การใช้ AI ในเกมส์คอมพิวเตอร์

AI IN COMPUTER GAMES

ชัยวิรัตน์ ธรรมาวุฒิกุล

ทศพล ทับทิมเพชรางกูล

ปริญญานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

ปีการศึกษา 2552

ปริญญานิพนธ์ปีการศึกษา 2552

สาขาวิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

เรื่อง การใช้ AI ในเกมส์คอมพิวเตอร์

AI IN COMPUTER GAMES

ผู้จัดทำ

1.นายชัยวิรัตน์ ธรรมาวุฒิกุล รหัสนักศึกษา 49010188

2.นายทศพล ทับทิมเพชรางกูล รหัสนักศึกษา 49010310

_____ อาจารย์ที่ปรึกษา

(รศ.ดร.ครรชิต ไมตรี)

การใช้ AI ในเกมส์คอมพิวเตอร์

นายชัยวิรัตน์ ธรรมาวุฒิกุล 49010188
นายทศพล ทับทิมเพชรางกูล 49010310
รศ.ดร.ดรรชิต ไมตรี อาจารย์ที่ปรึกษา
ปีการศึกษา 2552

บทคัดย่อ

ในแวดวงของการพัฒนาซอฟต์แวร์นั้นผลงานที่ได้ใช่ว่าจะมีแต่เรื่องเคร่งเครียดและดูจริงจัง นั่นเพราะด้านหนึ่งของการพัฒนาซอฟต์แวร์ที่มีเสน่ห์น่าหลงใหลอยู่ในตัวของมันเองก็คือซอฟต์แวร์ด้านความบันเทิงซึ่งเกมก็คือหนึ่งในนั้น ในปัจจุบันเกมส่วนใหญ่ยังออกแบบมาให้ผู้เล่นสามารถเล่นเพียงคนเดียวได้ แต่รูปแบบของเกมที่เป็นการแข่งขันหรือเป็นการร่วมมือกันระหว่างผู้เล่นกำหนดให้คอมพิวเตอร์เล่นแทนผู้เล่นคนอื่น ซึ่งการพัฒนาให้คอมพิวเตอร์เล่นแทนได้เหมือนมีมนุษย์เป็นผู้เล่นนั้นไม่ใช้เรื่องง่าย

จากการศึกษาค้นคว้า เรื่องเกมส์ในศาสตร์ด้านปัญญาประดิษฐ์(AI) ทำให้เกิดวิธีการมากมายหลายแบบ ซึ่งบางแบบสามารถให้ผลลัพธ์ที่ดีแต่ก็มีจุดอ่อนเรื่องค่าใช้จ่ายเช่นหน่วยความจำและเวลา ในโครงงานนี้ได้เสนอวิธีการค้นหาแบบฮิวริสติกร่วมกับมินิแม็กซ์มาใช้ในการเล่นเกมส์โดยใช้อัลฟาเบต้าพรูนนิ่งช่วยเพิ่มประสิทธิภาพ ทำการวิเคราะห์แสดงให้เห็นการใช้คอมพิวเตอร์สวมบทบาทแทนมนุษย์คู่ต่อสู้ที่มีความเก่งกาจในระดับที่แตกต่างกัน จำเป็นที่จะต้องมีค่าใช้จ่ายที่แตกต่างกันด้วยเสมอ การใช้คอมพิวเตอร์สวมบทบาทแทนมนุษย์ แล้วมาเล่นกับมนุษย์ก็ไม่ใช่ว่าจะเล่นได้ดีเสมอไปเนื่องจากในบางครั้งอาจถูกคาดเดาพฤติกรรมได้ เพราะวิธีที่ดีที่สุดย่อมไม่หลากหลาย ซึ่งในความจริงแล้วคู่ต่อสู้ทางความคิดที่รับมือยากที่สุดก็คือมนุษย์ตัวจริงนั่นเอง เพราะฉะนั้นการเลือกใช้วิธีที่สามารถเลียนแบบมนุษย์ได้ใกล้เคียงมากที่สุด จะทำให้ได้ผลลัพธ์เสมือนมีคู่ต่อสู้ที่คู่ควรกันมาเล่นด้วย

