจริง ๆ) และเป็นเฉพาะสถานะที่เป็นไปเพื่อเป้าหมายเท่านั้น งานของเอเยนต์คือหาลำดับของการกระทำที่จะนำไปสู่สถานะเป้าหมาย (Goal state) แต่ก่อนที่จะหาลำดับของการกระทำเช่นนี้ได้ ต้องทราบก่อนว่าการกระทำนั้นต้องทำอะไร และสถานะที่ว่าไว้คืออะไร

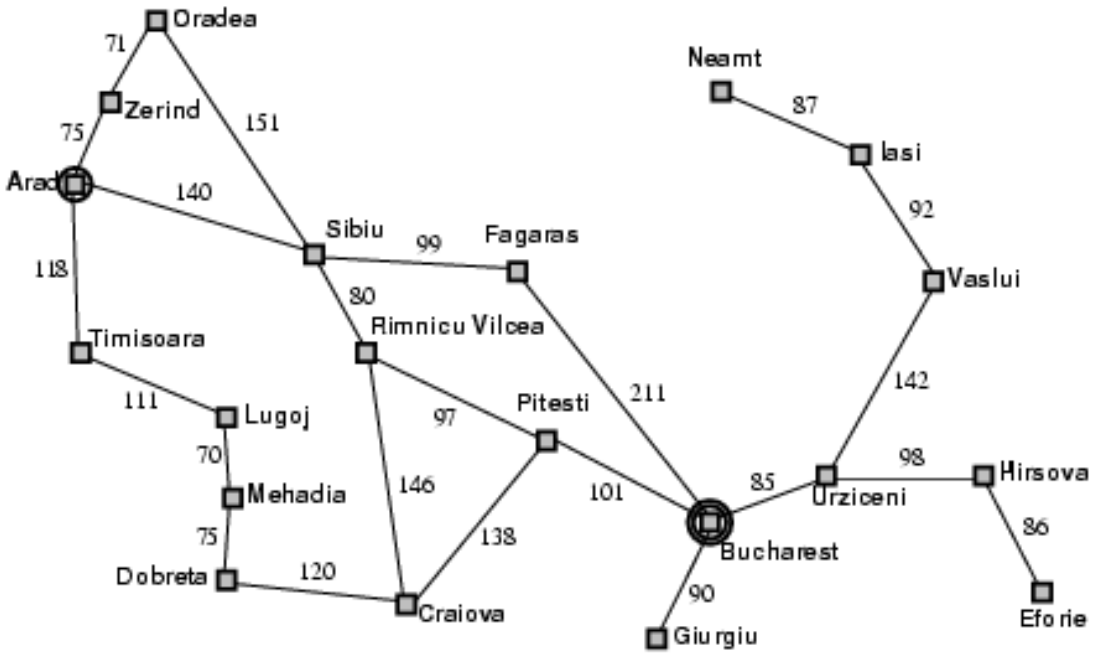
3.1.2 สร้างข้อปัญหา (Problem formulation) คือกระบวนการตัดสินใจหาว่าสิ่งใดคือการกระทำที่จะนำมาใช้ได้ และสิ่งใดคือสถานะที่ต้องพิจารณา เพื่อนำไปสู่การบรรลุเป้าหมาย

จากตัวอย่าง กำหนดให้การกระทำที่เอเยนต์ต้องนำมาพิจารณาได้แก่การขับรถจากเมืองหนึ่งไปยังอีกเมืองหนึ่ง (โดยไม่พิจารณาการกระทำรายละเอียดปลีกย่อย เช่น เลื่อนเท้าไปข้างหน้า 1 นิ้ว หรือหมุนพวงมาลัยไปทางซ้าย 1 องศา เพราะเป็นเรื่องละเอียดเกินไปจนอาจทำให้เดินทางออกจากเมืองไม่สำเร็จก็ได้) ส่วนสถานะได้แก่การอยู่ ณ เมืองหนึ่ง ๆ

3.1.3 ค้นหาลำดับการกระทำ (Search) เป็นกระบวนการค้นหาลำดับของการกระทำที่จะนำไปสู่เป้าหมาย เริ่มจากสถานะเริ่มต้น (Start state) จนถึงสถานะเป้าหมาย (Goal state) เป็นลำดับไป

จากตัวอย่าง เมื่อเอเยนต์กำหนดเป้าหมายและสถานะได้ตามข้อ 3.1.2 แล้ว ทราบว่าเป้าหมายคือการขับรถไปเมืองบูคาเรสต์ เอเยนต์ก็เริ่มพิจารณาว่าจะขับไปที่ใด เริ่มต้นจากที่เอเยนต์อยู่ปัจจุบัน คือเมืองอารัด มีถนน 3 สายออกจากเมือง สายหนึ่งไปเมืองซิบิว สายหนึ่งไปเมืองทิมิโซรา และอีกสายหนึ่งไปเมืองเซรินด์ ยังไม่มีสายใดไปสู่เป้าหมาย เอเยนต์ก็ยังไม่รู้ว่าควรจะไปต่อในเส้นทางใด กล่าวอีกนัยหนึ่งคือ เอเยนต์ยังไม่รู้ว่าขับรถไปเส้นทางใดเป็นการกระทำที่ดีที่สุด เพราะไม่มีความรู้พอจะบอกได้ว่าผลจากการกระทำแต่ละอย่างนั้นเป็นอย่างไร หากไม่ได้รับความรู้เพิ่มเติม อาจต้องหยุดอยู่กับที่ ทางที่ดีที่สุดคือเลือกมาทางหนึ่งอย่างเดาสุ่ม

แต่สมมุติว่าเอเยนต์มีแผนที่โรมาเนียอยู่แล้ว ดังรูปที่ 3.1 เอเยนต์จะมีข้อมูลเพียงพอที่จะบอกได้ว่าตนเองอยู่ในสถานะใด และจะมีการกระทำอย่างไรได้บ้าง เอเยนต์สามารถใช้ข้อมูลนี้พิจารณาหาลำดับของสถานะต่อไปในการเดินทางจากเมืองแต่ละเมืองดังกล่าว แล้วหาเส้นทางที่มุ่งสู่เมืองบูคาเรสต์ได้ในที่สุด ทันทีที่หาเส้นทางจากเมืองอารัดไปบูคาเรสต์บนแผนที่ได้ ก็ทำตนให้บรรลุเป้าหมายได้โดยเริ่มต้นการกระทำไปตามนั้น ลำดับของการกระทำนี้เรียกว่าหนทางแก้ปัญหาหรือคำตอบ (Solution) เป็นสิ่งที่ได้มาจากอัลกอริทึมของการค้นหา



รูปที่ 3.1 ภาพแสดงแผนที่บอกเส้นทางอย่างง่ายในโรมาเนีย

3.1.4 การดำเนินการ (Execution phase) หลังจากพบวิธีแก้ปัญหาแล้ว นำการกระทำเหล่านั้นมากระทำตามลำดับ ทำให้เกิดเป็นขั้นตอนในการดำเนินการ

การออกแบบเอเยนต์แก้ปัญหาเช่นนี้เรียกว่าการออกแบบชนิด “กำหนดสร้าง หาทางแก้ ดำเนินการ” (“Formulate, search, execute” design)

ปัญหาและวิธีแก้ปัญหาที่กำหนดสร้างขึ้นมาตามที่กล่าวมานี้ มีองค์ประกอบ 4 ข้อคือ

1. สถานะเริ่มต้น (Initial state) เป็นสถานะแรกของเอเยนต์ เช่นจากตัวอย่าง สถานะแรกของเอเยนต์ในโรมาเนียคือเมืองอารัด เขียนในรูปฟังก์ชันได้ว่า In(Arad)

2. การกระทำ (Action) ที่เอเยนต์กระทำได้ เขียนเป็นฟังก์ชันเรียกว่า Successor function บ่งบอกถึงการกระทำและสถานะใหม่ของเอเยนต์ที่เป็นผลมาจากการกระทำนั้น ถ้ากำหนดสถานะ x แล้ว SUCCESSOR-FN(x) จะคืนค่าเป็นเซตของคู่ลำดับ <action, successor> โดยที่แต่ละ Action คือการกระทำที่ทำกับสถานะ x ได้ ส่วน Successor คือสถานะที่เกิดขึ้นต่อจากสถานะ x หลังจากมีการทำ action ดังกล่าวแล้ว เช่น ต่อจากสถานะ In(Arad) Successor function จะคืนค่าเป็นเซตของคู่ลำดับดังนี้

{ <Go(Sibiu), In(Sibiu) >, < Go(Timisoara), In(Timisoara) >, < Go(Zerind), In(Zerind) > }

สถานะเริ่มต้นกับการกระทำของเอเยนต์รวมกันก่อให้เกิดเป็นเซตของสถานะทุกสถานะที่สามารถไปถึงได้โดยเริ่มจากสถานะเริ่มต้น เรียกเซตนี้ว่า State space ของปัญหา State space เขียนได้ในรูปกราฟที่ประกอบด้วยโหนดเป็นสถานะต่าง ๆ เส้นที่เชื่อมระหว่างโหนด 2 โหนด แสดงถึงการกระทำ เส้นทาง (Path) ต่าง ๆ ใน State space ก็คือชุดของสถานะที่เชื่อมกันด้วยชุดของการกระทำ

