บทที่ 4
การค้นหาแบบใช้ข้อมูล
(Informed Search)

การใช้วิธีค้นหาในบทที่ 3 เป็นวิธีทั่วไปที่ไม่ได้นำข้อมูลหรือความรู้เกี่ยวกับปัญหามาช่วยในกระบวนการทำงาน แต่ละวิธีจึงอาจจะทำงานได้อย่างไม่มีประสิทธิภาพเท่าใดนัก แต่ถ้านำความรู้ที่เฉพาะเจาะจง และเกี่ยวข้องกับตัวปัญหามาช่วยแล้ว จะสามารถค้นหาได้รวดเร็วและมีประสิทธิภาพมากขึ้น

4.1 วิธีการค้นหาแบบใช้ข้อมูล (Informed search strategies)

วิธีการโดยทั่วไปของการค้นหาแบบนี้เรียกว่า “การค้นหาแบบดีที่สุดมาก่อน” (Best-first search) อัลกอริทึมที่ใช้อาจจะเป็นอัลกอริทึมสำหรับการค้นหาใน Tree หรือใน Search graph ก็ได้ โดยที่โหนดที่เลือกมา Expand เลือกจากการหาค่าฟังก์ชันประเมินค่า (Evaluation function : f(n)) ปกติแล้วโหนดที่มีการประเมินค่าต่ำที่สุดจะนำมา Expand ก่อน เพราะการประเมินค่าจะเป็นการวัดระยะห่างจากโหนดนั้นไปสู่โหนดเป้าหมาย (Goal)

การใช้คำว่า Best-first search เป็นคำอุปมาเท่านั้น ยังไม่ตรงกับความจริงเท่าใด เพราะถ้าสามารถ Expand โหนดที่ดีที่สุดจริง ๆ ได้ วิธีนี้จะไม่ใช่การค้นหา แต่จะเป็นการพุ่งตรงเข้าสู่เป้าหมายเลย แต่สำหรับการเลือกโหนดจะพยายามเลือกให้ดีที่สุด โดยพิจารณาจากฟังก์ชันประเมินค่า ถ้าฟังก์ชันนี้เที่ยงตรงจริง โหนดที่เลือกมาจะเป็นโหนดที่ดีที่สุดจริง ๆ แต่บางครั้งก็ไม่เป็นไปตามที่คาดหวังเสมอไป

Best-first search มีอัลกอริทึมต่าง ๆ กัน ขึ้นอยู่กับฟังก์ชันประเมินค่าที่ต่างกันโดยมีองค์ประกอบสำคัญของอัลกอริทึมคือฟังก์ชันฮิวริสติก (Heuristic function) เขียนว่า h(n)

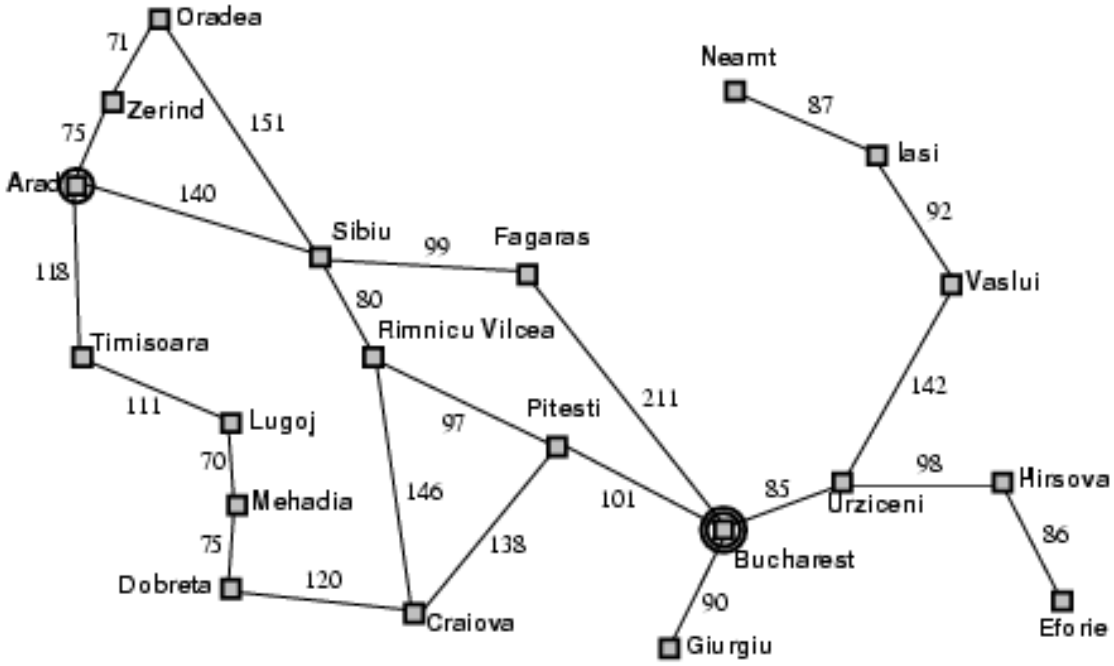
h(n) = ค่าประมาณของเส้นทางที่มีค่าใช้จ่ายต่ำที่สุดจากโหนด n ถึงโหนดเป้าหมาย

ตัวอย่างเช่น การเดินทางในโรมาเนีย อาจจะประมาณค่าใช้จ่ายต่ำสุดจากเมืองอารัด ไปยังเมืองบูคาเรสต์ โดยใช้ระยะห่างเป็นเส้นตรงจากเมืองอารัดไปยังเมืองบูคาเรสต์ ดังนั้น ถ้าโหนด n เป็นเป้าหมายแล้ว จะได้ h(n) = 0

4.1.1 Greedy best-first search

เป็นวิธีที่พยายาม Expand โหนดที่อยู่ใกล้ชิดกับเป้าหมายมากที่สุด โดยมีหลักการว่าโหนดที่อยู่ใกล้เป้าหมายที่สุดย่อมพาไปสู่คำตอบได้เร็วที่สุด ดังนั้น ฟังก์ชันประเมินค่าจะพิจารณาแต่ฟังก์ชันฮิวริสติกเพียงอย่างเดียว นั่นคือ

$$f(n) = h(n)$$



รูปที่ 4.1 แผนที่บอกเส้นทางอย่างง่ายในโรมาเนีย (บอกระยะทางมีหน่วยเป็น กม.)

จากตัวอย่างปัญหาการเดินทางในโรมาเนีย ดูจากแผนที่ในรูปที่ 4.1 ฮิวริสติกที่ใช้คือระยะห่างที่วัดเป็นเส้นตรง เขียนแทนด้วย h_{SLD} (SLD=Straight line distance) ถ้าเป้าหมายคือเมืองบูคาเรสต์ เราจำเป็นต้องรู้ระยะห่างที่วัดเป็นตรงจากเมืองใด ๆ ไปยังเมืองบูคาเรสต์ ข้อมูลนี้สามารถคำนวณหาได้จากแผนที่ที่บอกสเกลการวัดระยะทาง สมมุติว่าค่า h_{SLD} หาได้ดังตารางที่ 4.1

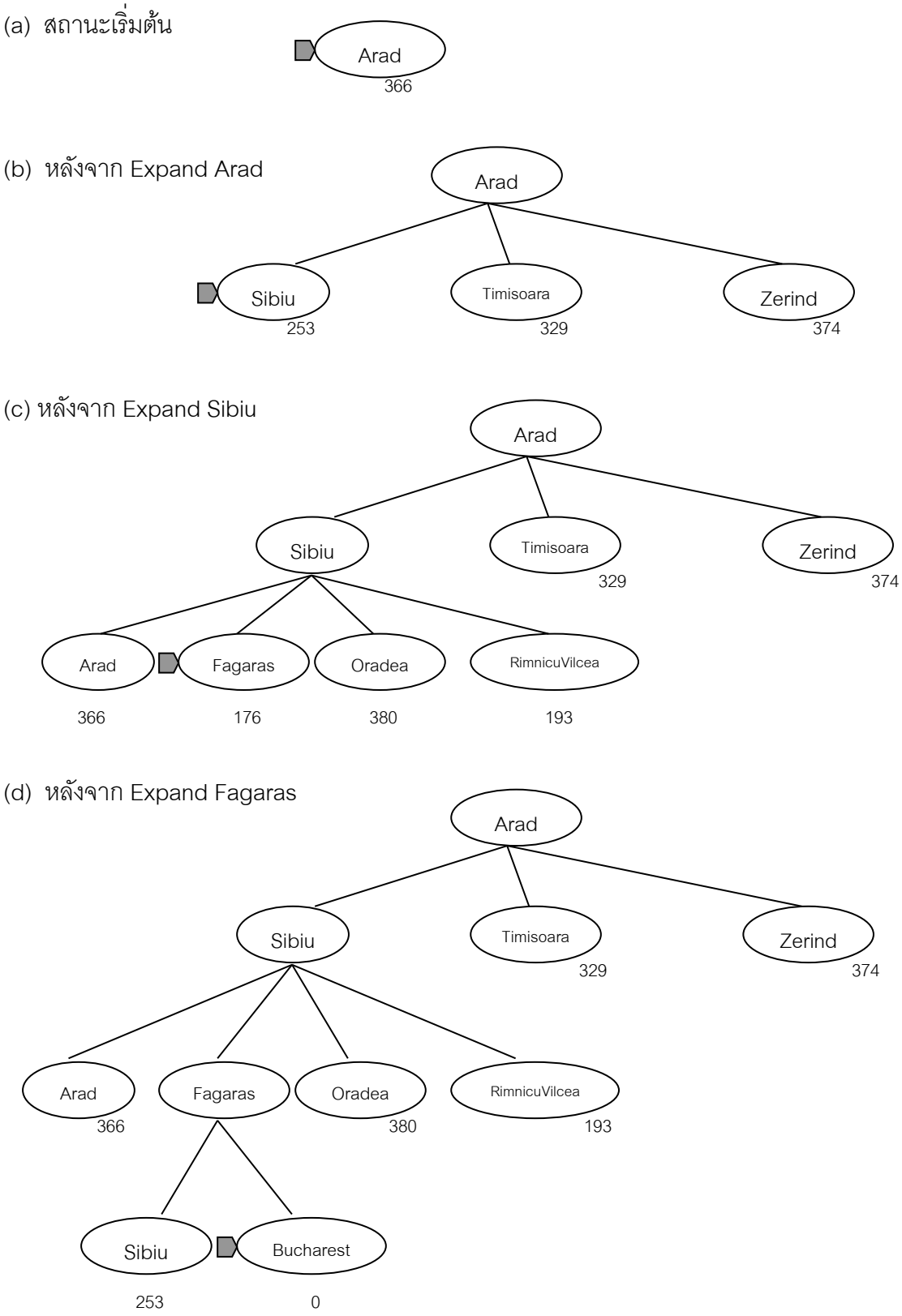
รูปที่ 4.2 แสดงกระบวนการค้นหาแบบ Greedy best-first search ที่ใช้ h_{SLD} หาเส้นทางจากโหนด Arad ไปยังโหนด Bucharest โหนดแรกที่ Expand ต่อจากโหนด Arad คือ Sibiu เพราะเป็นโหนดที่อยู่ใกล้กับโหนด Bucharest มากกว่าโหนด Zerind หรือโหนด Timisoara โหนดต่อไปที่ถูก Expand คือ Fagaras เพราะเป็นโหนดที่ใกล้ที่สุด จากโหนด Fagaras สร้าง