AI IN COMPUTER GAMES

Mr.Chaiwirat Thammawutikul 49010188

Mr.Thossapon Thubtimpetcharangkul 49010310

Assoc.Prof.Dr.Kanchit Mitree Advisor

ABSTRACT

In the community of software development, software product is not always stressful and troublesome. Because, there is another branch of software development intimating which is entertainment software and game develops is one of them. Mostly, games in the present are designed to play with only single player. However the game requires another person to interact with player and that make us require AI. To make computer be able to act as human this is not an easy task.

By resent studies an amount of AI technique was being developed. Some of them make a good performance of solving problem but suffer varieties of cost such as memory and time and using the computer as human is not always the best choice because that sometime they are predictable, hence the best choice not variant ant that made the toughest opponent is still be human. So we choose selected the AI which identical the most of human to acquire the result that is like having friend playing with us.

กิตติกรรมประกาศ

ปริญญานิพนธ์ฉบับนี้สำเร็จได้ ด้วยความกรุณาจากอาจารย์ที่ปรึกษา รศ.ดร.ครรชิต ไมตรี ที่คอยให้คำแนะนำชี้แนะช่วยแก้ไขปัญหาตลอดจนให้ความรู้และประสบการณ์ที่เป็นประโยชน์แก่ข้าพเจ้า ซึ่งท่านต้องสละเวลาอันมีค่ามาตรวจสอบความคืบหน้าและให้คำปรึกษาในทุกๆสัปดาห์ และท่านยังเป็นแรงแรงผลักดันสำคัญ ที่ทำให้ปริญญานิพนธ์ฉบับนี้ เสร็จสมบูรณ์ในที่สุด

ขอกราบขอบพระคุณคณาจารย์สาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบังทุกๆท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้ให้แก่ข้าพเจ้า

ขอขอบคุณสาขาวิชาวิศวกรรมคอมพิวเตอร์ คณะวิศวกรรมศาสตร์ สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบังที่ได้สนับสนุนไฟฟ้า และอินเตอร์เน็ต รวมถึงปริญญานิพนธ์ของรุ่นพี่ สำหรับเป็นแหล่งศึกษาเพิ่มเติม

นายชัยวิรัตน์ ธรรมาวุฒิกุล

นายทศพล ทับทิมเพชรางกูล

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

ตาราง **หน้า**

สารบัญภาพ

รูป **หน้า**

สารบัญภาพ (ต่อ)

รูป **หน้า**

บทที่ 1
บทนำ

1.1 ความเป็นมาของปัญหา

การพัฒนาเกมส์ในปัจจุบันสิ่งที่จะกำหนดว่าเกมส์แต่ล่ะเกมส์ว่าน่าสนใจหรือไม่ นอกจากกราฟฟิกที่สวยงามแล้ว ความสามารถของปัญญาประดิษฐ์(AI) ก็มีส่วนสำคัญเหมือนกันที่จะทำให้เกมส์มีความน่าตื่นตาตื่นใจ เนื่องจากเกมส์บางเกมส์จำเป็นต้องมีผู้เล่นตั้งแต่ 2 คนขึ้นไปจึงจะสามารถเริ่มเกมส์ได้ AI จึงเป็นส่วนเติมเต็มที่จะทำให้เกมส์นั้นสามารถเล่นได้และให้ความรู้สึกเหมือนกับมีผู้เล่นที่เป็นมนุษย์มาร่วมเล่นด้วยจริงๆ

เนื่องจากในการแก้ปัญหาของเกมส์ต่างๆนั้น สามารถนำความรู้ทางด้าน AI ไปประยุกต์เพื่อหาคำตอบที่ต้องการได้อย่างมีประสิทธิภาพ แต่ปัญหาที่พบคือ ขนาดของปริภูมิสถานะที่มีขนาดใหญ่มาก จึงจำเป็นต้องมีการนำความรู้ทางด้าน AI นั้นมาประยุกต์ให้เข้ากับเกมส์นั้นๆเพื่อให้ได้คำตอบที่ถูกต้อง และ ได้วิธีที่มีประสิทธิภาพมากที่สุด