3. การทดสอบเป้าหมาย (Goal test) คือการหาว่าสถานะที่ให้มานั้นเป็นสถานะเป้าหมายใช่หรือไม่ บางครั้งสถานะเป้าหมายอาจมีอยู่มากกว่าหนึ่งจำนวน คือเป็นเซตของสถานะเป้าหมายที่ประกาศออกมาให้ทราบอย่างชัดเจนก่อนแล้ว ถ้าเป็นเช่นนี้ก็เพียงแค่ตรวจสอบว่าสถานะที่ให้มาเป็นหนึ่งในเซตนั้นหรือไม่ เช่นเป้าหมายของเอเยนต์ในโรมาเนียมีเพียงค่าเดียวอยู่ในเซต นั่นคือ {In(Bucharest)} แต่บางครั้งเป้าหมายมีลักษณะที่เป็นนามธรรม ทำให้การสร้างเซตเป้าหมายที่ชัดเจนเป็นไปได้ยาก ตัวอย่างเช่นการเล่นหมากรุก เป้าหมายคือต้องการบรรลุสถานะ “ฆาต” (Checkmate) นั่นคือ ตัวขุนของฝ่ายตรงข้ามถูกโจมตี และไม่สามารถหนีพ้นได้ ถือเป็นจุดแพ้ชนะของเกมหมากรุก

4. Path cost function เป็นฟังก์ชันกำหนดค่าใช้จ่าย (หรือต้นทุน) ให้กับแต่ละเส้นทาง เอเยนต์จะเลือกฟังก์ชันค่าใช้จ่ายที่สอดคล้องกับเกณฑ์การวัดสมรรถนะ เช่น เอเยนต์ที่ต้องการไปถึงเมืองบูคาเรสต์ เวลาเป็นสิ่งสำคัญ ดังนั้นค่าใช้จ่ายแต่ละเส้นทางจึงควรจะเป็นระยะทางที่วัดเป็นกิโลเมตร (ค่าใช้จ่ายในที่นี้ไม่ได้หมายความถึงจำนวนเงินเสมอไป) และค่าใช้จ่ายของเส้นทางในที่นี้หมายถึงผลรวมค่าใช้จ่ายของการกระทำต่าง ๆ ที่อยู่ในเส้นทางนั้น

5. คำตอบของปัญหา คือหนทางแก้ปัญหา (Solution) หมายถึงเส้นทางตั้งแต่สถานะเริ่มต้นไปจนถึงสถานะเป้าหมาย คุณภาพของคำตอบหรือหนทางแก้ปัญหานี้วัดได้จากค่า Path cost ในกรณีที่มีทางแก้ปัญหาหลายทาง ถ้าเป็นทางแก้ปัญหาที่ดีที่สุด (Optimal solution) แล้ว จะมีค่า Path cost ต่ำที่สุดเมื่อเทียบกับ Path cost ของทางแก้ปัญหาอื่น ๆ

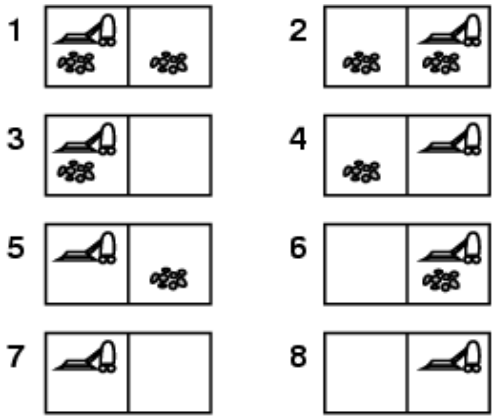
3.2 ปัญหาตัวอย่าง

วิธีแก้ปัญหาตามที่กล่าวมาแล้วในหัวข้อ 3.1 สามารถนำมาประยุกต์ใช้กับปัญหาที่มีสภาพแวดล้อมในงาน (Task environment) ลักษณะต่าง ๆ ได้ (ดูบทที่ 2) ในที่นี้จะยกตัวอย่างปัญหาที่แตกต่างกัน 2 ประเภท คือ ปัญหาของเล่น (Toy problem) กับปัญหาของจริง (Real-world problem) เช่นปัญหาเครื่องดูดฝุ่นอย่างง่าย (ในบทที่ 2) ปัญหาเกมบางอย่าง ส่วนปัญหาของจริงเป็นปัญหาที่ต้องสนใจวิธีการแก้ไขอย่างจริงจัง อาจจะไม่มีคำบรรยายชัดเจน แต่ก็จะได้เห็นตัวอย่างของการสร้างปัญหาจริง

3.2.1 ปัญหาของเล่น เป็นปัญหาที่ใช้แสดงหรือใช้เป็นแบบฝึกหัดในการทำวิธีแก้ปัญหาแบบต่าง ๆ หลายแบบ สามารถบรรยายตัวปัญหาได้ชัดเจน เป็นปัญหาที่ดูง่าย ศึกษาง่าย

3.2.1.1 ปัญหาเครื่องดูดฝุ่นอย่างง่าย (จากบทที่ 2) สร้างตัวปัญหาได้ดังนี้

1. สถานะ (State) เอเยนต์มีสถานที่อยู่ 2 แบบ คือช่องที่มีฝุ่น กับช่องที่ไม่มีฝุ่น สถานะทั้งหมดที่เป็นไปได้คือ $2 \times 2^2 = 8$ สถานะ ดังในรูปที่ 3.2



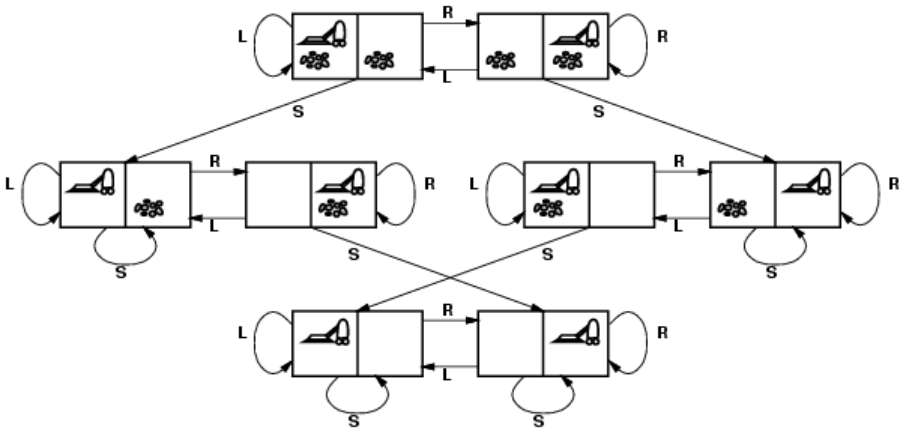
รูปที่ 3.2 สถานะของเครื่องดูดฝุ่นที่เป็นไปได้ทั้งหมด 8 สถานะ

2. สถานะเริ่มต้น (Initial state) สามารถกำหนดให้สถานะใด ๆ เป็นสถานะเริ่มต้นก็ได้

3. Successor function เป็นฟังก์ชันที่สร้างสถานะใหม่โดยการทดลองการกระทำ 3 อย่างคือ Left, Right, Suck ทำให้ได้ State space ตามรูปที่ 3.3

4. Goal test ทดสอบว่าทุกช่องสะอาดแล้วใช่หรือไม่

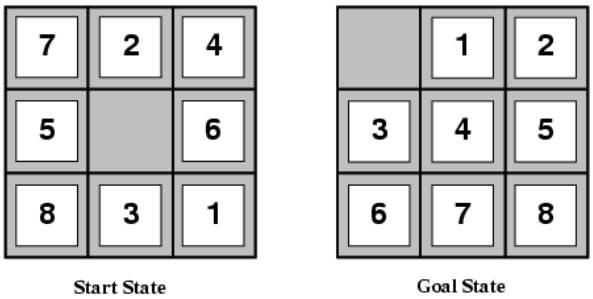
5. Path cost เมื่อมีการทำงานแต่ละขั้นตอนคิดค่าใช้จ่ายเป็น 1 หน่วย ดังนั้น Path cost จึงเป็นจำนวนขั้นตอนที่อยู่ในเส้นทางการแก้ปัญหา