โหนด Bucharest ซึ่งเป็นเป้าหมาย ในปัญหาเฉพาะเจาะจงเช่นปัญหานี้ การค้นหาแบบ greedy best-first ใช้ h_{SLD} ค้นหาคำตอบโดยไม่มีการ Expand โหนดอื่นใดที่อยู่นอกเส้นทาง ดังนั้น ค่าใช้จ่ายในการค้นหา (Search cost) จึงต่ำที่สุด แต่ไม่ใช่ดีที่สุด เพราะถ้าดูเส้นทางในแผนที่จริง จะพบว่า ระยะทางจาก Sibiu ไป Fagaras แล้วไป Bucharest จะมากกว่าระยะทางจาก Sibiu ไป Rimnicu Vilcea แล้วไป Bucharest อยู่ 32 กม.

ตารางที่ 4.1 ค่า h_{SLD} หาจากระยะห่างวัดเป็นเส้นตรง (Straight-line distance) จากแต่ละเมืองไปยังเมือง Bucharest

เมือง	h_{SLD}	**เมือง**	h_{SLD}
Arad	366	Mehadia	241
Bucharest	0	neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

ค่า h(n) ที่พยายามทำให้ต่ำที่สุด บางครั้งก็นำปัญหามาให้ สมมุติว่าปัญหาคือต้องการเดินทางเมือง Iasi ไปยัง Fagaras ฮิวริสติกจะให้ข้อแนะนำว่า ต้อง Expand ไปยังโหนด Neamt ก่อน เพราะใกล้กับ Fagaras มากที่สุด แต่ตรงจุดนี้เป็นทางตัน คำตอบที่ถูกต้องของปัญหานี้คือไปที่ Vaslui ก่อน แล้วจึงไป Urziceni, Bucharest และ Fagaras ตามลำดับ ในกรณีนี้ฮิวริสติกทำให้ต้อง Expand โหนดที่ไม่จำเป็น (Neamt) และนอกจากนี้ ถ้าไม่ระวังคอยตรวจหาสถานะที่เคยทำซ้ำ ก็อาจจะทำให้หลงวนอยู่ไม่รู้จบได้ (โดยค้นหาตามเส้นทางกลับไปกลับมาระหว่าง Neamt กับ Iasi)



รูปที่ 4.2 ตัวอย่างการค้นหาแบบ Greedy best-first search

Greedy best-first มีลักษณะคล้ายกับ Depth-first search ตรงที่ค้นหาเป็นเส้นทางเดียวไปจนถึงเป้าหมาย และย้อนกลับเมื่อเจอทางตัน จึงมีข้อเสียเช่นเดียวกับ Depth-first search นั่นคือไม่ Optimal และไม่ Complete ในกรณีที่แย่ที่สุดจะมี Time และ Space complexity เป็นฟังก์ชัน $O(b^m)$ เมื่อ m เป็นระดับความลึกที่ลึกที่สุดของ Search space

4.1.2 A* search

เป็นวิธีค้นหาแบบ Best-first search ที่รู้จักกันอย่างกว้างขวาง วิธีนี้ประเมินค่าโหนดโดยคิดค่าใช้จ่ายตั้งแต่โหนดเริ่มต้นจนถึงโหนด n (ฟังก์ชันนี้เรียกว่า g(n)) รวมกับค่าใช้จ่ายจากโหนด n ไปถึงเป้าหมาย (ฟังก์ชันนี้เรียกว่า h(n)) ค่าของฟังก์ชันการประเมินค่า f(n) จึงเขียนได้ดังนี้

$$f(n) = g(n) + h(n)$$

เมื่อ g(n) เป็นค่าใช้จ่ายตั้งแต่โหนดเริ่มต้น (ราก) จนถึงโหนด n

และ h(n) เป็นค่าใช้จ่ายโดยประมาณ (Estimated cost) ที่ต่ำที่สุดของเส้นทางจากโหนด n จนถึงเป้าหมาย

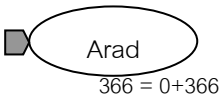
เมื่อพิจารณาจากค่า g และ h แล้ว f(n) จึงเป็นค่าใช้จ่ายโดยประมาณของคำตอบที่ราคาถูกที่สุดที่ผ่านโหนด n ปกติแล้วถ้าเราต้องการหาคำตอบที่มีค่าใช้จ่ายถูกที่สุด จะต้องพยายามหาโหนดที่มีค่า g(n) + h(n) ต่ำที่สุด ดังนั้นวิธีที่ใช้ค่า f นี้จึงสมเหตุสมผล และได้ผลดี แต่ทั้งนี้ขึ้นอยู่กับว่าฟังก์ชันฮิวริสติก h(n) เป็นฟังก์ชันที่เหมาะสมเพียงใด ถ้าฟังก์ชันนี้ดีพอ A* search จะมีลักษณะ Complete และ Optimal

ฟังก์ชันฮิวริสติกที่ดีเป็นฮิวริสติกที่ยอมรับได้ (Admissible heuristic)หมายความว่าเมื่อ h(n) ประมาณค่าใช้จ่ายที่ไปสู่เป้าหมายได้พอดีไม่เกินเหตุ (ไม่เป็น Overestimate cost) ฮิวริสติกที่ยอมรับได้นี้มีลักษณะที่ดีที่สุดอยู่แล้ว (Optimistic) เนื่องจากฟังก์ชันนี้มองค่าใช้จ่ายของการแก้ปัญหาต่ำกว่าที่เป็นจริง ส่วนค่า g(n) นั้นเป็นค่าใช้จ่ายตามจริงไปจนถึงโหนด n อยู่แล้ว ดังนั้นเมื่อรวมกันกับ h(n) จึงเห็นได้ว่า f(n) จะไม่ Overestimate ค่าใช้จ่ายตามจริงของคำตอบ

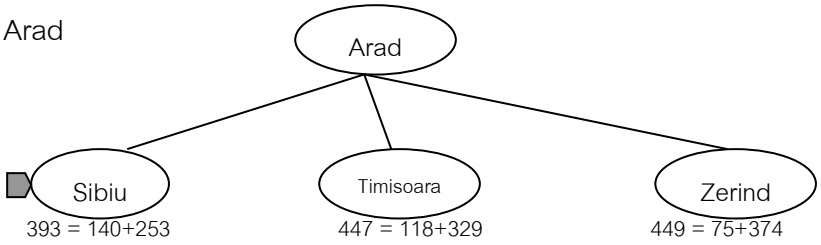
ตัวอย่างที่เห็นได้ชัดเจนคือ h_{SLD} ที่วัดจากเมืองใด ๆ ไปยัง Bucharest ระยะห่างที่วัดเป็นเส้นตรงนี้ยอมรับได้ เพราะระยะทางสั้นที่สุดระหว่างจุด 2 จุด คือเส้นตรงที่เชื่อมจุดทั้งสองนั้น เส้นตรงไม่มีทาง Overestimate รูปที่ 4.3 แสดงการค้นหาแบบ A* ไปสู่เมือง Bucharest ค่า g หาได้จากการคิดระยะทางตามจริงที่ปรากฏในแผนที่นับจากจุดเริ่มต้น (Arad) ส่วนค่า h คือ h_{SLD} ตามที่แสดงไว้ในตารางที่ 4.1 จะสังเกตได้ว่าเมือง Bucharest อยู่ในส่วนของ Tree ที่ขั้นตอน (e) แต่ไม่ถูกเลือกมา Expand เพราะค่า f-cost ของ Bucharest เป็น 450 ซึ่งสูงกว่าของ Pitesti (417)

เมื่อคำตอบที่ผ่าน Pitesti มีค่าใช้จ่ายต่ำกว่าคือ 417 อัลกอริทึมจึงไม่เลือกคำตอบที่มีค่าใช้จ่ายสูงกว่านั้น (ไม่เลือก 450)