1.2 แนวทางแก้ไข

จากปัญหาที่ขาดผู้เล่น หรือขาดผู้เล่นที่มีความชำนาญเพื่อฝึกฝน ทำให้ทางผู้จัดทำจึงสร้างปัญญาประดิษฐ์มาใช้แทนตัวผู้เล่น แต่ทว่าตัวปัญญาประดิษฐ์นั้นต้องมีความสามารถ และประสิทธิภาพสูงคือ ข้อแรกสามารถเล่นเกมส์กับผู้เล่นโดยถูกกติกา และเก็บคะแนนเฉลี่ยได้เกินกว่าหรือเท่ากับค่าเฉลี่ยการเล่นเกมส์ทั่วไปของผู้เล่น ข้อสอง ระยะเวลาในการคิด และประมวลผลนั้นต้องไม่นานเกินไปจนผู้ใช้รับไม่ได้และต้องไม่เกิดการค้างขึ้น คือสามารถเล่นได้จนจบไม่ว่าจะเกิดกรณีใดๆ (ยกเว้น เกิดการผิดพลาดทางฮาร์ดแวร์) ปัญญาประดิษฐ์นี้จะทำงานได้โดยไม่ต้องมีผู้ดูแลระบบมาคอยควบคุมดูแล

1.3 วัตถุประสงค์ของโครงงาน

1) สำรวจและศึกษาวิธีการใช้ AI ในเกมส์คอมพิวเตอร์ง่ายๆ

2) วิเคราะห์วิธีการที่ศึกษามาโดยทดสอบกับเกมส์ง่ายๆที่เป็นไปได้

3) ศึกษาวิธีการสร้างเกมส์ด้วยเครื่องมือ XNA Game Studio

4) สามารถนำความรู้ทางด้าน AI มาใช้ในการพัฒนาเกมส์โอเทลโล่

1.4 ประโยชน์ที่คาดว่าจะได้รับ

1) เข้าใจถึงวิธีการของ AI ที่สามารถนำมาใช้สำหรับการพัฒนาเกมส์

2) สามารถประยุกต์วิธีการของ AI เพื่อให้เหมาะสมและมีประสิทธิภาพที่สุดในการพัฒนาเกมส์

3) สามารถสร้างเกมส์อย่างง่ายด้วยเครื่องมือ XNA Game Studio

1.5 ขอบเขตของการศึกษา

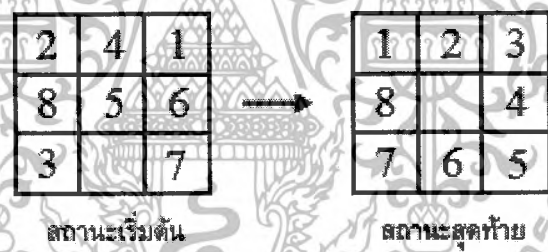
ทำการศึกษาและค้นคว้าเกี่ยวกับการค้นหาในปริภูมิสถานะ (state space) ด้วยวิธีต่างๆ โดยจะทำการสังเกตถึงความเป็นไปได้ ความเหมาะสม ข้อดีและข้อเสีย ของการนำวิธีต่างๆมาประยุกต์ใช้ในเกมส์แต่และแบบ จากนั้นทำการวิเคราะห์วิธีที่ศึกษามาโดยการทดลองสร้างเกมส์อย่างง่ายขึ้นเพื่อทำการทดลอง