รูปที่ 3.3 State space ของเครื่องดูดฝุ่น L=Left, R=Right, S=Suck

เปรียบเทียบกับในโลกจริง ปัญหาของเล่นเช่นนี้มีตำแหน่งที่อยู่ของเครื่องดูดฝุ่นจำกัดเพียง 2 ตำแหน่งเท่านั้น (ช่องทางซ้ายกับช่องทางขวา) ฝุ่นมีจำกัด (มีกับไม่มี) การดูดฝุ่นเชื่อถือได้ คือเมื่อดูดแล้วต้องสะอาด เมื่อทำความสะอาดแล้วจะไม่มีฝุ่นจับอีก สถานะที่เห็นถูกกำหนดในเทอมของตำแหน่งเครื่องดูดฝุ่นกับตำแหน่งของฝุ่น ซึ่งมีอย่างละ 2 แบบ แต่ถ้าสภาพแวดล้อมใหญ่กว่านี้ เช่น มี n ตำแหน่ง จะมีสถานะจำนวน $n.2^n$ สถานะ

3.2.1.2 เกม 8-puzzle เป็นเกมที่ใช้กระดานแบ่งเป็นตารางขนาด 3x3 ช่อง มีแผ่นที่กำกับด้วยตัวเลข 1-8 อยู่ 8 แผ่น และมีช่องว่าง 1 ช่อง แผ่นที่อยู่ติดกับช่องว่างสามารถเลื่อนย้ายไปอยู่ในช่องว่างนั้นแทน จุดประสงค์ของเกมคือต้องการเลื่อนช่องต่าง ๆ ไปจนมีลักษณะตามที่เป้าหมายตั้งไว้เช่นในรูปที่ 3.4



รูปที่ 3.4 ตัวอย่างของเกม 8-puzzle

1. สถานะ เกมนี้บอกสถานะของเกมโดยระบุตำแหน่งการวางแผ่นทั้ง 8 แผ่น รวมทั้งตำแหน่งของช่องว่างในกระดาน

2. สถานะเริ่มต้น คือสถานะใด ๆ ที่กำหนดให้เป็นจุดเริ่มของเกม

3. Successor function เป็นฟังก์ชันที่สร้างสถานะใหม่โดยลองจากการกระทำ 4 อย่างคือการเลื่อนช่องว่างไปซ้าย ขวา ขึ้น หรือลง

4. Goal test ทดสอบว่าสถานะนั้นตรงกันกับเป้าหมายที่กำหนดไว้หรือไม่ เช่นในรูป 3.4 สถานะที่มีลักษณะตรงกันกับ Goal state แสดงว่าบรรลุเป้าหมาย

5. Path cost แต่ละขั้นตอนที่มีการขยับแผ่นว่างคิดค่าใช้จ่ายเท่ากับ 1 หน่วย ดังนั้น Path cost ในที่นี้จึงเป็นจำนวนขั้นตอนในเส้นทางการแก้ปัญหา

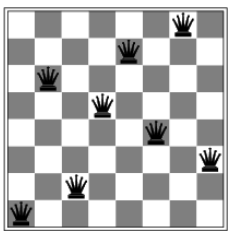
3.2.1.3 ปัญหา 8-queens คือปัญหาการวางตัวควีน 8 ตัวลงในกระดานหมากรุกโดยไม่ให้ควีนตัวใด ๆ อยู่ในตำแหน่งที่จะถูกควีนตัวอื่นโจมตีได้ (ควีนในหมากรุกฝรั่งสามารถจู่โจมระยะไกลในแถวเดียวกัน หรือคอลัมน์เดียวกัน หรือแนวทะแยงมุมก็ได้) ดูตัวอย่างการวางหมากได้จากรูปที่ 3.5 (รูปนี้ยังไม่ใช่คำตอบของปัญหา) ความจริงปัญหานี้มีอัลกอริทึมที่มีประสิทธิภาพในการแก้ปัญหาอยู่แล้ว แต่ก็ยังคงเป็นปัญหาทดลองที่น่าสนใจที่ใช้เพื่อการศึกษาอัลกอริทึมของการค้นหา วิธีสร้างปัญหานี้มี 2 แบบ แบบแรกเริ่มต้นจากสถานะว่างเปล่า คือกระดานเปล่า การกระทำแต่ละแอคชันคือการวางควีนลงบนกระดาน ส่วนอีกแบบหนึ่งคือเริ่มต้นจากกระดานที่มีควีนวางอยู่ 8 ตัว แล้วค่อยย้ายควีนเหล่านั้นไปทั่ว ๆ ทั้งสองวิธีไม่คำนึงถึง Path cost เพราะต้องการแต่สถานะสุดท้ายเท่านั้น

1. สถานะ การจัดวางควีนตั้งแต่ 0-8 ตัวลงในช่องกระดานรูปแบบใด ๆ ก็ตามคือสถานะ

2. สถานะเริ่มต้น คือกระดานที่ยังไม่วางควีนลงไปเลย

3. Successor function ได้แก่การวางควีนลงไปในช่องกระดานช่องที่ว่างอยู่

4. Goal test ทดสอบว่าเป็นสถานะที่มีควีน 8 ตัวในกระดาน ไม่มีตัวใดอยู่ในตำแหน่งที่ถูกโจมตีจากตัวอื่น



รูปที่ 3.5 กระดาน 8-Queens

3.2.2 ปัญหาของจริง (Real-world problems)

ปัญหาที่พบในโลกจริงที่ได้เห็นมาแล้วเป็นปัญหาประเภทหาเส้นทางซึ่งระบุสถานที่และเส้นทางที่เชื่อมต่อเมืองแต่ละเมือง อัลกอริทึมที่ใช้หาเส้นทางมักนำมาใช้ในงานแอพพลิเคชันหลากหลาย เช่นการหาเส้นทางในเครือข่ายคอมพิวเตอร์ การวางแผนดำเนินงานทางทหาร ระบบวางแผนเดินทางของสายการบิน เป็นต้น ปัญหาเหล่านี้จะเป็นเรื่องที่ซับซ้อนขึ้น

3.2.2.1 ปัญหาการเดินทางโดยสายการบิน

สถานะ : แต่ละสถานะแทนด้วยสถานที่ (เช่นสนามบิน) และเวลาในปัจจุบัน

สถานะเริ่มต้น : แต่ละปัญหาจะกำหนดสถานะเริ่มต้นเอง

Successor function : เป็นฟังก์ชันที่ให้ค่าเป็นสถานะที่เกิดจากการบินตามกำหนดการ พร้อมทั้งบอกเวลาออกเดินทาง เวลาถึงเป้าหมาย จากสนามบิน ณ ปัจจุบันไปอีกสนามบินหนึ่ง

Goal test : เอเยนต์อยู่ที่จุดหมายภายในเวลาที่กำหนดหรือไม่

Path cost : ขึ้นอยู่กับค่าใช้จ่ายในเทอมต่าง ๆ ที่เกี่ยวข้อง เช่น เวลารอคอย เวลาที่บิน กระบวนการตรวจคนเข้าเมือง คุณภาพของที่นั่ง เวลาในช่วงวัน ประเภทของเครื่องบิน การสะสมระยะทางบินเพื่อรับรางวัล เป็นต้น

3.2.2.2 ปัญหาการเดินทางของพนักงานขาย (Traveling salesperson problem, TSP) เป็นปัญหาการเดินทางซึ่งต้องไปเยี่ยมแต่ละเมืองเพียงครั้งเดียวเท่านั้น เป้าหมายคือต้องหาเส้นทางที่สั้นที่สุด อัลกอริทึมของปัญหา TSP ถูกนำไปใช้ในงานต่าง ๆ เช่นการวางแผนเจาะแผ่นบอร์ดวงจรไฟฟ้าด้วยเครื่องเจาะอัตโนมัติ

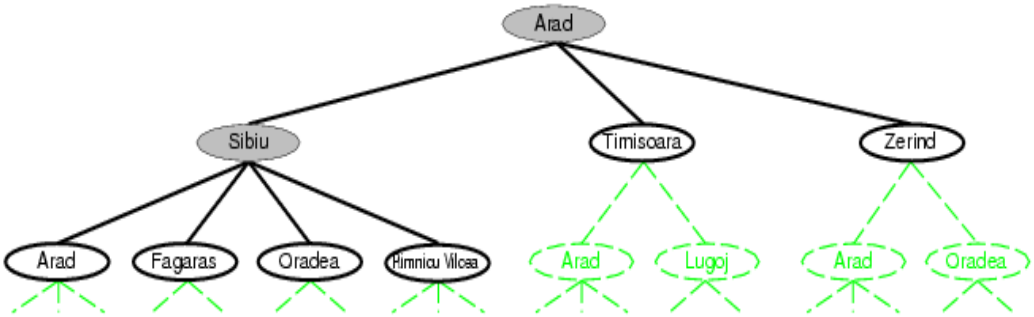
3.2.2.3 Robot navigation เป็นปัญหาการหาเส้นทางแบบทั่วไป หุ่นยนต์สามารถเคลื่อนที่ได้เป็นเส้นทางต่อเนื่อง นั่นคือมีชุดของการกระทำและสถานะไม่จำกัด จึงต้องหาเทคนิคใหม่ ๆ และทันสมัยเพื่อหาทางจำกัด State space ของการค้นหา