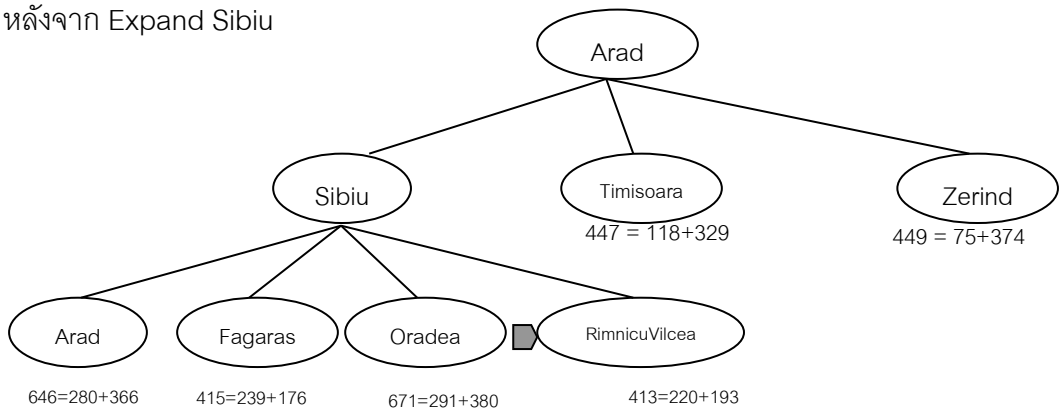
(a) สถานะเริ่มต้น



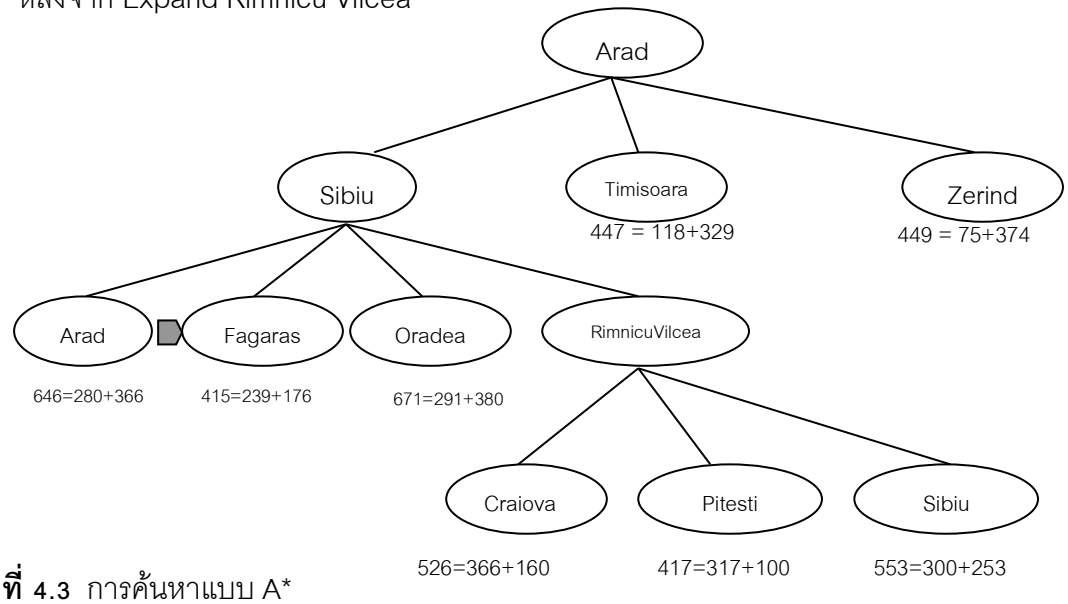
(b) หลังจาก Expand Arad



(c) หลังจาก Expand Sibiu

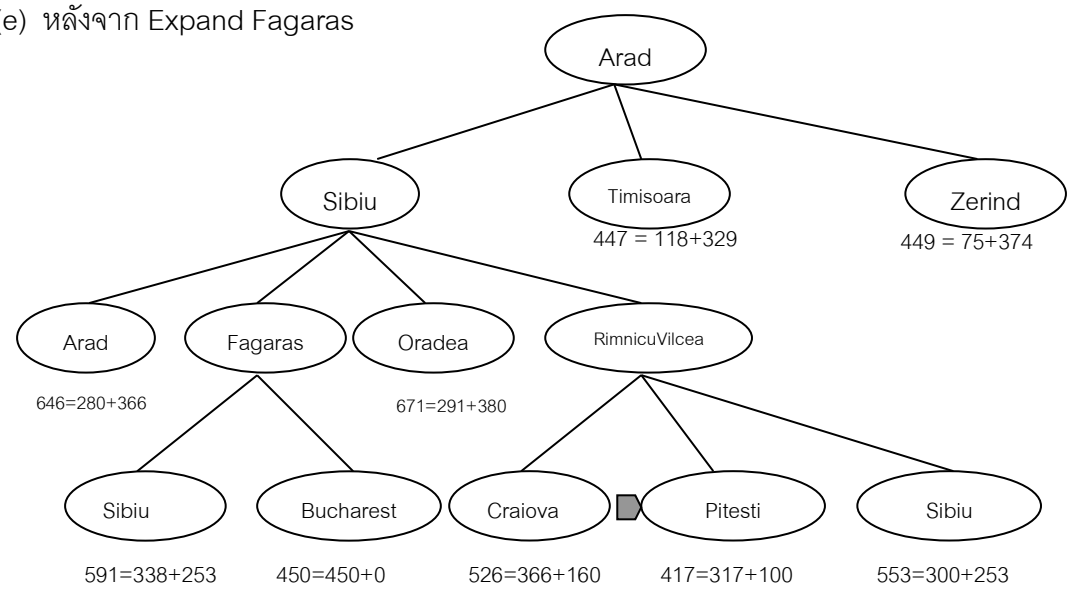


(d) หลังจาก Expand Rimnicu Vilcea

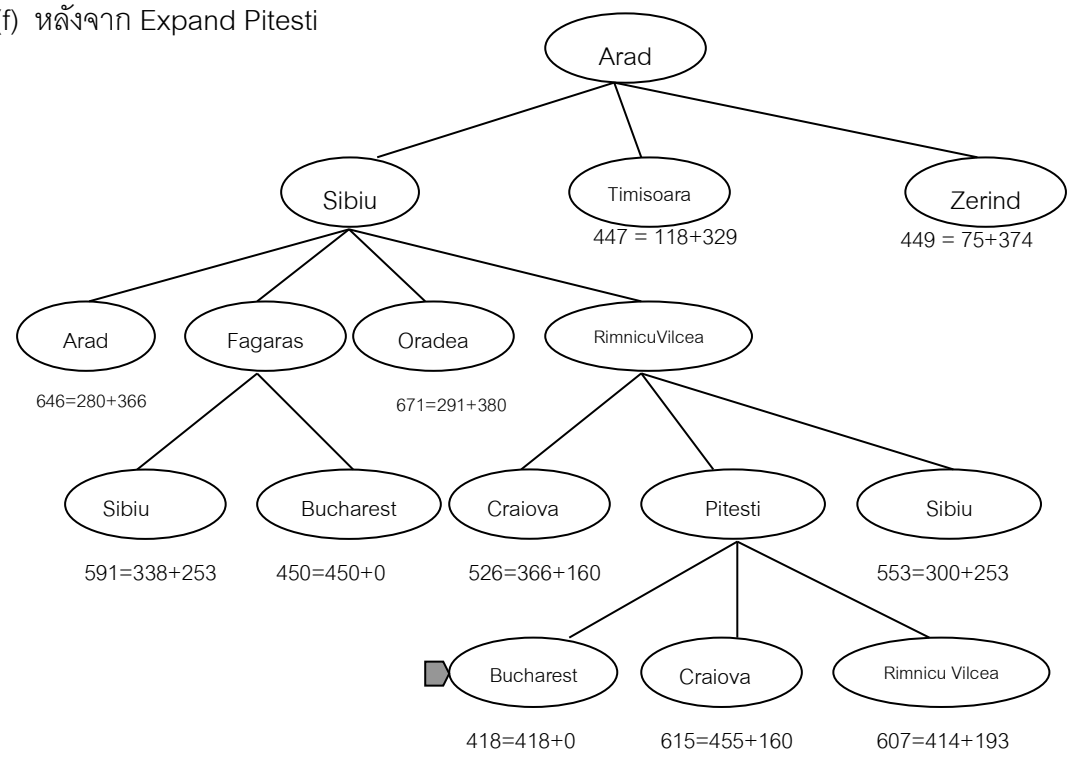


รูปที่ 4.3 การค้นหาแบบ A*

(e) หลังจาก Expand Fagaras



(f) หลังจาก Expand Pitesti



รูปที่ 4.3 (ต่อ) การค้นหาแบบ A*

การพิสูจน์ว่า ถ้า h(n) ยอมรับได้ แล้ว A* search จะ Optimal ทำได้โดยใช้ Tree search และสมมุติให้มีโหนดเป้าหมายที่ Optimal อีกโหนดหนึ่งในกิ่งของ Tree ที่ยังไม่ได้ Expand เพราะอยู่ลึกลงไปอีกใน Tree สมมุติให้โหนดนี้ชื่อ G2

และสมมุติว่าค่าใช้จ่ายของคำตอบที่ดีที่สุดของเป้าหมายที่หาได้ก่อนหน้านี้เป็น C*

เนื่องจาก G2 เป็นโหนดเป้าหมายด้วยเช่นกัน h(G2) จึงมีค่าเป็น 0 (ค่า h ของโหนดเป้าหมายเท่ากับ 0 เสมอ) และเพราะ G2 เป็นโหนดที่อยู่ไกลกว่าลึกลงไปใน Tree ยังไม่ถูก Expand ดังนั้นจึงได้ว่าค่า f ของ G2 ต้องมากกว่า C*

$$f(G2) = g(G2) + h(G2)$$
$$= g(G2) + 0$$
$$= g(G2) > C^*$$

พิจารณาโหนด n ที่อยู่ในเส้นทางของคำตอบที่ดีที่สุด (ถ้ามีคำตอบ ก็ต้องมีโหนดเช่นนี้อยู่แน่นอน) ถ้า h(n) ไม่ Overestimate ค่าใช้จ่ายของ Solution path แล้ว จะได้ว่า

$$f(n) = g(n) + h(n) \leq C^*$$

ต่อไปต้องแสดงให้เห็นว่า $f(n) \leq C^* \leq f(G2)$ เพื่อว่าจะไม่ต้อง Expand G2 และทำให้ A* สามารถหาคำตอบที่ดีที่สุดได้

ฟังก์ชันฮิวริสติกที่ยอมรับได้จะมีคุณสมบัติอีกข้อหนึ่งคือ Consistency นั่นคือ สำหรับโหนด n ใด ๆ และโหนดลูก (Successor) n' ของโหนด n ที่เกิดจากแอคชัน a ค่าใช้จ่ายที่ประมาณได้จาก n ไปถึงเป้าหมายจะไม่มากกว่าค่าใช้จ่ายจากโหนด n ไปยัง n' บวกกับค่าใช้จ่ายจาก n' ถึงเป้าหมาย หรือเขียนได้ว่า

$$h(n) \leq c(n, a, n') + h(n') \qquad (4.1)$$

ตัวอย่างเช่นฮิวริสติกที่เป็น h_{SLD} มีลักษณะที่ยอมรับได้ และ Consistency เนื่องจากเป็นระยะห่างที่วัดเป็นเส้นตรง ซึ่งเป็นเส้นทางที่สั้นที่สุดตามทฤษฎีเรขาคณิต จึงไม่มีทางมากไปกว่าค่าใช้จ่ายตามจริงที่เกิดขึ้นจากการเดินทางจากโหนด n ไปยังโหนดลูก n' ได้