บทที่ 2
ปริภูมิสถานะและการค้นหา

การแก้ปัญหาส่วนใหญ่ในปัญญาประดิษฐ์ จะมองปัญหาในรูปของการค้นหา (Search) ในการแก้ปัญหาโดยหลักการนี้ เราจะสร้างปริภูมิ (space) ขึ้นมาหนึ่งปริภูมิ สมาชิกแต่ละตัวในปริภูมินี้แทนตัวเลือกของคำตอบ จากนั้นก็ทำการค้นหาด้วยวิธีการค้นหาอย่างใดอย่างหนึ่งเพื่อให้ได้คำตอบที่ต้องการ

2.1 การนิยามปัญหาในรูปของการค้นหาในปริภูมิสถานะ

สมมติว่าเราต้องการแก้ปัญหา 8-Puzzle ซึ่งประกอบด้วยถาดขนาด 3x3 หน่วย ภายในถาดบรรจุแผ่นป้ายขนาด 1x1 หน่วยจำนวน 8 แผ่น ในแผ่นป้ายแต่ละแผนจะมีหมายเลขกำกับอยู่และมีช่องว่างอยู่ 1 ช่องที่แผ่นป้ายสามารถเคลื่อนเขามาแทนที่ได้ ปัญหาคือให้เลื่อนแผ่นป้ายเหล่านี้จากตำแหน่งเริ่มต้นให้เป็นตำแหน่งสุดท้ายซึ่งเป็นคำตอบตามรูปที่ 2.1



รูป 2.1 8-Puzzle

2.1.1 ในการนิยามปัญหาในรูปของการค้นหาในปริภูมิสถานะ (state space search) นั้นสามารถทำได้ดังนี้

1) นิยามปริภูมิสถานะโดยแสดงวัตถุทั้งหมดที่เกี่ยวของกับปัญหา ตัวอย่างเช่นในปัญหา 8-Puzzle เราอาจใช้รายการเพื่อแทนวัตถุเหล่านี้

2) ให้สถานะเริ่มต้น (initial state) แทนตำแหน่งเริ่มต้นของวัตถุทั้งหมดที่เกี่ยวข้องกับปัญหาและสถานะสุดท้ายหรือสถานะเป้าหมายหรือคำตอบ (final state or goal state or solution) แทนตำแหน่งสุดท้าย เช่นตัวอย่างในปัญหา 8-Puzzle ให้สถานะเริ่มต้นเป็น {2,4,1,8,5,6,3,0,7} และสถานะเป้าหมายเป็น {1,2,3,8,0,4,7,6,5} หากกฎที่ใช้

เปลี่ยนสถานะจากสถานะหนึ่งไปเป็นอีกสถานะหนึ่ง ซึ่งเราเรียกกฎเหล่านี้ว่าตัวกระทำการ (operator) ในที่นี้ตัวกระทำการก็คือการเลื่อนแผ่นป้าย

3) ใช้เทคนิคของการค้นหาในการเปลี่ยนจากสถานะเริ่มต้นไปยังสถานะเป้าหมาย ตัวอย่างตัวกระทำการในกรณีของ 8-Puzzle เป็นดังนี้ แผ่นป้ายหนึ่งๆ จะเลื่อนมาที่ช่องว่างได้ถ้าอยู่ติดกับช่องว่าง และสถานะจะเปลี่ยนจากสถานะเดิมไปยังสถานะใหม่ซึ่งตำแหน่งของแผ่นป้ายสลับที่กับตำแหน่งของช่องว่าง ตัวกระทำการมีทั้งหมด 4 ตัว ดังนี้

3.1) ด้านบนของช่องว่างมีแผ่นป้าย -> สลับตำแหน่งของช่องว่างกับแผ่นป้าย

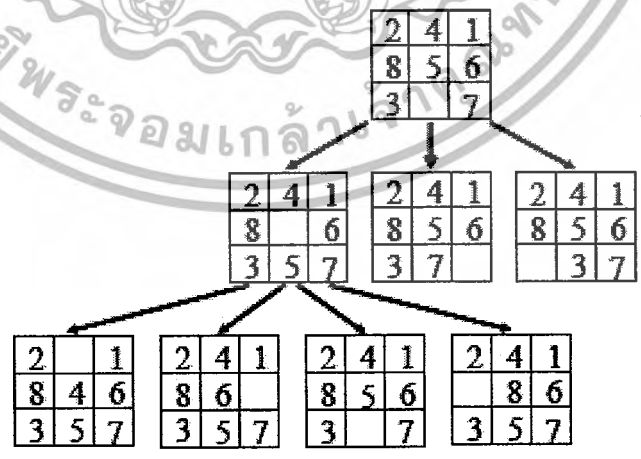
3.2) ด้านขวาของช่องว่างมีแผ่นป้าย -> สลับตำแหน่งของช่องว่างกับแผ่นป้าย