3.3 การหาทางแก้ปัญหา

หลังจากกำหนดตัวปัญหาขึ้นมาแล้ว สิ่งที่ต้องทำต่อไปคือการแก้ปัญหา การแก้ปัญหาทำได้โดยการค้นหาไปใน State space เทคนิคที่ใช้ในการค้นหามีอยู่เป็นจำนวนมาก แต่ในบทนี้จะได้กล่าวถึงเทคนิคการค้นหาแบบต่าง ๆ ใน Search tree ซึ่งสร้างขึ้นโดยเริ่มจากสถานะเริ่มต้นเป็น

โหนดราก แล้วสร้างโหนดต่อมาด้วย Successor function จึงเห็นได้ว่าโหนดที่อยู่ใน Search tree นี้ทั้งหมดล้วนแต่เป็นสถานะที่อยู่ใน State space ทั้งสิ้น

ตัวอย่างของ Search tree ในปัญหาการเดินทางในโรมาเนียแสดงได้ดังรูปที่ 3.6 สถานะเริ่มต้นที่เป็นรากได้แก่เมืองอารัด โหนดลูกของเมืองอารัดหาได้จาก Successor function คือการเดินทางไปยังอีกเมืองหนึ่ง ซึ่งทำให้สร้าง (Generate) โหนดใหม่ได้ 3 โหนดคือ Sibiu, Timisoara, และ Zerind การสร้างโหนดใหม่จากโหนดเดิมเช่นนี้เรียกอีกอย่างหนึ่งว่าการ expand states



รูปที่ 3.6 ตัวอย่าง Search tree (บางส่วน) ของปัญหาการเดินทางในโรมาเนีย

จากรูปที่ 3.6 หลังจากโหนดรากสร้างโหนดลูกมา 3 โหนดเพื่อแทน 3 สถานะแล้ว การแก้ปัญหาจะต้องเลือกเพียงหนึ่งสถานะที่คิดว่าดีที่สุดเพื่อนำมา Expand ต่อไป เช่นถ้าเลือกสถานะเป็น Sibiu เมื่อ Expand แล้วจะสร้างโหนดลูกต่อไป 4 สถานะ คือ Arad, Fagaras, Oradea, Rimnicu Vilcea แล้วตรวจสอบว่าทั้ง 4 สถานะนี้เป็น Goal state หรือไม่ เมื่อไม่ใช่ ก็ย้อนกลับไปเลือกสถานะ Timisoara หรือ Zerind ขึ้นอยู่กับวิธีการค้นหา แล้ว Expand สถานะที่ได้รับเลือก ทำเช่นนี้ไปจนกว่าจะพบทางแก้ปัญหา หรือไม่ก็ไม่มีสถานะใดให้ Expand ได้อีก

การเลือกสถานะเพื่อทำการ Expand ต้องใช้วิธีการค้นหา (Search strategy) มาเป็นตัวตัดสินใจ วิธีการต่าง ๆ ที่กล่าวถึงในบทนี้เป็นแบบไม่อาศัยข้อมูลอื่นใดของปัญหา เรียกว่า Uninformed search strategies

โครงสร้างข้อมูลของโหนดประกอบด้วยองค์ประกอบ 5 อย่างคือ

1. สถานะ เป็นสถานะใดสถานะหนึ่งที่อยู่ใน State space
2. โหนดแม่ (Parent node) คือโหนดใน Search tree ที่เป็นผู้สร้างโหนดนั้น ๆ
3. การกระทำ (Action) คือการกระทำที่มีต่อโหนดแม่เพื่อทำให้เกิดโหนดนั้น

4. Path cost ค่าใช้จ่าย แทนด้วยฟังก์ชัน g(n) เป็นค่าใช้จ่ายเริ่มตั้งแต่สถานะเริ่มต้นจนถึงโหนดที่ n

5. ความลึก (Depth) หมายถึงระดับความลึกของ Tree ที่นับจากราก (สถานะเริ่มต้น) ลงมาตามเส้นทางจนกระทั่งมาถึงโหนดนั้น

การแก้ปัญหาจาก Search tree จะทำได้ดีมีประสิทธิภาพหรือไม่นั้น สามารถวัดได้หลายทาง เกณฑ์การวัดประสิทธิภาพที่จะพิจารณาในที่นี้มี 4 ข้อ ได้แก่

1. Completeness ดูว่าอัลกอริทึมของการค้นหานั้นทำให้หาทางแก้ปัญหา (Solution) ได้หรือไม่

2. Optimality วิธีการค้นหานั้นทำให้ได้ทางแก้ปัญหาที่ดีที่สุดหรือไม่

3. Time complexity ใช้เวลาในการแก้ปัญหานานเท่าใด

4. Space complexity ใช้เนื้อที่หน่วยความจำเท่าใดในการค้นหา

ในการพิจารณา Time และ Space วิธีการของทฤษฎีทางคอมพิวเตอร์มักจะใช้วิธีวัดจากขนาดของ State space ที่จัดตัวเป็นรูปกราฟ เพราะสามารถมองรูปกราฟได้จากโครงสร้างข้อมูล หรืออีกนัยหนึ่งคืออินพุตของโปรแกรมการค้นหา แต่ AI เมื่อพูดถึงกราฟ มักแทนด้วยสถานะเริ่มต้นกับ Successor function และวัดความซับซ้อนด้วยปริมาณ 3 อย่าง คือ

1. Branching factor (b) ซึ่งหมายถึงจำนวนของโหนดลูก (Successor) สูงสุดของโหนดใด ๆ

2. d คือระดับความลึกของสถานะเป้าหมายที่อยู่ในระดับความลึกน้อยที่สุด (อยู่ตื้นที่สุด)

3. m คือระดับความลึกสุดของ Path ใน State space

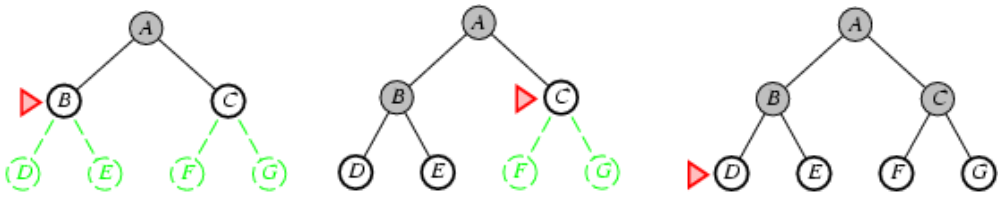
3.4 วิธีค้นหาแบบไม่ใช้ข้อมูล (Uninformed search strategies)

วิธีค้นหาแบบไม่ใช้ข้อมูล บางครั้งเรียกว่า Blind search เป็นการค้นหาโดยไม่มีข้อมูลมาเป็นเครื่องช่วยตัดสินใจ อัลกอริทึมของการค้นหาจะ Expand ให้เกิดโหนดลูก (Successor) ขึ้นแล้วดูว่าโหนดนั้นเป็นสถานะเป้าหมายหรือไม่ การค้นหามีหลายวิธีแตกต่างกันไปตามลำดับของโหนดที่ได้รับการ Expand สำหรับวิธีค้นหาที่อาศัยข้อมูลช่วย เรียกว่า Informed search หรือ Heuristic search จะกล่าวถึงในบทที่ 4

3.4.1 Breadth-first search

เป็นวิธีค้นหาที่ง่ายที่สุด โหนดรากจะถูก Expand เป็นลำดับแรก แล้วโหนดลูกทุกโหนดของโหนดรากจะถูก Expand เป็นลำดับต่อไป แล้ว Expand Successors ของโหนดลูกเหล่านี้ ต่อไปอีกเรื่อย ๆ กล่าวโดยทั่วไปคือ ทุก ๆ โหนดที่อยู่ในความลึกระดับเดียวกันของ Search

tree จะถูก Expand จนครบหมดก่อนโหนดใด ๆ ที่อยู่ในความลึกระดับต่อไปตามลำดับ ดูตัวอย่างได้ในรูปที่ 3.7



รูปที่ 3.7 Breadth-first search ที่ใช้กับไบนารีทรี แสดงการ Expand ตามจุดที่ชี้ไว้

การประเมินค่า Breadth-first search กระทำโดยพิจารณาจากเกณฑ์การวัดประสิทธิภาพ 4 ข้อ ดังนี้

1. Completeness วิธีค้นหาแบบนี้มีลักษณะ Complete เพราะถ้าโหนดที่เป็น Goal state อยู่ในชั้นที่ตื้นที่สุดของ Tree ในระดับความลึก d แล้ว Breadth-first search จะสามารถหาโหนดนี้พบได้ในที่สุด หลังจากที่ Expand ทุกโหนดที่อยู่ในระดับตื้นกว่านี้ไปหมดแล้ว (d-1 ระดับ)

2. Optimality สำหรับ goal state ที่อยู่ในระดับชั้นที่ตื้นที่สุดของ Search tree ไม่จำเป็นต้องเป็นโหนดที่ดีที่สุดเสมอไป แต่ถ้า Path cost เป็นฟังก์ชันที่ไม่มีค่าลดลง (Non-decreasing function) ไปตามความลึกของ Tree แล้ว จะพบว่า เป้าหมายที่ได้นั้นเป็นโหนดหรือสถานะที่ดีที่สุด