สิ่งที่ตามมาหลังจากได้ h(n) ที่ Consistency คือ ค่าของ f(n) ตาม Path ใด ๆ จะไม่มีทางลดลง พิสูจน์ได้โดยสมมุติว่า n' เป็นโหนดลูกของ n แล้ว จะได้

$g(n') = g(n) + c(n, a, n')$ เมื่อมีการกระทำ a ใด ๆ

จากนิยามของฟังก์ชันการประเมินค่า f จะได้

$$f(n') = g(n') + h(n')$$
$$= g(n) + c(n, a, n') + h(n')$$

จาก (4.1) จะได้

$$g(n) + c(n, a, n') + h(n') \geq g(n) + h(n)$$
$$\geq f(n)$$

ดังนั้น จึงได้ว่า $f(n') \geq f(n)$

เมื่อดูลำดับการ Expand โหนดต่าง ๆ ของ A* แล้ว ลำดับของ f(n) ของโหนด n ใด ๆ ที่เกิดขึ้นจะเป็นลำดับที่ไม่มีค่าลดลง (Nondecreasing) ดังนั้น โหนดเป้าหมายโหนดแรกที่ถูกพบขณะเลือกมา Expand จะเป็นคำตอบที่ดีที่สุด เนื่องจากโหนดอื่นที่เหลือที่ยังไม่ได้รับเลือกมา Expand จะต้องมีค่าใช้จ่ายสูงกว่า

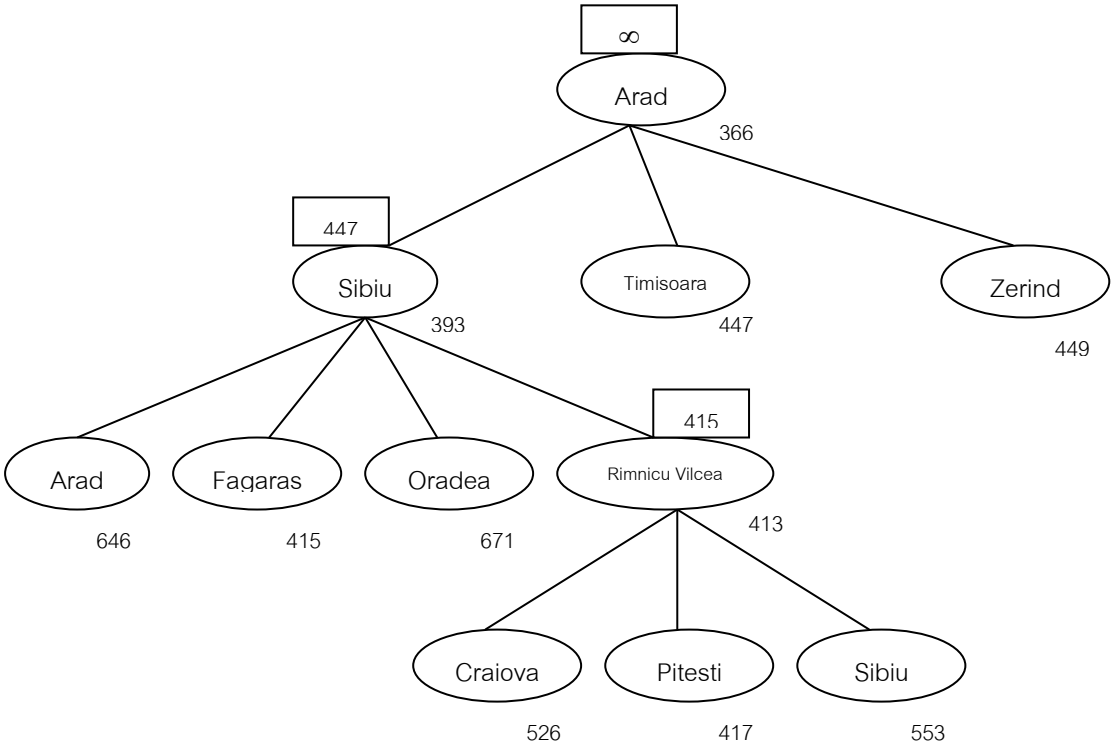
แม้ว่า A* จะมีประสิทธิภาพ มีลักษณะ Complete และ Optimal แต่ก็ยังไม่ใช่อัลกอริทึมที่นำไปใช้กับการค้นหาทุกปัญหาได้ เนื่องจากในปัญหาส่วนใหญ่ จำนวนโหนดที่ต้องหาก่อนที่จะพบเป้าหมายภายใน Search space มักจะมีจำนวนมาก เนื้อที่หน่วยความจำที่ต้องใช้กับ A* มีขนาดใหญ่ จนกลายเป็นจุดอ่อนของวิธีนี้ เวลาในการทำงานก็ใช้มากเช่นเดียวกัน A* จึงไม่เหมาะกับปัญหาขนาดใหญ่

4.1.2 Memory-bounded heuristic search

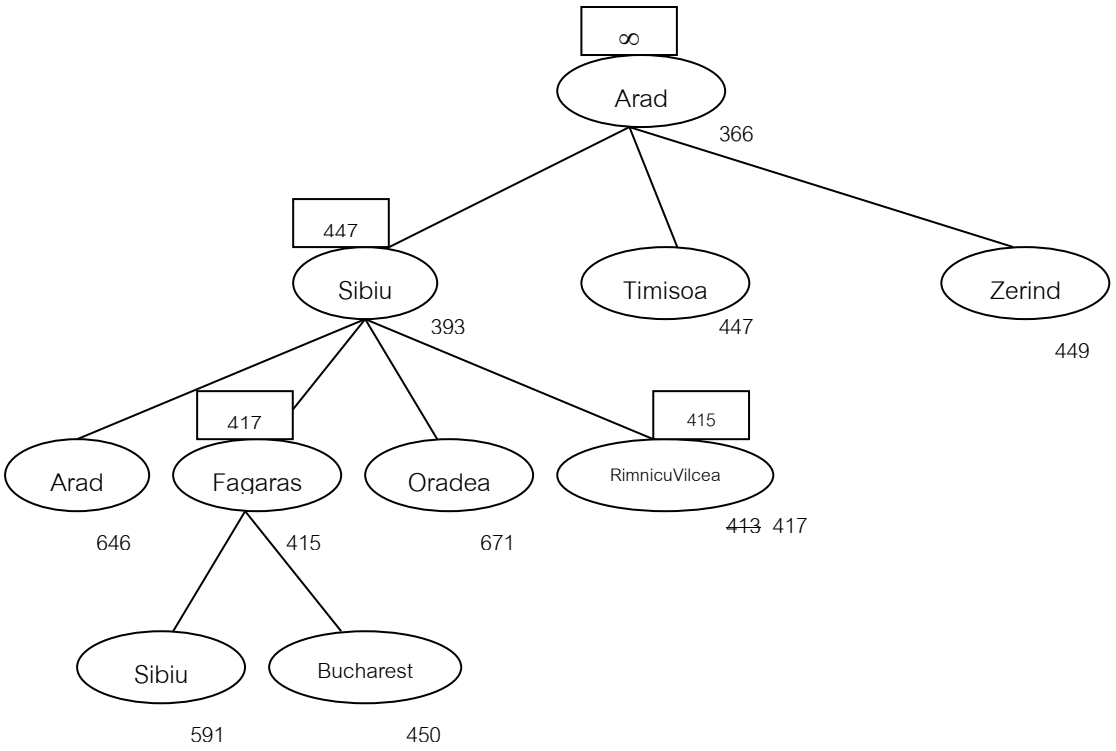
เป็นวิธีค้นหาที่ต้องการลดขนาดของหน่วยความจำที่ใช้ มีอยู่ 2 วิธีคือ RBFS (Recursive best-fist search) และ MA* (Memory-bounded A*)

4.1.2.1 RBFS เป็นอัลกอริทึมแบบเรียกซ้ำ (Recursive) อย่างง่าย มีการทำงานแบบ Best-first search ทั่วไป มีโครงสร้างการทำงานคล้ายกับ Depth-first search แต่แทนที่จะค้นหาลงไปตามเส้นทางที่กำลังใช้อยู่ วิธี RBFS จะเก็บค่า f ของเส้นทางอื่นที่เป็นค่าดีที่สุดเอาไว้ เส้นทางนี้จะเป็นของโหนดบรรพบุรุษของโหนดปัจจุบัน ถ้าโหนดปัจจุบันมีค่า f เกินค่านี้ การทำงานแบบเรียกซ้ำจะย้อนกลับไปเลือกเส้นทางอื่นที่ดีกว่าที่เก็บไว้ พร้อมกันนั้นก็จะแทนที่ค่า f ของโหนดในเส้นทางด้วยค่า f ที่ดีที่สุดของโหนดลูก ๆ ของมัน เพื่อให้จำได้ว่า Leaf ที่ดีที่สุดใน Subtree ที่ทิ้งไปนี้มีค่า f เท่าใด ต่อไปจะได้ตัดสินใจถูกว่าจะย้อนกลับมา Expand subtree นี้อีกครั้งดีหรือไม่ในภายหลัง กระบวนการของ RBFS แสดงได้ในรูปที่ 4.4