3.3) ด้านล่างของช่องว่างมีแผ่นป้าย -> สลับตำแหน่งของช่องว่างกับแผ่นป้าย

3.4) ด้านซ้ายของช่องว่างมีแผ่นป้าย -> สลับตำแหน่งของช่องว่างกับแผ่นป้าย

ตัวกระทำการที่ 3.1) มีความหมายว่าถ้าด้านบนของช่องว่างมีแผ่นป้าย ให้สลับตำแหน่งของช่องว่างกับแผ่นป้าย และจะเกิดสถานะใหม่ซึ่งมีตำแหน่งของช่องว่างกับแผ่นป้ายสลับที่กัน

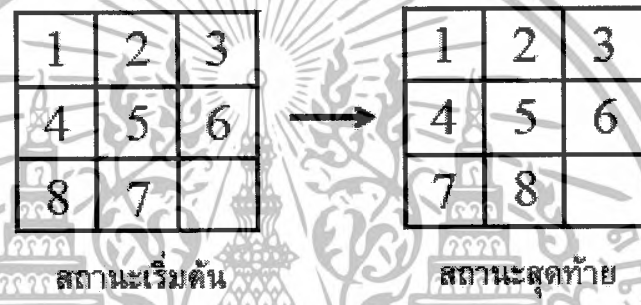
เมื่อเรานิยามตัวกระทำการเหล่านี้แล้ว ปริภูมิสถานะก็จะเกิดขึ้นโดยปริยาย และเราจะเห็นปริภูมิสถานะว่าสถานะใดเชื่อมต่อกับสถานะใด ก็เมื่อเราเริ่มจากสถานะเริ่มต้นหนึ่งตัวแล้วใช้ตัวกระทำการเหล่านี้สร้างการเชื่อมต่อของสถานะ ในกรณีของปัญหา 8-Puzzle เมื่อใช้ตัวกระทำการด้านบน เริ่มจากสถานะเริ่มต้นในรูป 2.1 จะสร้างปริภูมิสถานะบางส่วนดังแสดงในรูป 2.2



รูป 2.2 บางส่วนของปริภูมิสถานะในปัญหา 8-Puzzle

จากรูปจะเห็นว่าการเชื่อมต่อของสถานะถูกกำหนดโดยตัวกระทำการ ดังนั้นในปัญหาหนึ่งๆ เมื่อเรานิยามตัวกระทำการได้แล้ว เราก็จะได้ปริภูมิของสถานะสำหรับปัญหานั้นๆอย่างไรก็ดี เราจะพบว่าปริภูมิสถานะโดยทั่วไปจะมีลักษณะเป็นโครงสร้างแบบกราฟมากกว่าจะเป็นแบบต้นไม้ เนื่องจากบางสถานะอาจเกิดซ้ำกันได้ เช่นในกรณีของตัวอย่างในรูป 2.2 จะเห็นได้ว่าสถานะที่ 3 (นับจากซ้าย) ที่แถวล่างสุดซ้ำกับสถานะเริ่มต้น ซึ่งการเป็นกราฟนี้เองทำให้การค้นหาในปริภูมิสถานะมีความยุ่งยากมากขึ้นกว่าปริภูมิที่เป็นแบบต้นไม้

นอกจากนั้นสถานะเริ่มต้นตัวหนึ่งที่กำหนดให้ อาจไม่สามารถนำไปสู่สถานะเป้าหมายบางตัวได้ เนื่องจากสถานะเป้าหมายนั้นไม่มีเส้นทางที่เชื่อมต่อจากสถานะเริ่มต้นตัวอย่างเช่นในรูป 2.3