3. Time complexity, Space complexity วิธี Breadth-first search เป็นวิธีที่ใช้เวลาและเนื้อที่หน่วยความจำจำนวนมาก ตัวอย่างเช่น สมมุติให้ทุกสถานะใน State space มีโหนดลูกหรือ Successor จำนวน b โหนด ดังนั้นในระดับแรกของ Tree โหนดรากจะสร้างโหนดลูก b โหนด ซึ่งแต่ละโหนดสร้างโหนดลูกขึ้นมาอีก b โหนดเช่นกัน จึงเกิดโหนดจำนวน b^2 โหนดในระดับที่สอง แต่ละโหนดในระดับที่สองนี้ก็สร้างโหนดลูกอีก b โหนด ทำให้เกิดโหนดจำนวน b^3 โหนดในระดับที่สาม และเป็นเช่นนี้เรื่อยไป สมมุติว่าคำตอบอยู่ที่ระดับความลึก d กรณีที่แย่ที่สุด (Worst case) จะต้อง Expand โหนดไปทุกโหนดยกเว้นโหนดสุดท้ายของระดับความลึก d (ในกรณีที่แย่ที่สุดโหนดสุดท้ายจะเป็นโหนดเป้าหมาย) นั่นคือสร้างโหนดขึ้นมาจำนวน b^{d+1} - b โหนดที่ระดับความลึก d+1 จำนวนโหนดรวมทั้งหมดที่สร้างขึ้นมาคือ

$$b + b^2 + b^3 + \ldots + b^d + (b^{d+1} - b) = O(b^{d+1})$$

ทุกโหนดที่สร้างขึ้นมานี้ต้องอยู่ในหน่วยความจำ เพราะต้องนำมาพิจารณาหรือไม่ก็เป็นบรรพบุรุษของโหนดที่จะต้องได้รับการพิจารณาต่อไป จะตัดทิ้งไปไม่ได้ การคิด Space complexity ก็เช่นเดียวกันกับ Time complexity (แต่ต้องเพิ่มโหนดรากเข้าไปด้วย)

ลักษณะของฟังก์ชัน $O(b^{d+1})$ จะเพิ่มขึ้นแบบเอ็กซ์โพเนนเชียล ตัวอย่างเช่นถ้าให้โหนดใด ๆ ของ Breadth-first search มี Branching factor (b)=10 ปัญหามีคำตอบอยู่ที่ความลึก d ใด ๆ เครื่องคอมพิวเตอร์สามารถสร้างโหนดได้ 10,000 โหนดใน 1 วินาที แต่ละโหนดใช้เนื้อที่เก็บ 1,000 ไบต์ (เครื่องคอมพิวเตอร์ที่ใช้เป็นเครื่อง PC ที่ทันสมัย) เวลาและหน่วยความจำที่ใช้จะเป็นดังตารางที่ 3.1

ตารางที่ 3.1 เวลาและหน่วยความจำที่ใช้ใน Breadth-first search เมื่อกำหนด Branching factor b=10; ความเร็ว 1,000 โหนด/วินาที และหน่วยความจำ 1,000 ไบต์/โหนด

ความลึก	จำนวนโหนด	เวลา	หน่วยความจำ
2	1100	.11 วินาที	1 megabyte
4	111,100	11 วินาที	106 megabytes
6	10^7	19 นาที	10 gigabytes
8	10^9	31 ชั่วโมง	1 terabyte
10	10^{11}	129 วัน	101 terabytes
12	10^{13}	35 ปี	10 petabytes
14	10^{15}	3,523 ปี	1 exabyte

จากตารางนี้ จะพบว่าหน่วยความจำที่ใช้ในการค้นหาก่อปัญหาใหญ่ได้มากกว่าเรื่องของเวลาที่ต้องใช้ คนเราอาจรอได้ถึง 31 ชั่วโมง หรือนานกว่านั้นก็ได้สำหรับปัญหาสำคัญที่มีคำตอบอยู่ที่ระดับความลึก 8 แต่คอมพิวเตอร์ที่มีหน่วยความจำหลักถึง 1 เทอราไบต์นั้นมีไม่มาก ดังนั้นวิธีการค้นหาแบบ Breadth-first search จึงมีข้อจำกัดที่สำคัญ

เวลาที่ใช้ในการค้นหาของ Breadth-first search ก็เป็นปัจจัยหนึ่งซึ่งทำให้วิธีการนี้ไม่น่าเลือกใช้ เช่นการต้องรอถึง 35 ปีกว่าจะได้คำตอบ

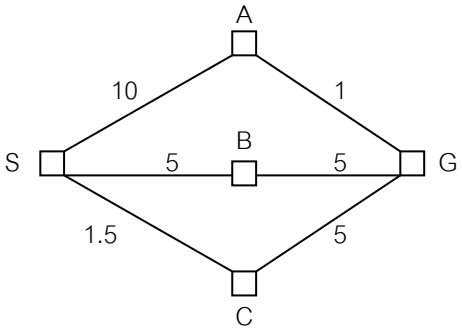
สรุปได้ว่าถ้าปัญหาในการค้นหามีความซับซ้อนที่พุ่งสูงแบบเอ็กซ์โปเนนเชียลแล้ว ไม่สามารถใช้วิธีค้นหาแบบไม่ใช้ข้อมูลได้ ยกเว้นปัญหาที่เล็กมาก ๆ

3.4.2 Uniform-cost search

วิธีนี้ปรับปรุงมาจากวิธีค้นหาแบบ Breadth-first search แต่มีการปรับอัลกอริทึมเพื่อให้หาค่าใช้จ่ายที่ดีที่สุดในทุกขั้นตอนได้ เนื่องจากวิธีแบบ Breadth-first search หาคำตอบที่ดีที่สุดได้เมื่อทุกระดับความลึกมีค่าใช้จ่ายเท่ากันหมด เพราะการ Expand โหนดทำจากโหนดที่อยู่ในระดับตื้นที่สุดก่อนเสมอ Uniform-cost search จึงเลือก expand โหนดที่มี Path cost หรือ g(n) ต่ำที่สุด แทนที่จะ Expand โหนดที่อยู่ชั้นตื้นที่สุด สังเกตุได้ว่า ถ้า Path cost เท่ากันทุกระดับความลึก Uniform-cost search ก็จะเหมือนกับ Breadth-first search

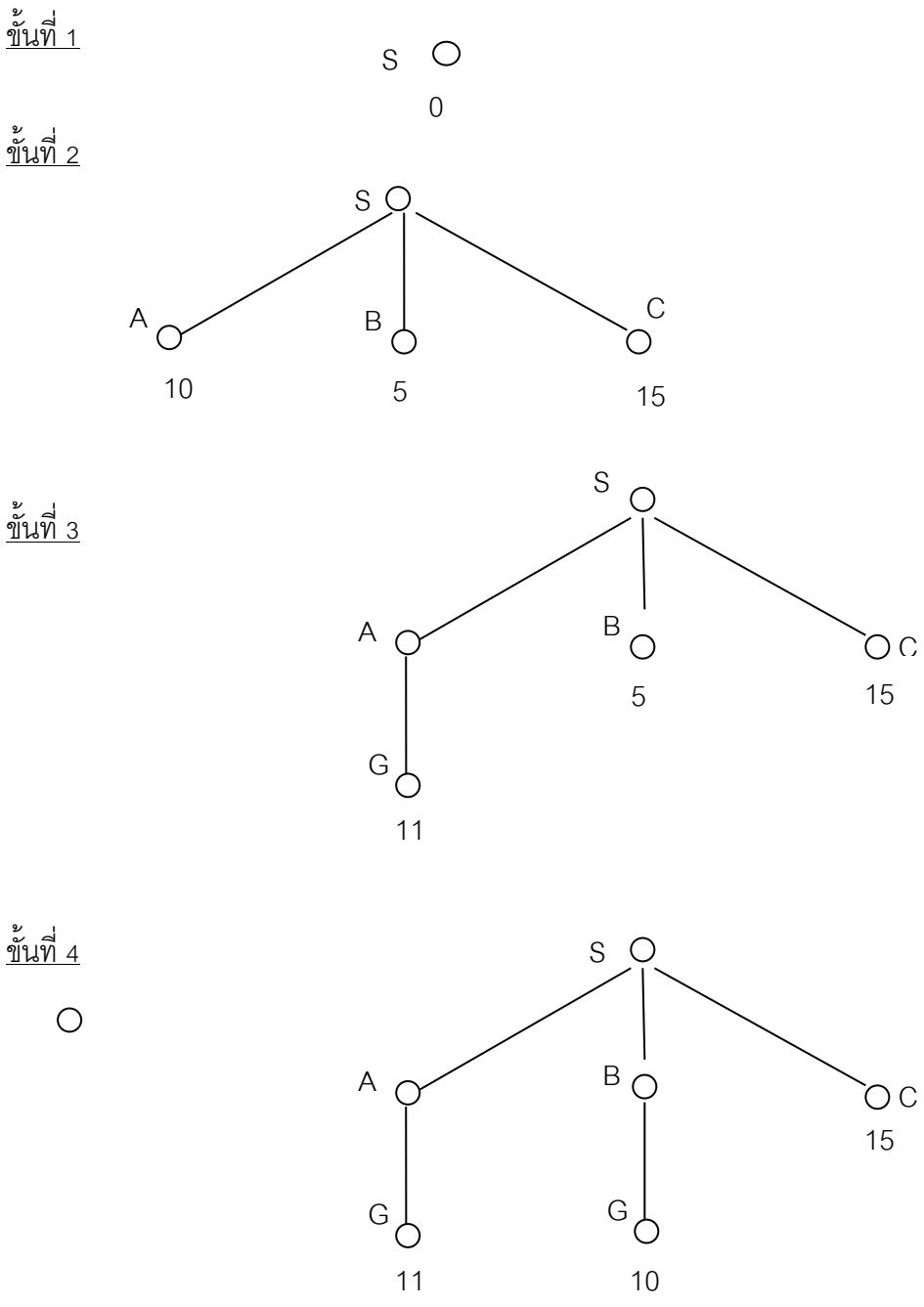
การค้นหาแบบ Uniform-cost search ไม่สนใจจำนวนขั้นตอนหรือจำนวนโหนดที่อยู่ในเส้นทาง แต่สนใจเส้นทางที่ทำให้ค่าใช้จ่ายต่ำสุดเท่านั้น ดังนั้นจึงอาจจะติดอยู่ในลูปตลอดกาลได้ ถ้ามีการ Expand โหนดแล้วได้โหนดที่เป็นสถานะเดิม แต่เนื่องจากโหนดเดิมนั้นมีค่าใช้จ่ายต่ำที่สุด ก็จะเลือก Expand โหนดนั้นตลอด ทำให้เกิดการวนลูปขึ้น แต่ถ้าค่าใช้จ่ายในเส้นทางนั้นเพิ่มขึ้นเรื่อย ๆ ไปตามระดับชั้นความลึกแล้ว วิธี Uniform-cost search จะหาคำตอบที่ดีที่สุดได้แน่นอน