(a) หลังจาก Expand Arad, Sibiu, และ Rimnicu Vilcea

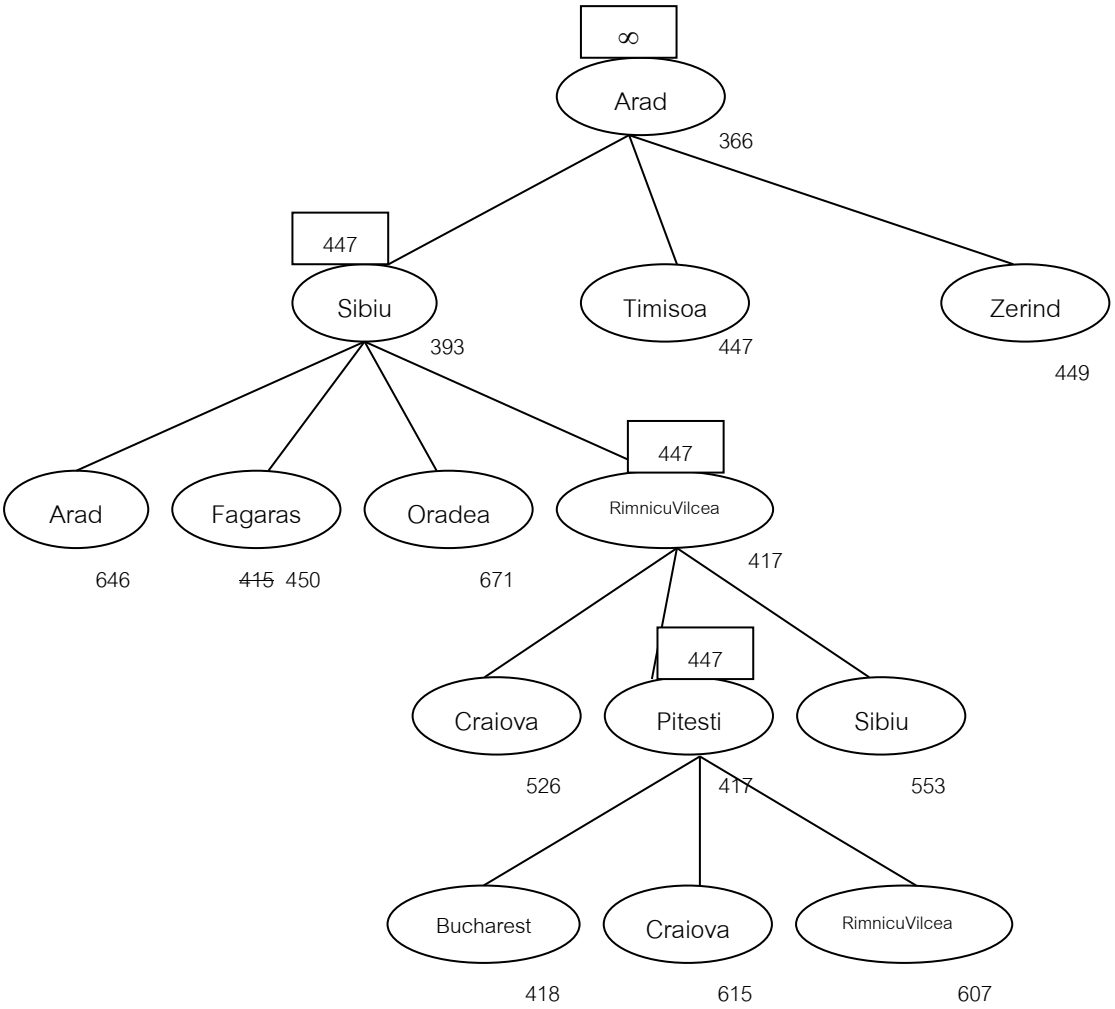


(b) หลังจากย้อนกลับมาที่ Sibiu และ Expand Fagaras



รูปที่ 4.4 ขั้นตอนของการค้นหาแบบ RBFS ในการหาเส้นทางในโรมาเนีย

(c) หลังจากเปลี่ยนกลับมาที่ Rimnicu Vilcea และ Expand Pitesti



รูปที่ 4.4 (ต่อ) ขั้นตอนของการค้นหาแบบ RBFS ในการหาเส้นทางในโรมาเนีย

จากรูป ค่า f-value ในการ Call แบบเรียกซ้ำ แสดงไว้ในกล่องสี่เหลี่ยมบนแต่ละโหนด รูป (a) แสดงเส้นทางผ่าน Rimnicu Vilcea จนถึงโหนด Leaf คือ Pitesti พบว่ามีค่า f เป็น 417 ซึ่งมากกว่าบรรพบุรุษ คือทางเลือกที่ผ่านมาแล้วได้แก่โหนด Fagaras (415) รูป (b) แสดงการเรียกซ้ำ (recursive) กลับไปยังเส้นทางของ Fagaras โดยทิ้ง Subtree ของ Pitesti เอาไว้ก่อน แต่นำค่า f ของโหนดนี้ คือ 417 ไปแทนที่ค่า f ของโหนด Rimnicu Vilcea แล้วมา Expand โหนด Fagaras แทน พบว่าโหนดลูกของ Fagaras ที่มีค่า f ดีที่สุดคือ 450 ซึ่งมากกว่าค่า f ของโหนดบรรพบุรุษ คือ Rimnicu Vilcea (417)

รูป (c) แสดงว่าการเรียกซ้ำต้องวนกลับไปหาโหนด Rimnicu Vilcea แล้วเก็บค่า f ที่ดีที่สุดของ Subtree ที่ทิ้งไว้ก่อน (450) โดยนำไปแทนที่ค่า f ของโหนด Fagaras ต่อไป Expand โหนด Rimnicu Vilcea จะได้โหนดที่ดีที่สุดคือ Pitesti ขณะนี้เส้นทางอื่นที่ดีที่สุด (Timisoara) ก็ยังมีค่า f เป็น 447 การ Expand จึงทำต่อไปจนถึงเป้าหมายคือ Bucharest

RBFS มีประสิทธิภาพดี แต่จะต้องทำงานซ้ำวนไปมาหลายครั้ง จะเห็นว่าตัวอย่างที่กล่าวมาแล้วนั้น มีเส้นทางผ่าน Rimnicu Vilcea แล้วเปลี่ยนใจ ลองไปเส้นทาง Fagaras แล้วเปลี่ยนใจกลับมาใหม่อีกครั้ง การเปลี่ยนใจนี้เกิดขึ้นเนื่องจากทุกครั้งที่เส้นทางที่ดีที่สุด ณ ขณะนั้นขยายไกลออกไปเรื่อย ๆ ก็จะมีโอกาสอย่างมากที่จะมีค่า f ขยับเพิ่มขึ้น เมื่อเกิดขึ้นแล้ว เส้นทางที่ดีเป็นอันดับ 2 ก็อาจกลายเป็นเส้นทางดีที่สุด การค้นหาจึงต้องย้อนกลับมา

RBFS จะ Optimal ถ้า h เป็นฮิวริสติกที่ยอมรับได้ การใช้หน่วยความจำของวิธีนี้เป็นฟังก์ชัน O(bd) แต่เนื่องจากไม่สามารถตรวจสอบสถานะซ้ำในระหว่างที่อยู่บนเส้นทางได้ จึงอาจจะต้องทำงานกับสถานะเดิมหลายครั้ง การใช้เวลาของ RBFS จึงวิเคราะห์ได้ยาก

4.1.2.2 SMA* (Simplified memory-bounded A*) มีวิธีการที่นำมาจาก A* คือ Expand โหนดที่ดีที่สุดไปจนถึงระดับ Leaf จนกว่าหน่วยความจำถูกใช้เต็มที่ ณ จุดนี้ เมื่อไม่สามารถเพิ่มโหนดใหม่เข้าไปใน Search tree ได้อีก SMA* ก็จะเลือกโหนดที่มีค่า f สูงที่สุดออกทิ้งไปทั้ง subtree แล้วเก็บค่า f นั้นไว้ที่ตัวโหนดแม่เพื่อสำรองไว้ (คล้ายกับ RBFS) เมื่อไหร่ที่เส้นทางอื่น ๆ ไม่มีค่า f ที่ดีกว่านี้ จึงจะนำ Subtree นี้กลับมาใช้ใหม่

โดยสรุป SMA* จะ Expand โหนดที่ดีที่สุด และตัดโหนดที่แย่ที่สุดทิ้งไป แต่ถ้าทุกโหนด Leaf มีค่า f เท่ากัน การจะ Expand และลบโหนดเดียวกันนั้นทำไม่ได้ SMA* อาจแก้ปัญหาโดย Expand โหนดใหม่ล่าสุดที่ดีที่สุด และเลือกลบโหนด leaf ที่แย่ที่สุดอันเก่าสุดทิ้งไป โอกาสที่โหนดทั้ง 2 ประเภทจะเป็นโหนดเดียวกันได้ ก็ต่อเมื่อใน Tree นั้น มีโหนด Leaf เพียงโหนดเดียวเท่านั้น และในกรณีนี้ Search tree จะต้องมีเส้นทางเดียวจากรากไปจนถึงโหนด Leaf ที่ทำให้หน่วยความจำเต็ม และถ้าโหนด Leaf นี้ยังไม่ใช่เป้าหมายอีก แสดงว่าไม่สามารถหาคำตอบได้ เพราะหน่วยความจำมีเท่านี้ โหนดเช่นนี้จะถูกตัดออกไปได้ทันที

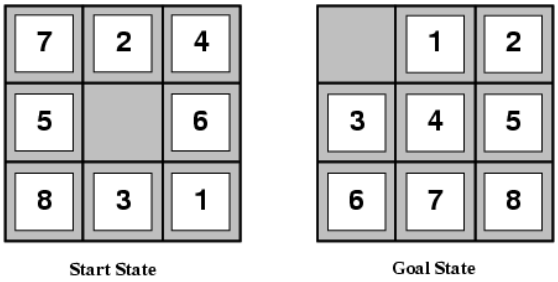
อย่างไรก็ตาม ถ้าเป็นปัญหาที่หาคำตอบได้ นั่นคือ ถ้า d เป็นระดับความลึกของโหนดเป้าหมายที่อยู่ตื้นที่สุดแล้ว และ Expand แล้วไม่เกินขนาดของหน่วยความจำ SMA* จะ Complete และถ้าเส้นทางนั้นเป็น Optimal solution path แล้ว SMA* จะ Optimal ถ้าไม่เช่นนั้น SMA* จะให้คำตอบที่ดีที่สุดเท่าที่จะหาได้เท่านั้น

4.2 ฟังก์ชันฮิวริสติก (Heuristic function)

จากตัวอย่างการเดินทางในโรมาเนีย ฟังก์ชันฮิวริสติกที่นำมาใช้ร่วมกับฟังก์ชันประเมินค่า คือ ระยะทางวัดเป็นเส้นตรงจากโหนดนั้น ไปยังโหนดเป้าหมาย (h_{SLD}) ฮิวริสติกที่ใช้ในแต่ละปัญหาจะแตกต่างกันไป ขึ้นอยู่กับความรู้ที่มีในปัญหานั้น