รูป 2.3 ตัวอย่างคำตอบที่เข้าถึงไม่ได้ด้วยสถานะเริ่มต้น

จากที่กล่าวข้างต้นจะเห็นได้ว่าการแก้ปัญหาพื้นฐานทางปัญญาประดิษฐ์แบบหนึ่งคือการมองปัญหาในรูปแบบของการค้นหาในปริภูมิสถานะ โดยเริ่มจากการนิยามสถานะนิยามโครงสร้างของมูลที่เก็บวัตถุที่เกี่ยวของกับปัญหากำหนดสถานะเริ่มต้นและสถานะเป้าหมาย รวมทั้งหาตัวกระทำการว่าจะต้องใช้ตัวกระทำการอะไรบ้างในการนิยามสถานะจากนั้นก็เป็นการเลือกเทคนิคการค้นหาที่เหมาะสมกับปัญหาที่เราต้องการ หัวข้อต่อไปนี้จะกล่าวถึงเทคนิคการค้นหาแบบต่างๆ ที่สามารถนำมาใช้ในการค้นหาในปริภูมิสถานะ

2.2 เทคนิคการค้นหาในปัญญาประดิษฐ์

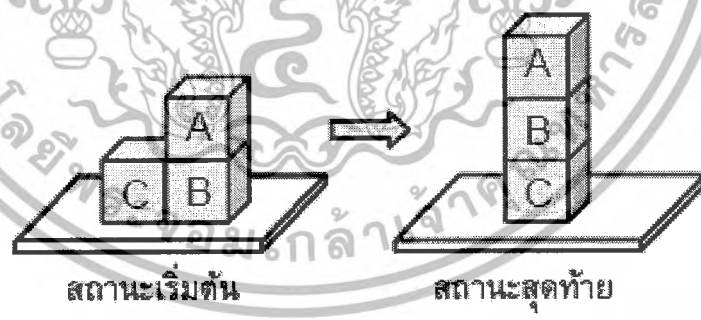
เทคนิคการค้นหาสามารถแบ่งได้เป็นสองประเภทหลักๆ คือ การค้นหาแบบบอด (blind search) ซึ่งเป็นเทคนิคการค้นหาที่ไม่มีตัวช่วยในการค้นหา แต่จะมีรูปแบบการค้นหาที่แน่นอนตายตัว เช่น จะค้นหาสถานะจากบนลงล่างในปริภูมิค้นหา เป็นต้น ส่วนเทคนิคการค้นหาอีกประเภทหนึ่งคือการค้นหาแบบฮิวริสติก (heuristic search) ซึ่งจะใช้ความรู้รูปแบบหนึ่งที่เรียกว่าฮิวริสติกมาช่วยทำให้การค้นหามีประสิทธิภาพยิ่งขึ้น

การค้นหาแบบบอดสามารถแบ่งย่อยได้ดังนี้คือ การค้นหาทั้งหมด (exhaustive search)หมายถึง การค้นหาทั้งหมดทั่วทั้งปริภูมิสถานะและการค้นหาบางส่วน (partial search) เป็นการค้นหาเพียงบางส่วนของปริภูมิสถานะ ปัญหาส่วนใหญ่ทางปัญญาประดิษฐ์มีปริภูมิสถานะที่มีขนาดใหญ่มาก ทำให้เราไม่สามารถค้นหาได้ทั่วทั้งปริภูมิ จำเป็นต้องค้นหาเพียงบางส่วนของปริภูมิเท่านั้น ดังนั้นจึงมีความเป็นไปได้ว่าคำตอบที่ได้อาจไม่ใช่คำตอบที่ดีที่สุด