พิจารณาตัวอย่างปัญหาการค้นหาเส้นทาง (Route-finding problem) ดังรูปที่ 3.8



รูปที่ 3.8 สิ่งที่กำหนดให้ในปัญหาการค้นหาเส้นทาง

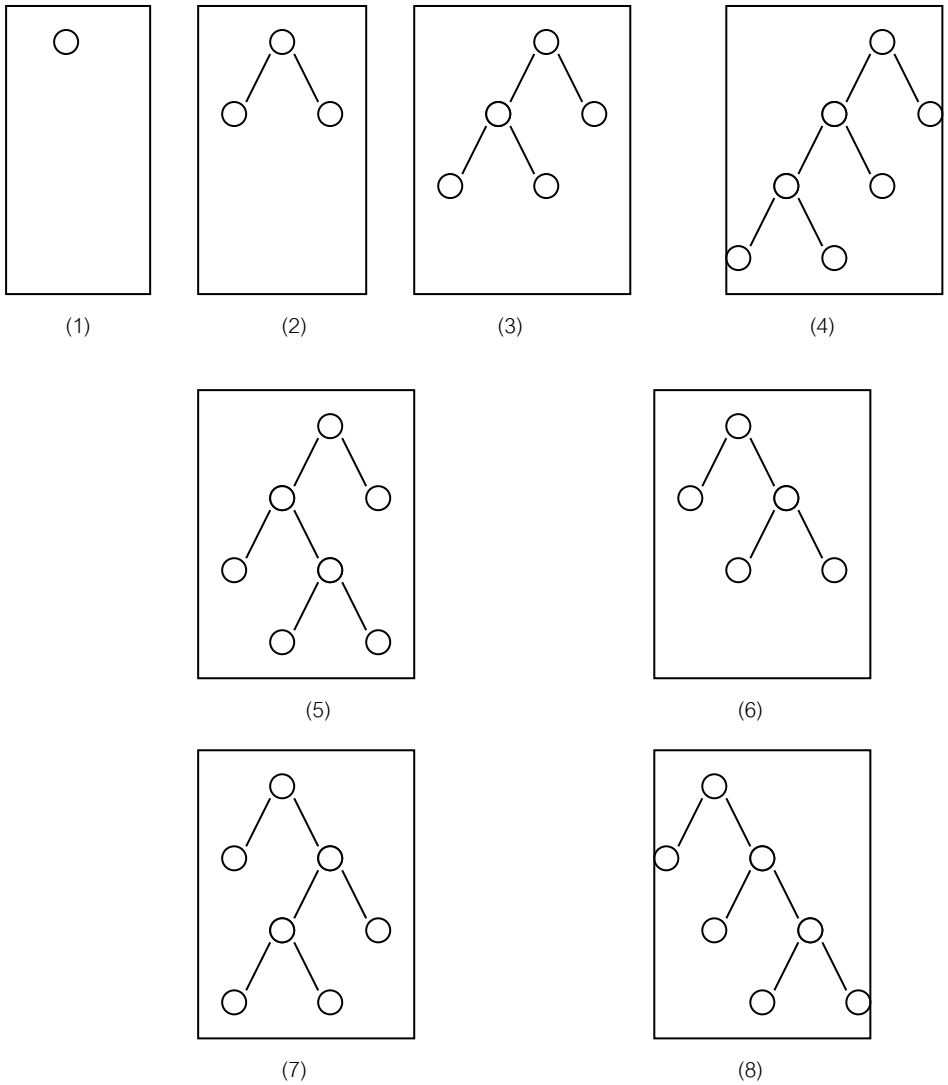
การค้นหาเป็นไปตามขั้นตอนที่แสดงในรูปที่ 3.9 จากรูปจะเห็นว่า State space แสดงค่าใช้จ่ายของแต่ละ Operator และจะมีการค้นหาได้ตามรูปต้นไม้ที่เห็น แต่ละโหนด มีค่า g(n) กำกับไว้ จะเห็นว่า Path SAG เป็นคำตอบแต่ไม่ใช่คำตอบที่ดีที่สุด (Optimal solution) จึงเลือกเส้นทางใหม่คือ SBG ซึ่งมีค่า Path cost ต่ำกว่าคือ 10 แทนที่จะใช้เส้นทางแรก



รูปที่ 3.9 Uniformed-cost search ใช้กับปัญหา Route-finding

3.4.3 Depth-first search

เป็นวิธีที่ Expand ไปจนถึงโหนดที่อยู่ลึกที่สุดในกิ่งของ Search tree เสมอ ถ้าพบจุดตัน (ไม่ใช่ Goal แต่ Expand ต่อไม่ได้แล้ว) จะย้อนกลับขึ้นมา Expand โหนดที่อยู่ในระดับตื้นขึ้น แสดงวิธีการทีละขั้นตอนได้ดังรูปที่ 3.10



รูปที่ 3.10 แผนภาพแสดงการค้นหาแบบ Depth-first search

ข้อดีของ Depth-first search คือ ใช้เนื้อที่หน่วยความจำน้อย เพราะเก็บ Path เส้นเดียวจากโหนดรากไปถึงใบ (คือโหนดที่ไม่มี Successor) รวมกับโหนดที่ในระดับเดียวกันที่ยังไม่ได้ Expand

หลังจากที่โหนดถูก Expand แล้ว ถ้าสำรวจดูโหนดลูกทั้งหลายไปจนถึงใบว่าไม่ใช่เป้าหมาย ก็สามารถนำโหนดนั้นออกจากหน่วยความจำไปได้เลย

ถ้า Search tree มี State space ที่มี Branching factor เป็น b และความลึกสุดคือ m หน่วยความจำที่ต้องใช้จะเป็น bm + 1 โหนด (คิดในเทอมของโหนด ขึ้นอยู่กับว่าโหนดนั้นใช้เนื้อที่เท่าใด)

ข้อเสียของวิธีนี้คือ ถ้าเลือกเส้นทางผิด อาจหลงทางและกลับขึ้นมาไม่ได้ เช่น Search tree ที่ลึกลงไปไม่รู้จบ ไม่มีทางกลับ หรืออาจจะติดอยู่ใน Infinite loop จึงไม่สามารถหาเป้าหมายพบ วิธีนี้จึงไม่มี Completeness นอกจากนี้ถ้าพบคำตอบก็อาจจะไม่ใช่คำตอบที่ดีที่สุด เพราะถ้าปัญหามีเป้าหมายมากกว่าหนึ่งขึ้นไป โหนดเป้าหมายที่อยู่ในชั้นที่ตื้นกว่าอาจจะยังไม่ถูก Expand เพราะมัวไป Expand อีกกิ่งหนึ่งที่มีโหนดเป้าหมายอีกโหนดหนึ่งอยู่ในระดับที่ลึกกว่า Depth-first search จึงไม่ Optimal ถ้าหยุดการค้นหาเมื่อพบเป้าหมายแรก