4.2.1 ฮิวริสติกของปัญหาเกม 8-puzzle

ปัญหาเกม 8-puzzle แสดงภาพเกมไว้ในรูปที่ 4.5



รูปที่ 4.5 เกม 8-puzzle

เมื่อทดลองเล่นเกม 8-puzzle ที่เลือกกระดานตัวอย่างมาแบบสุ่ม 1 กระดาน เป็นจำนวนหลาย ๆ ครั้ง พบว่าการไปถึงคำตอบของปัญหาตัวอย่างนี้ มีการขยับประมาณ 22 ขั้นตอน Search tree ของเกมนี้มีค่าเฉลี่ย Branching factor ประมาณ 3 โดยคิดจากการขยับแผ่นว่างดังนี้

ถ้าแผ่นว่างอยู่ตำแหน่งกลางกระดาน จะขยับได้ 4 แบบ

ถ้าแผ่นว่างอยู่ตำแหน่งมุมกระดาน จะขยับได้ 2 แบบ

ถ้าแผ่นว่างอยู่ตำแหน่งติดด้านของกระดาน จะขยับได้ 3 แบบ

ค่าเฉลี่ยของการเกิดแอคชันคือ (4+2+3)/3 = 3 คือค่าเฉลี่ยของ Branching factor

ถ้าค้นหาไปถึงระดับความลึก 22 จะต้องค้นหาโหนดจำนวน $3^{22} = 3.1 \times 10^{10}$ โหนด แต่ถ้าตัดโหนดซ้ำออก ก็จะช่วยลดขั้นตอนการค้นหาลงได้มาก เพราะโหนด (สถานะ) ที่เกิดขึ้นในเกมนี้ หรือสมาชิกของ State space หาได้จากการเรียงช่องตาราง 9 ช่องต่าง ๆ กันได้ 9! วิธี หรือเท่ากับ 362,880 สถานะที่แตกต่างกัน ซึ่งจะเห็นว่าถ้าทำงานกับโหนดที่ไม่ซ้ำ จะลดปริมาณโหนดที่ต้องค้นหาได้เป็นอย่างมาก แต่ถึงกระนั้น โหนดนับแสนโหนดก็ทำให้กระบวนการค้นหายุ่งยากมาก การพยายามหาฮิวริสติกมาช่วยเลือกโหนดที่จะ Expand อาจพิจารณาจาก 2 ฟังก์ชันดังนี้

1. ให้ฮิวริสติก h_1 คือจำนวนแผ่นที่อยู่ในตำแหน่งช่องผิด เช่นในรูปที่ 4.5 ทุกแผ่นวางอยู่ในตำแหน่งที่ผิดไปจากสถานะเป้าหมาย ดังนั้น ที่สถานะเริ่มต้น (ราก) มี $h_1 = 8$ ในกรณีนี้ h_1 เป็นฮิวริสติกที่ยอมรับได้ (ไม่ Overestimate) เพราะแผ่นที่อยู่ผิดที่จะต้องถูกเคลื่อนย้ายอย่างน้อย 1 ครั้ง (การคิดค่าใช้จ่ายของเกมนี้คือ คิดค่าใช้จ่ายเป็น 1 สำหรับการขยับแผ่นใด ๆ 1 ครั้ง)

2. ให้ฮิวริสติก h_2 คือผลรวมของระยะห่างของทุกแผ่นจากเป้าหมาย (นับระยะห่างในแนวนอนและแนวตั้งเท่านั้น) เรียกระยะห่างนี้ว่า City block distance หรือ Manhattan distance ค่า h_2 จะเป็นฮิวริสติกที่ยอมรับได้ เพราะแต่ละแผ่นจะขยับเข้าใกล้เป้าหมายได้ทีละ 1 ขั้นตอนเท่านั้น จากรูปที่ 4.5 หาค่า h_2 ได้ดังนี้

$$h_2 = 3+1+2+2+2+3+3+2$$
$$= 18$$

4.2.2 ผลของฮิวริสติกที่มีต่อการค้นหา

คุณภาพของฮิวริสติกดูได้จากค่า Effect branching factor (b*) ของการค้นหาที่ใช้ฮิวริสติกนั้นมาช่วย สมมุติว่ามีปัญหาเฉพาะปัญหาหนึ่ง ใช้ A* สร้างโหนดขึ้นมาได้ทั้งหมดเป็นจำนวน N โหนด และคำตอบอยู่ที่ความลึก d แล้ว จะได้ว่า Tree ที่มีความลึก d และมีกิ่งเป็นจำนวนเท่ากันอย่างสม่ำเสมอนั้นจะต้องมี Branching factor เป็น b* เพื่อให้ Tree นี้สามารถบรรจุได้ N+1 โหนด

ดังนั้น จะได้ว่า

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

ตัวอย่างเช่น ถ้า A* พบคำตอบที่ระดับความลึก 5 และสร้างโหนดขึ้นมาจำนวน 52 โหนด จะมี Effective branching factor เป็น 1.92 ค่านี้จะเปลี่ยนไปตามลักษณะของปัญหาแต่ละปัญหา การทดลองวัดค่า b* กับปัญหาชุดเล็กก่อน จะช่วยนำไปสู่การหาค่าฮิวริสติกที่ดีได้ ฮิวริสติกที่ออกแบบมาดีแล้วจะทำให้ค่า b* ใกล้เคียงกับ 1 ทำให้ปัญหาที่ค่อนข้างใหญ่ลดลงจนสามารถแก้ปัญหาได้ง่ายขึ้น

ตารางที่ 4.2 การเปรียบเทียบค่าใช้จ่ายและ Effective branching factor ในการค้นหาแบบ IDS และ A* โดยใช้ฮิวริสติก h_1 และ h_2

	Search cost			Effective branching factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	-	539	113	-	1.44	1.23
16	-	1301	211	-	1.45	1.25
18	-	3056	363	-	1.46	1.26
20	-	7276	676	-	1.47	1.27
22	-	18094	1219	-	1.48	1.28
24	-	39135	1641	-	1.48	1.26

มีการทดสอบค่า h_1 และ h_2 ในปัญหา 8-puzzle โดยสุ่มสร้างปัญหาขึ้นมา 1200 ปัญหา โดยมีความลึกของคำตอบอยู่ในช่วง 2 ถึง 24 แล้วใช้วิธี Iterative deepening search กับ A* search ที่ใช้ฮิวริสติกทั้ง 2 ฟังก์ชัน ผลที่ได้เป็นตามตารางที่ 4.2 ซึ่งแสดงจำนวนเฉลี่ยของโหนดที่สร้างขึ้นในกระบวนการ และ Effective branching factor ที่คำนวณได้ จะเห็นว่า h_2 เป็นฮิวริสติกที่ดีกว่า h_1 และถ้านาน ๆ ไป A* จะดีกว่า Iterative deepening search ซึ่งเป็นการค้นหาแบบไม่มีข้อมูล

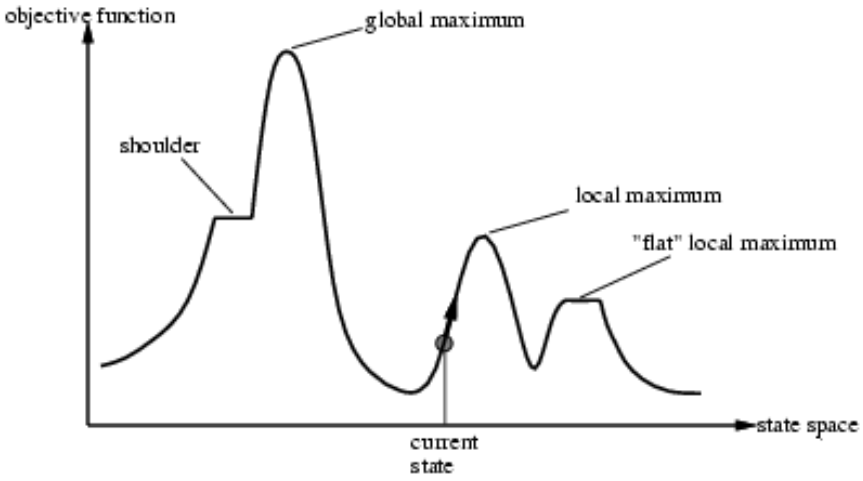
4.2.3 การสร้างฟังก์ชันฮิวริสติกที่ยอมรับได้

ฮิวริสติก h_1 และ h_2 ที่นำมาใช้กับ A* ต่างก็เป็นฮิวริสติกที่ดี การสร้างฮิวริสติกเช่นนี้ทำได้จากการประมาณค่าความยาวของเส้นทางที่จะไปสู่เป้าหมายของเกม 8-puzzle ที่ปรับให้ง่ายขึ้น แต่ก็สามารถนำมาใช้เป็นค่าโดยตรงของเกมได้ด้วย สมมุติว่าถ้ากฎการเล่นเกมเปลี่ยนไปโดยยอมให้แผ่นตัวเลขแต่ละแผ่นสามารถขยับเลื่อนไปได้ทุกที่ แทนที่จะขยับไปยังช่องว่างที่อยู่ติดกันเท่านั้น ค่า h_1 จะกลายเป็นจำนวนขั้นตอนที่แน่นอนในการเดินไปสู่เป้าหมายที่สั้นที่สุดทันที

เช่นเดียวกัน ถ้ายอมให้แผ่นตัวเลขขยับไปยังช่องติดกันได้ทุกทิศทุกทาง (รวมทั้งแนวทแยงมุม) แม้ว่าช่องนั้นจะไม่ใช่ช่องว่างก็ตาม h_2 จะกลายเป็นจำนวนขั้นตอนที่แน่นอนในการเดินไปสู่เป้าหมายที่สั้นที่สุดทันที ปัญหาที่ลดข้อจำกัดของการกระทำให้เหลือน้อยลงเช่นนี้เรียกว่าปัญหาผ่อนคลาย (Relaxed problem) ค่าใช้จ่ายของ Optimal Solution ของปัญหาผ่อนคลายนี้นำมาใช้เป็นฮิวริสติกที่ยอมรับได้ของปัญหาดั้งเดิม ฮิวริสติกนี้ยอมรับได้ เพราะโดยนิยามแล้ว Optimal solution ของปัญหาดั้งเดิมก็เป็นคำตอบของปัญหาผ่อนคลายด้วยเช่นกัน ดังนั้นค่าใช้จ่ายของปัญหาเดิมอย่างน้อยก็ต้องเท่ากันกับ Optimal solution ของปัญหาผ่อนคลาย แต่เนื่องจากฮิวริสติกที่หามาได้นี้เป็นค่าใช้จ่ายโดยตรงของปัญหาผ่อนคลาย (ที่สั้นที่สุดแล้ว) ค่านี้ต้องมีลักษณะ Consistency และจะไม่มีทาง Overestimate ค่าใช้จ่ายจริงได้