การค้นหาเพียงบางส่วนโดยการค้นหาแบบบอดนั้นสามารถแบ่งได้เป็นสองประเภทคือการค้นหาแนวกว้างก่อน (breadth-first search) ซึ่งเป็นการค้นหาในแนวกว้างก่อนเมื่อพิจารณาจากโครงสร้างต้นไม้ของปริภูมิสถานะ และการค้นหาแนวลึกก่อน (depth-first search) คือหาแนวลึกก่อนเมื่อพิจารณาจากโครงสร้างต้นไม้ ส่วนการค้นหาแบบฮิวริสติก (heuristic search) มีหลายเทคนิคด้วยกัน เช่น อัลกอริทึมปีนเขา (hill-climbing search) อัลกอริทึมอบเหนียวจำลอง (simulated annealing algorithm) การค้นหาดีสุดก่อน (best-first search) การค้นหา A* (A* search) เป็นต้น

2.3 การค้นหาแบบบอด

ส่วนนี้อธิบายอัลกอริทึมการค้นหาแบบบอด (blind search) โดยจะเริ่มจากการค้นหาแนวกว้างก่อน แล้วต่อด้วยการค้นหาแนวลึกก่อน โดยตัวอย่างที่ใช้เพื่ออธิบายอัลกอริทึมคือปัญหาโลกของบล็อก (block world problem) ซึ่งแสดงในรูป 2.4 ด้านล่างนี้



รูป 2.4 ปัญหาโลกของบล็อก

ปัญหาคือกำหนดสถานะเริ่มต้นของการจัดเรียงตัวของบล็อกให้ต้องการจัดเรียงใหม่ให้ได้ตามสถานะสุดท้ายโดยมีตัวกระทำการดังนี้

1) บล็อก X ไม่มีบล็อกอื่นทับ วาง X บนโต๊ะ

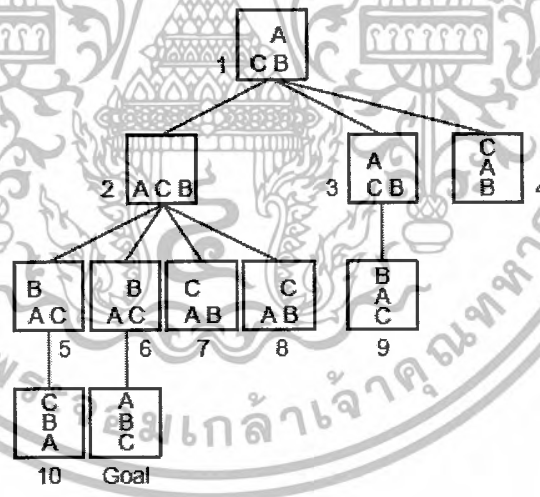
2) บล็อก X และ Y ไม่มีบล็อกอื่นทับ วาง X บน Y

โดยที่ X และ Y เป็นตัวแปรและสามารถแทนที่ด้วย ‘A’, ‘B’ หรือ ‘C’ เช่นเมื่อใช้ตัวกระทำการ 1) กับสถานะเริ่มต้น จะได้ว่าถ้าบล็อก ‘A’ ไม่มีบล็อกอื่นทับ แล้ววาง ‘A’ บนโต๊ะได้ เป็นต้น

2.3.1 การค้นหาแนวกว้างก่อน

ในการค้นหาแนวกว้างก่อน (breadth-first search) นี้ สถานะทุกตัว (ซึ่งบางทีเรียกว่าบัพ (node) เมื่อมองปริภูมิค้นหาอยู่ในรูปของต้นไม้) ที่อยู่ในระดับเดียวกันของต้นไม้จะถูกตรวจสอบก่อนสถานะที่อยู่ในระดับถัดไป วิธีการของการค้นหาแบบนี้ทำโดยเริ่มจากการสร้างสถานะลูกของสถานะเริ่มต้นก่อน แล้วตรวจสอบว่ามีสถานะใดที่เป็นสถานะสุดท้ายหรือไม่ ถ้าหากว่ามีก็เป็นอันว่าการค้นหาสิ้นสุด ถ้าไม่มีก็จะสร้างสถานะลูกของสถานะเหล่านั้น แล้วทำการตรวจสอบสถานะลูกทุกตัวของสถานะเหล่านั้น ถ้าพบสถานะสุดท้ายการค้นหาก็สิ้นสุด ถ้าไม่พบก็สร้างสถานะลูกของสถานะเหล่านั้นต่อไปอีก ทำเช่นนี้ไปเรื่อยๆ จนกว่าจะพบสถานะสุดท้ายหรือจนไม่สามารถสร้างสถานะลูกใหม่ได้อีก