3.4.4 Depth-limited search

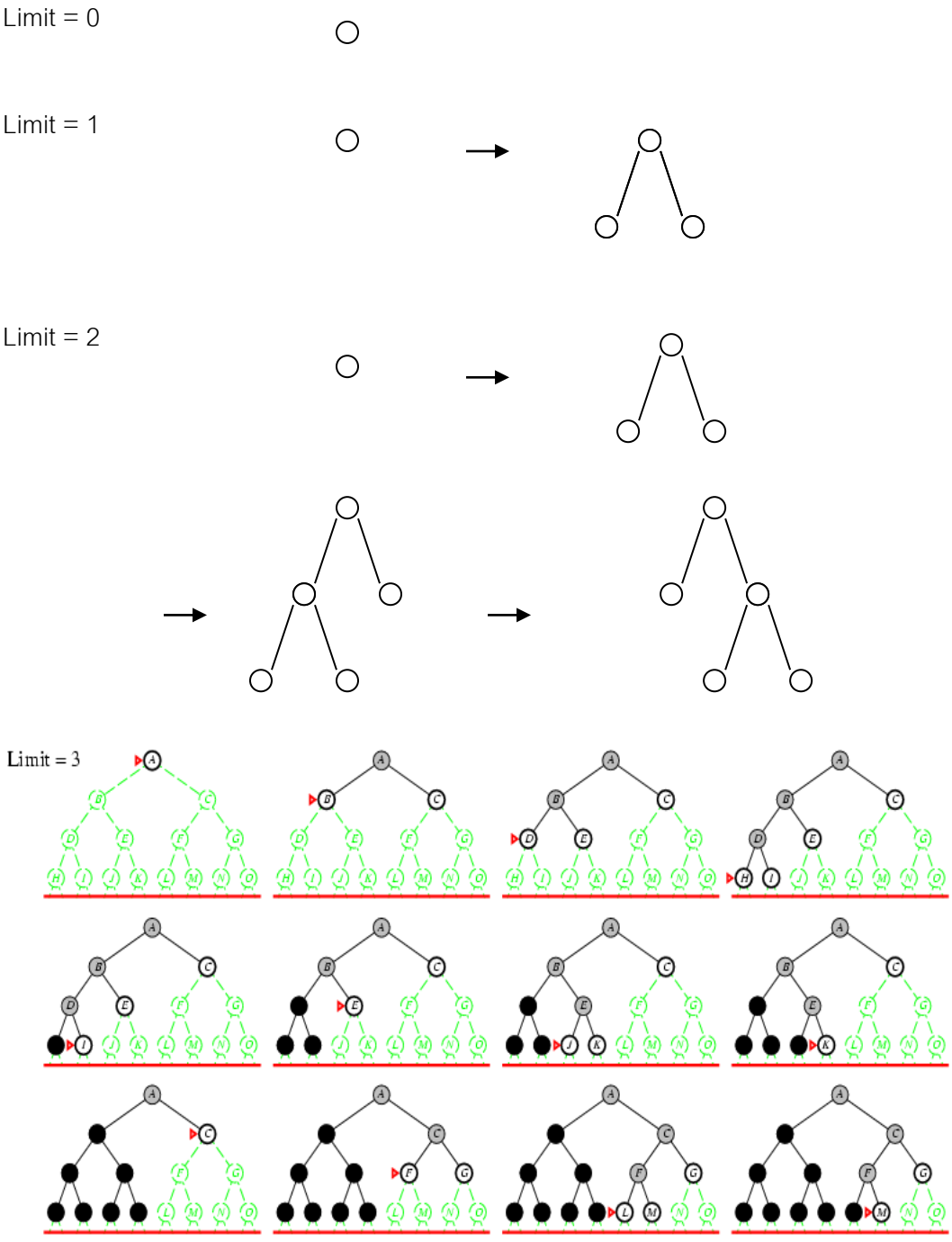
วิธีนี้ต้องการแก้ปัญหาของ Depth-first searh โดยหลีกเลี่ยงการลงไปลึกใน Tree ซึ่งอาจจะทำให้กลับขึ้นมาไม่ได้ วิธี Depth-limited search จึงกำหนดความลึกของ Tree ไว้ด้วยค่าจำกัดค่าหนึ่ง และไม่ยอมให้ Expand ลงไปลึกกว่านั้น

สมมุติว่ากำหนดให้ค่าจำกัดความลึกนี้เป็น l โหนดใด ๆ ที่ระดับความลึก l จะถูกสมมุติว่าเป็นโหนดระดับสุดท้ายที่ไม่มี Successor ต่อไป เมื่อเป็นเช่นนี้จึงสามารถตัดปัญหาเรื่องการลงไปใน Tree ที่ลึกไม่มีที่สิ้นสุด (infinite path) แต่วิธีนี้ก็อาจจะหาคำตอบไม่ได้ถ้าหากเลือกค่า l ผิด ไปเลือกค่า l ที่น้อยกว่า d (l < d เมื่อ d เป็นระดับความลึกขอเป้าหมาย) นั่นคือเป้าหมายที่ตื้นที่สุดก็ยังอยู่ลึกกว่าค่าจำกัดความลึกที่กำหนดไว้ ทำให้หาไม่พบ แต่ถ้าค่า l มากกว่า d แล้ว เป้าหมายที่หาได้ก็อาจจะไม่ Optimal ก็ได้ (เหตุผลเดียวกันกับใน Depth-first search)

การหาค่าจำกัดความลึกขึ้นอยู่กับตัวปัญหา ถ้ามีความรู้เกี่ยวกับตัวปัญหามากขึ้นเราจะสามารถหาค่านี้ได้ดี เช่นปัญหาการเดินทางในโรมาเนีย เมื่อรู้ว่าแผนที่ของโรมาเนียมีเมืองอยู่ 20 เมือง เราอาจจะกำหนดว่าให้ค่าจำกัดความลึกเป็น l=19 ซึ่งเป็นข้อที่เป็นไปได้ดี แต่ถ้าศึกษาแผนที่โรมาเนียให้ดีแล้ว จะพบว่าเมืองหนึ่ง ๆ สามารถไปถึงได้โดยเส้นทางหลายเส้นทางจากหลายเมืองต่าง ๆ กัน อย่างมากแล้วใช้ระยะทางไม่เกิน 9 ขั้นตอน จำนวน 9 นี้เรียกได้ว่าเป็นเส้นผ่านศูนย์กลางของ State space และนำมาใช้เป็นค่าจำกัดความลึกที่มีประสิทธิภาพมากขึ้นได้ แต่กับปัญหาส่วนใหญ่แล้ว เราไม่สามารถรู้ค่าจำกัดที่ดีได้เลยจนกว่าจะแก้ปัญหาสำเร็จ

3.4.5 Iterative deepening depth-first search

อาจจะเรียกสั้น ๆ ว่า Iterative deepening search เป็นวิธีแบบทั่วไปมากที่สุด มีกระบวนการคล้ายกับแบบ Depth-first search มีวิธีการค้นหาตามรูปที่ 3.11 ดังนี้



รูปที่ 3.11 ไดอะแกรมแสดงการค้นหาแบบ Iterative deepening depth-first search

วิธีการนี้ต้องตั้งค่าจำกัดความลึกขึ้นเป็นค่าแรกก่อน แล้วเพิ่มค่าจำกัดนี้ขึ้นทีละน้อย ตั้งแต่เริ่มต้นเป็น 0 (หาเฉพาะราก) แล้วจึงให้ค่าจำกัดความลึกเป็น 1 ต่อไปเป็น 2 แล้วเพิ่มเป็น 3 ตามลำดับจนกระทั่งพบโหนดเป้าหมาย กระบวนการหาซ้ำจะเริ่มไปจนค่าจำกัดความลึกเป็น d (ความลึกของโหนดเป้าหมาย) วิธีนี้รวมข้อดีของ Breadth-first search และ Depth-first search เข้าด้วยกัน จึงมีทั้ง Completeness และ Optimality นอกจากนั้นยังใช้หน่วยความจำน้อยอีกด้วย (เท่ากับฟังก์ชัน O(bd)) การค้นหาแบบนี้ดูได้จากรูป 3.11 ซึ่งใช้ตัวอย่าง Binary tree ในการค้นหา