4.3 การค้นหาแบบ Hill-climbing

เป็นวิธีที่ทำงานวนซ้ำเป็นลูปเพื่อหาทางไปสู่สถานะที่มีค่า f สูงขึ้นเรื่อย ๆ เปรียบเหมือนเดินขึ้นเนินเขา และจะหยุดทำงานเมื่อขึ้นถึงยอด คือที่ซึ่งไม่มีโหนดใกล้เคียงที่มีค่าสูงกว่า อัลกอริทึมนี้ไม่ใช้ Search tree โครงสร้างข้อมูลของโหนดประกอบด้วยตัวสถานะและฟังก์ชันประเมินค่า f วิธีการแบบ Hill-climbing ไม่มองไกลมากนัก แต่จะดูแค่โหนดที่อยู่ใกล้กับโหนดปัจจุบัน ซึ่งอาจจะเรียกว่าเพื่อนบ้านของโหนดปัจจุบันก็ได้ ดูทิวทัศน์การเปรียบเทียบของ Hill-climbing ได้ในรูปที่ 4.6 เห็นได้ว่าในการปีนเขาอาจจะพบจุดสูงสุดในบริเวณหนึ่งซึ่งไม่ใช่จุดสูงสุดของทั้งหมด



รูปที่ 4.6 ทิวทัศน์ของ Hill-climbing

อัลกอริทึมของ Hill-climbing อย่างง่าย มีดังนี้

1. หาฟังก์ชันประเมินค่าที่สถานะเริ่มต้น และทดสอบว่าถ้าสถานะนี้เป็นสถานะเป้าหมาย หยุดทำงาน แต่ถ้าไม่ใช่ ทำต่อในข้อ 2

2. ทำลูปวนซ้ำดังต่อไปนี้ จนกว่าจะหาพบคำตอบ หรือจนกว่าจะไม่สามารถหาการกระทำใด ๆ ที่ทำได้กับสถานะขณะนั้นได้อีกต่อไป

ก. เลือกการกระทำอย่างใหม่มากระทำกับสถานะปัจจุบัน (Current state หรือ โหนด) เพื่อทำให้เกิดสถานะใหม่

ข. หาค่าของสถานะใหม่

(1) ถ้าสถานะใหม่นี้คือสถานะเป้าหมาย หยุดการทำงาน

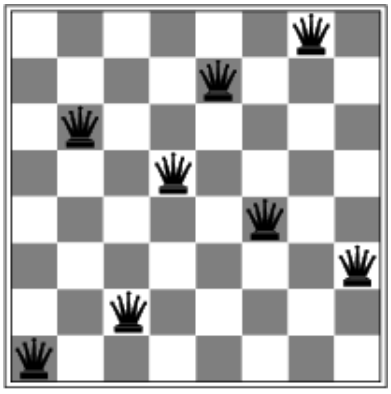
(2) ถ้าไม่ใช่สถานะเป้าหมาย แต่มีค่าดีกว่าสถานะปัจจุบัน เปลี่ยนสถานะนี้ให้กลายเป็นสถานะปัจจุบันแทน

(3) ถ้าหาค่าได้ไม่ดีไปกว่าสถานะปัจจุบัน ให้ทำงานวนลูป (ข้อ 2) ต่อไป

กรณีศึกษาของวิธีค้นหาแบบ Hill-climbing ได้แก่ปัญหา 8-queens (วางควีน 8 ตัวบนกระดานหมากรุก ให้ทุกตัวอยู่ในตำแหน่งปลอดภัยจากกันและกัน) อัลกอริทึมพื้นฐานเริ่มต้นจากกำหนดสถานะ แต่ละสถานะประกอบด้วยกระดานที่มีควีนวางอยู่ 8 ตัว โดยวางแถวละ1ตัว Successor function คือ สร้างสถานะใหม่ทั้งหมดเท่าที่เป็นไปได้ โดยขยับควีนเพียงตัวเดียวไปยังช่องอีกช่องหนึ่งในแถวเดียวกัน ดังนั้นแต่ละสถานะจะมีสถานะเป็นโหนดลูกจำนวน 8x7 = 56 Successor ฮิวริสติกฟังก์ชัน h คือจำนวนคู่ควีนที่โจมตีกันและกัน ค่า h ที่ต่ำที่สุดคือ 0 ซึ่งเป็น f ของคำตอบที่ต้องการ ตัวอย่างในรูป 4.7 เป็นสถานะของ 8-queens ซึ่งมี h = 17 จากรูปแสดงค่าของโหนดลูกทุก ๆ โหนดไว้ว่า ถ้าขยับไปที่ตำแหน่งใดแล้วจะมีค่า h เป็นเท่าใด โหนดลูกที่ดีที่สุดให้ค่า h = 12 ถ้ามีโหนดดีที่สุดหลายโหนดเช่นนี้ อัลกอริทึมของ Hill-climbing จะเลือกโหนดลูกมาโหนดหนึ่งแบบสุ่ม แล้วใช้โหนดนั้นพิจารณาต่อไป สำหรับโหนดที่แสดงในรูปที่ 4.8 นั้น มีค่า h=1 แต่โหนดลูกต่อจากนี้ มีค่า h สูงกว่า 1 ทั้งสิ้น

บางครั้งเรียกวิธีนี้ว่า Greedy Local search เพราะ Hill-climbing จะเลือกแต่โหนดข้างเคียงที่ดีที่สุด โดยไม่คิดถึงอนาคตว่าจะไปต่ออย่างไร ไปได้หรือไม่ แต่ก็พบว่านี่เป็นวิธีที่ได้ผลดีวิธีหนึ่ง Hill-climbing มักจะหาคำตอบได้เร็ว เนื่องจากสามารถปรับเปลี่ยนตัวเองจากสถานะที่ไม่ดีได้ง่าย ตัวอย่างเช่น จากสถานะในรูป 4.7 ใช้เพียง 5 ขั้นตอนเพื่อไปสู่สถานะในรูป 4.8 ซึ่งทำให้ค่า h=1 และเข้าใกล้คำตอบ

รูปที่ 4.7 สถานะของ 8-queens ซึ่งมีค่าฮิวริสติกฟังก์ชัน h=17 และแสดงค่า h สำหรับ Successor ทั้งหมดที่เป็นได้เกิดจากการขยับควีนไปตำแหน่งอื่นในคอลัมน์เดียวกัน



รูปที่ 4.8 สถานะของ 8-queens ซึ่งพบ Local maximum หรือในที่นี้คือ Local minimum เพราะ h=1 แต่โหนดลูกมีค่าสูงกว่าทั้งสิ้น

ข้อเสียของ Hill-climbing คือการทำงานมักจะต้องหยุดชะงักก่อนถึงเป้าหมาย เนื่องจากเหตุผลดังนี้

1. พบ Local maximum คือค่าที่มากที่สุดภายในอาณาบริเวณจำกัดบริเวณหนึ่ง เปรียบเหมือนพบยอดเนินในพื้นที่บริเวณแคบ ๆ ในบริเวณนี้มีค่าสูงกว่าสถานะใกล้เคียงรอบข้าง แต่ก็ยังต่ำกว่าค่าที่สูงสุดเหนือทั้งหมด (Global maximum) อัลกอริทึม Hill-climbing ที่มาถึงบริเวณที่มี Local maximum จะถูกดึงขึ้นสู่ยอดเนิน แต่เมื่อถึงยอดแล้วจะหยุดชะงัก เพราะไม่มีทางไปอีกแล้ว เช่นรูปที่ 4.8 เป็น Local maximum (หรือ Local minimum สำหรับค่าใช้จ่าย h) สถานะรอบข้างมีค่า h สูงกว่านี้ทั้งสิ้น การขยับควีนตัวใดก็ตามล้วนแต่ทำให้สถานการณ์แย่ลง

2. พบสันเขา (Ridge) เกิดจากการที่มี Local maximum เป็นชุดต่อเนื่องติด ๆ กันไป เหมือนกับปีนเขาขึ้นลงตามสันเขาจำนวนมาก โหนดอาจจะมีขึ้น ๆ ลง ๆ แต่ไม่ว่าจะขยับอย่างไร ก็ทำให้กลับไปยังจุดที่แย่ลงกว่าเดิม นั่นคือ ไม่อาจจะข้ามสันเขาเหล่านี้ไปได้

3. พบที่ราบ (Plateau) คือบริเวณที่ราบซึ่งหาค่าฟังก์ชันประเมินค่าได้เท่ากัน แต่ค่าที่หามาได้นั้นอาจจะเป็น Flat local maximum ตามที่แสดงในรูปที่ 4.6 ซึ่งทำให้ไปถึงยอดเขาไม่ได้ หรืออาจติดอยู่ที่ไหล่เขา (Shoulder ในรูปที่ 4.6) ในกรณีของไหล่เขายังเดินหน้าต่อถึงยอดเขาได้ แต่ก็มีโอกาสติดอยู่บนที่ราบนี้หลุดออกไปไม่ได้เช่นกัน

วิธีแก้ไขจัดการกับปัญหาเหล่านี้พอจะมีทางทำได้บ้าง แต่ไม่สามารถรับรองผลได้ทุกครั้ง มี 3 วิธีดังนี้

1. ย้อนกลับไปยังโหนดก่อนหน้า (การย้อนกลับมักจะใช้คำศัพท์ว่า Backtrack) แล้วทดลองไปยังเส้นทางอื่นที่ดูแล้วน่าจะใช้ได้ไม่แพ้ทางแรก ถ้าจะใช้วิธีนี้ ต้องเก็บข้อมูลเส้นทางที่ยังไม่ได้รับเลือกเอาไว้สำหรับการย้อนกลับ เป็นวิธีที่ดีสำหรับจัดการกับ Local maximum