วิธีนี้อาจจะดูเหมือนว่าสิ้นเปลืองเวลามาก เพราะมีหลายสถานะที่ถูกสร้าง (Generate) ขึ้นมาซ้ำแล้วซ้ำเล่า แต่ที่จริงแล้วไม่ได้สิ้นเปลืองไปสักเท่าใด เนื่องจากใน Tree ที่มี Branching factor ใกล้เคียงกัน โหนดส่วนใหญ่จะอยู่ในชั้นล่าง ๆ โหนดชั้นบนที่สร้างขึ้นหลายครั้งไม่ถือว่ามาก เช่นโหนดที่อยู่ในระดับความลึก d (ระดับความลึกเดียวกับเป้าหมาย) จะถูกสร้างขึ้นเพียงครั้งเดียวเท่านั้น แต่โหนดที่อยู่ชั้นบนถัดขึ้นไป สร้างขึ้น 2 ครั้ง ไปตามลำดับ จำนวนของโหนดที่ถูกสร้างขึ้นนี้เท่ากับ

$$N(IDS) = (d)b + (d-1)b^2 + \ldots + (1)b^d$$

ทำให้เกิดเป็นฟังก์ชัน Time complexity $O(b^d)$ เมื่อเปรียบเทียบกับจำนวนโหนดที่เกิดจาก Breadth-first search ซึ่งเป็น

$$N(BFS) = b + b^2 + \ldots + b^d + (b^{d+1} - b)$$

จะเห็นว่า Breadth-first search สร้างโหนดที่ระดับความลึก d+1 แต่ Iterative deepening search สร้างแค่ในระดับความลึก d เท่านั้น ผลคือทำให้ Iterative deepening search ทำงานได้เร็วกว่า Breadth-first search ตัวอย่างเช่น ถ้าให้ b = 10 , d = 5 จำนวนโหนดของ 2 วิธีนี้คือ

$$N(IDS) = 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}450$$

$$N(BFS) = 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 + 999{,}990 = 1{,}111{,}100$$

โดยทั่วไปแล้ว วิธีการค้นหาแบบ Iterative deepening เป็นวิธีที่เหมาะสมสำหรับการค้นหาแบบไม่ใช้ข้อมูลที่มี State space ขนาดใหญ่ และไม่ทราบระดับความลึกของคำตอบ

3.5 การเลี่ยงสถานะซ้ำ

ในกระบวนการค้นหา สิ่งที่ซับซ้อนและทำให้เสียเวลาเป็นอย่างมากสิ่งหนึ่งคือการ Expand โหนด (หรือสถานะ) ที่เคยพบมาแล้ว และเคย Expand มาแล้วในเส้นทางอื่น ปัญหาบางปัญหาอาจจะไม่พบอุปสรรคข้อนี้ เช่นกรณีที่ State space มีลักษณะเป็น Tree จริง ๆ แต่ละโหนดมีเส้นทางไปถึงเพียงเส้นเดียวเท่านั้น แต่บางปัญหาก็ไม่สามารถหลีกเลี่ยงสถานะซ้ำได้ เช่นปัญหาการเดินทาง (Route-finding) เพราะ Search tree ของปัญหาเหล่านี้เป็นแบบไม่จำกัด (Infinite) อย่างไรก็ตาม มีวิธีหลีกเลี่ยงสถานะซ้ำได้ดังนี้

3.5.1 ไม่กลับไปยังสถานะที่เพิ่งจากมา ที่โหนดหนึ่ง ๆ จะไม่ Expand หรือสร้าง Successor ที่มีสถานะซ้ำกับโหนดแม่ของโหนดนั้น

3.5.2 ไม่สร้างเส้นทางที่เดินเป็นวงรอบ (Cycle) หรือไม่ Expand Successor ที่เหมือนกับบรรพบุรุษ (Ancestor) ของโหนดนั้น

3.5.3 ไม่สร้างสถานะที่เคยสร้างมาแล้ว กรณีนี้จะต้องเก็บทุกสถานะไว้ในหน่วยความจำเพื่อตรวจสอบว่าซ้ำหรือไม่

แบบฝึกหัดบทที่ 3

1. จงอธิบายความหมายของคำต่อไปนี้ตามที่ท่านเข้าใจ

   สถานะ (State) State space

   Search tree Search node,

   Goal Action

   Successor function

2. เพราะเหตุใดการสร้างเป้าหมาย (Goal formulation) ต้องมาก่อนการสร้างข้อปัญหา (Problem formulation)

3. จงกำหนดสถานะเริ่มต้น (Initial state) การทดสอบเป้าหมาย (Goal test) Successor function และ Path cost function ให้กับปัญหาในข้อต่อไปนี้

   3.1 ปัญหาการลงสีในแผนที่โดยใช้สีเพียง 4 สี สมมุติว่าอาณาจักรแห่งหนึ่งประกอบด้วยแคว้นเล็กแคว้นน้อยจำนวนหนึ่ง ต้องการระบายสีในแผนที่ให้แคว้นที่อยู่ติดกันมีสีต่างกัน และสมมติว่าไม่มีแคว้นใดมีอาณาเขตอยู่ติดกันเกินกว่า 4 แคว้น

   3.2 ปัญหาลิงกับกล้วย มีลิงสูง 3 ฟุต อยู่ในห้องที่มีกล้วยแขวนติดเพดานสูง 8 ฟุต ลิงต้องการกินกล้วยหวีนี้ ในห้องมีกล่องสูง 3 ฟุตที่ซ้อนกันได้ เคลื่อนที่ได้ และปีนได้ จำนวน 2 กล่อง

   3.3 ปัญหาเหยือกใส่น้ำ มีเหยือก 3 ใบ จุน้ำได้ 12, 8, และ 3 แกลลอนตามลำดับ และมีก๊อกน้ำสำหรับเติมน้ำได้ไม่จำกัด กิจกรรมที่ทำได้คือเติมน้ำลงเหยือก หรือเทน้ำออกจากเหยือก โดยการเติมแต่ละครั้งเป็นการเติมเต็มเหยือก การเทน้ำจะเทหมดเหยือก สามารถเทน้ำลงในเหยือกใบอื่น หรือเททิ้งลงพื้นก็ได้ ปัญหาคือต้องการเติมน้ำให้ได้ปริมาณเพียง 1 แกลลอนเท่านั้น

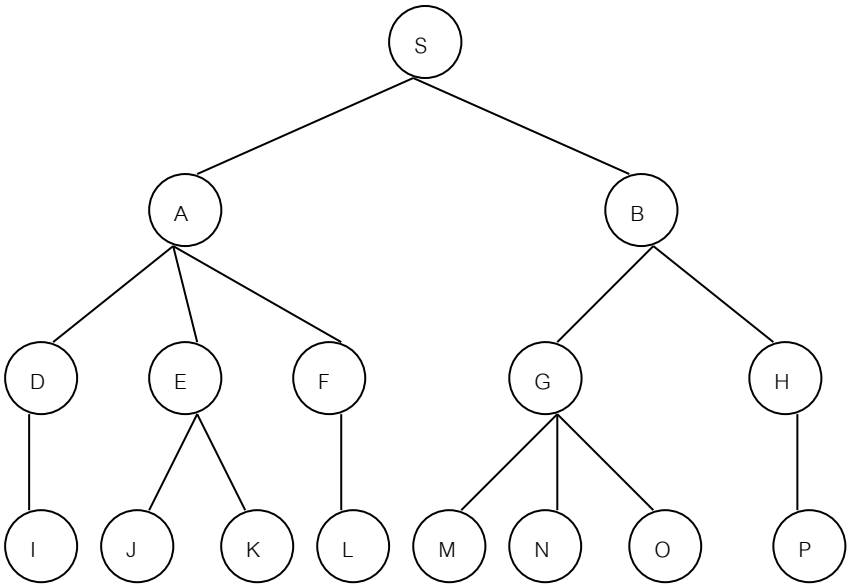
4. ปัญหามิชชันนารีกับคนป่า เป็นหนึ่งในปัญหาคลาสสิกและมีชื่อเสียงของ AI มีมิชชันนารี 3 คนกับคนป่า 3 คนอยู่ฝั่งเดียวกันของแม่น้ำ ทุกคนต้องการข้ามไปที่อีกฝั่งหนึ่ง แต่มีเรืออยู่ลำเดียวซึ่งบรรจุคนได้ครั้งละไม่เกิน 2 คนเท่านั้น ต้องการหาวิธีพาทุกคนข้ามแม่น้ำ แต่ต้องระวังไม่ปล่อยให้มิชชันนารีบนฝั่งใด ๆ มีจำนวนน้อยกว่าคนป่าบนฝั่งเดียวกัน เพราะเมื่อคนป่ามีจำนวนมากกว่ามิชชันนารีก็จะจับมิชชันนารีมากิน จงสร้างข้อปัญหาสำหรับเรื่องนี้ และเขียน State space สำหรับปัญหา

5. จากแผนภาพในรูปที่ 3.12 กำหนดให้สถานะเริ่มต้นที่ S สถานะเป้าหมายคือ M จงแสดงขั้นตอนการค้นหาตามลำดับ โดยใช้แผนภาพต้นไม้ทีละขั้นโดยละเอียด ใช้วิธีการค้นหาแบบ

   5.1 Breadth-first search

   5.2 Depth-first search

   5.3 Iterative deepening search



รูปที่ 3.12 แผนภาพแสดง Search tree สำหรับโจทย์ข้อ 5