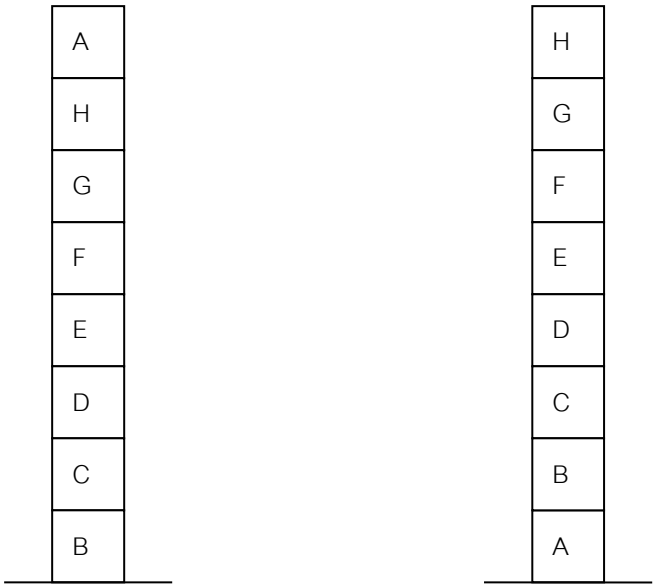
2. ทำการก้าวกระโดดไปยังทิศทางอื่นเพื่อพยายามหา Search space ในบริเวณอื่น เพื่อให้พ้นจากที่ราบ โดยใช้วิธีทำกระบวนการเดิมซ้ำ ๆ กันหลายครั้งในทิศทางเดียวกัน

3. มีกฎเพิ่มเติมก่อนการทดสอบโหนดว่าเป็นเป้าหมายหรือไม่ เพื่อแก้ไขการเลือกโหนดที่มีทิศทางหลากหลาย (ขึ้น ๆ ลง ๆ) ใช้ได้ดีเมื่อพบกับสันเขา

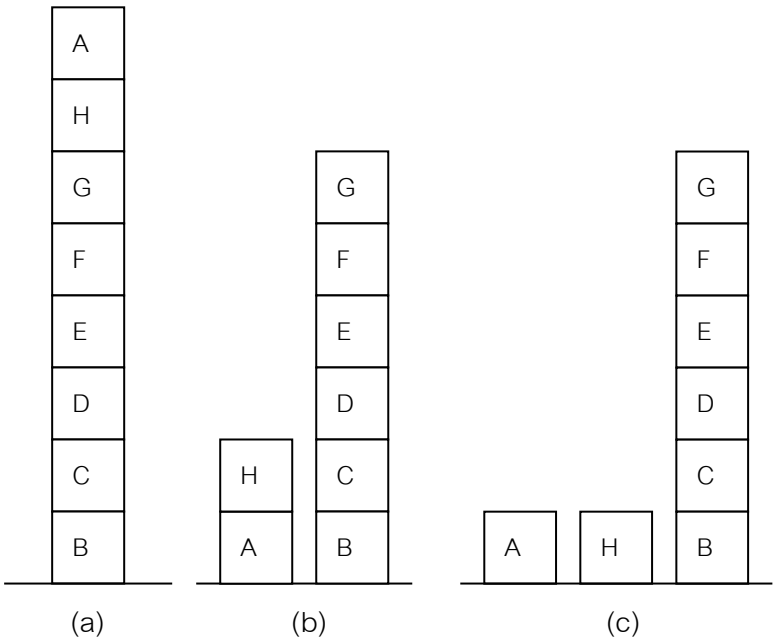
จากอัลกอริทึมของ Hill-climbing เท่าที่กล่าวมาจะเห็นว่า วิธีการค้นหาแบบนี้ไม่ Complete คืออาจจะหาเป้าหมายไม่พบ เนื่องจากการค้นหามาติดอยู่ที่บริเวณของ Local maximum อย่างไรก็ตาม กระบวนการทำงานของวิธีนี้หากยังไม่พบเป้าหมายก็ต้องพยายามต่อไปเรื่อย ๆ

ความสำเร็จของ Hill-climbing ขึ้นอยู่กับรูปร่างลักษณะของทิวทัศน์ของ State space เป็นอย่างมาก ภาพทิวทัศน์นี้เกิดจากการนำค่าฟังก์ชันประเมินค่าของแต่ละโหนดมาวาดรูป ถ้ามี Local maximum และที่ราบอยู่เป็นจำนวนน้อย โอกาสที่จะพบเป้าหมายอย่างรวดเร็วจะมีมาก โดยทั่วไปพบว่าวิธีค้นหานี้สามารถหาคำตอบได้ค่อนข้างเร็ว

ตัวอย่างอีกตัวอย่างหนึ่งของ Hill-climbing คือตัวอย่าง Block world ตัวอย่างนี้สมมุติให้มีกล่องจำนวนหนึ่ง ต้องการซ้อนกล่องนี้ให้เป้าหมายตามรูปที่ 4.9



รูปที่ 4.9 ปัญหา Hill-climbing ใน Block world



รูปที่ 4.10 การย้ายกล่องที่ทำได้ 3 แบบ หลังจากนำกล่อง A มาวางบนพื้นแล้ว

การกระทำที่ทำได้ใน Block world ได้แก่ การหยิบกล่องมาวางบนโต๊ะ และหยิบกล่องมาวางบนกล่อง โดยที่กล่องที่หยิบ และกล่องที่ถูกวางทับนั้นต้องไม่มีกล่องซ้อนอยู่

ฟังก์ชันฮิวริสติกที่ใช้ คือ กล่องใดวางอยู่ถูกตำแหน่ง (ซ้อนอยู่บนกล่องที่ถูกต้อง) ได้คะแนน +1 ส่วนกล่องที่วางผิดตำแหน่ง ได้คะแนน -1

จากตัวอย่างในรูปที่ 4.9 ค่า h ของสถานะเป้าหมายจะเป็น 8 ส่วนที่สถานะเริ่มต้น h เป็น 4 (เพราะได้ 1 คะแนนจากกล่อง C, D, E, F, G, H ซึ่งวางซ้อนกันถูกต้องตรงตามตำแหน่งที่สมควร และถูกหักกล่องละ 1 คะแนนสำหรับกล่อง A และ กล่อง B ซึ่งวางไม่ถูกที่)

จากสถานะเริ่มต้น มีการกระทำได้เพียงอย่างเดียวเท่านั้นคือย้าย A มาวางบนโต๊ะ ซึ่งสถานะนี้มีค่า h = 6 อัลกอริทึมจะยอมรับการกระทำนี้ ต่อจากสถานะนี้ มีการเคลื่อนไหวที่เป็นไปได้ 3 แบบ ตามรูปที่ 4.10 ทำให้เกิดสถานะลูก 3 สถานะ ค่า h ในสถานะ (a), (b), (c) เท่ากัน คือ 4 ถึงจุดนี้ Hill-climbing จะหยุดทำงาน เพราะทุกสถานะที่ไปได้ในขณะนี้มีค่า h ต่ำกว่าสถานะปัจจุบันทั้งสิ้น แสดงว่ามาถึงจุดที่เป็น Local maximum แล้ว (เพราะค่า 6 สูงที่สุดในบรรดาโหนดเพื่อนบ้านที่แวดล้อมอยู่) แต่ยังไม่ใช่ Global maximum

วิธีของ Hill-climbing มองไปข้างหน้าเพียงก้าวเดียว ไม่มองไปไกลกว่านั้น จึงทำให้ล้มเหลวไม่สามารถก้าวไปสู่ความสำเร็จได้ แต่ถ้าเราเลือกฟังก์ชันประเมินค่าหรือฮิวริสติกที่ดี วิธีนี้จะได้ผลดีขึ้น

สมมุติให้ฮิวริสติกใหม่ สำหรับแต่ละกล่องที่มีโครงสร้างฐานถูกต้อง (โครงสร้างฐานหมายถึงกล่องทุกกล่องที่วางเป็นฐานของกล่อง ๆ นี้ เริ่มจากบนพื้นขึ้นมา) ให้บวกคะแนนเท่ากับจำนวนกล่องที่วางถูกทั้งตั้งนั้น ส่วนกล่องที่มีโครงสร้างฐานไม่ถูก หักคะแนนเท่ากับจำนวนกล่องในตั้งที่ซ้อนกันไม่ถูกตั้งนั้น

เมื่อใช้ฟังก์ชันฮิวริสติกใหม่ สถานะเป้าหมายจะมีค่า h = 28 (กล่อง B ได้ 1 คะแนน กล่อง C ได้ 2 คะแนนตามลำดับ) ส่วนสถานะเริ่มต้นมีค่า h = -28 การย้ายกล่อง A ลงบนพื้นโต๊ะทำให้ได้ค่า h ในสถานะนี้เป็น -21 สถานะที่เกิดต่อจากนี้ ดูตามรูปที่ 4.10 มีค่า h ตามลำดับได้แก่ รูป (a) h = -28 รูป (b) h = -16 และ รูป (c) h = -15 อัลกอริทึมของ Hill-climbing จะเลือกการกระทำตามแบบรูป (c) ซึ่งเป็นทางเลือกที่ถูกต้อง และกระบวนการ Hill-climbing จะทำงานได้เป็นอย่างดี

แบบฝึกหัดบทที่ 4

1. จงติดตามการปฏิบัติงานของการค้นหาแบบ A* จากปัญหาการเดินทางในโรมาเนีย ซึ่งใช้ระยะทางวัดเป็นเส้นตรงเป็นฮิวริสติก แล้วแสดงลำดับของโหนดที่ได้รับการพิจารณาตามอัลกอริทึมที่ใช้ ตั้งแต่จุดเริ่มต้นจนถึงเป้าหมาย
2. มีคำกล่าวว่า Breadth-first search , Depth-first search และ Uniform-cost search เป็นกรณีพิเศษ (Special case) ของ Best-first search เพราะเหตุใดจึงกล่าวเช่นนั้น
3. จากปัญหา 8-puzzle และรูปที่กำหนดให้ข้างล่างนี้ จงหาค่าฮิวริสติกที่ไม่ Overestimate (หาค่า h_1 และ h_2 หรือ h_3 ในกรณีที่หาฟังก์ชันฮิวริสติกแบบอื่นได้)

1	2	3
8	5	6
4	7	

1	2	3
4	5	6
7	8	

4. จากรูปข้างล่างนี้ จงแสดงการทำงานเพื่อไปสู่เป้าหมายโดยวิธี RBFS (Recusive best-first search) โดยแสดงขั้นตอนให้ละเอียด กำหนดให้ตัวเลขที่กำกับแต่ละโหนดคือ ค่าของฟังก์ชันประเมินค่า