ในกรณีของปัญหาโลกของบล็อก เริ่มจากสถานะเริ่มต้น เมื่อเราค้นหาด้วยการค้นหาแบบแนวกว้างก่อน ก็จะได้สถานะที่เกิดขึ้นดังรูป 2.5 ในตัวอย่างนี้กำหนดว่าลำดับของแถวไม่มีความสำคัญ กล่าวคือสถานะ ‘A C B’ เท่ากับสถานะ ‘C B A’ เป็นต้น และในการสร้างสถานะจะไม่สร้างสถานะซ้ำเดิม เช่นจากสถานะที่ 2 เราสามารถสร้างสถานะลูกของมันตัวหนึ่งซึ่งเท่ากับสถานะที่ 1 แต่จะไม่นำสถานะนี้ใส่ลงเป็นสถานะลูกของสถานะที่ 2 เนื่องจากไปซ้ำกับสถานะที่ 1



รูป 2.5 การค้นหาแบบแนวกว้างก่อนในปัญหาโลกของบล็อก

ตัวเลข 1, 2, 3, … ในรูปด้านบนแสดงลำดับของสถานะที่ถูกสร้างขึ้น และ ‘Goal’ แสดงสถานะเป้าหมายหรือคำตอบ โดยเริ่มจากสถานะที่ 1 ซึ่งเป็นสถานะเริ่มต้น จากนั้นใช้ตัวกระทำการ (1) และตัวกระทำการ (2) และแทนที่ตัวแปร X และ Y ในตัวกระทำการทั้งสองให้เป็น ‘A’, ‘B’ หรือ ‘C’ ตามลำดับจะได้สถานะลูกเป็นสถานะที่ 2 สถานะที่ 3 และ สถานะที่ 4 ตามลำดับ เมื่อสร้างสถานะลูกของสถานะที่ 1 ครบทุกตัวแล้ว ก็จะลงมาในระดับถัดไปเพื่อสร้างสถานะลูกของสถานะที่ 2, 3 และ 4 เป็นเช่นนี้ไปจนกว่าจะได้สถานะสุดท้ายที่ต้องการ ซึ่งจะเห็นได้ว่าการค้นหาจะทำใน

แนวกว้างทีละระดับตั้งแต่บนลงล่าง จากตัวอย่างเราได้สถานะสุดท้ายเป็นสถานะที่ 11 และไม่ต้องทำการค้นหาต่อเพราะการค้นหาแบบนี้เป็นการค้นหาเพียงบางส่วนเท่านั้นแม้ว่าอาจจะมีเส้นทางอื่นอีกที่สามารถนำไปสู่สถานะสุดท้ายได้ อัลกอริทึมของการค้นหาแนวกว้างก่อนแสดงดังรูป 2.6

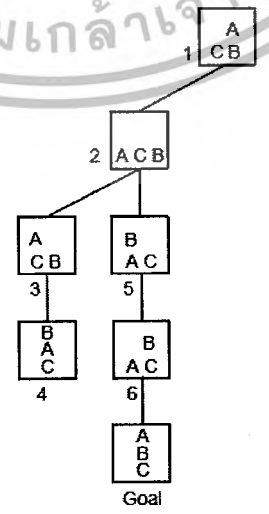
```
Algorithm: Breadth-First Search

1. Node-list := {initial state}
2. UNTIL a goal state is found or Node-list is empty DO
   2.1 Remove the first element from Node-list and call
       it E.
   2.2 For Each operator matching E DO
       2.2.1 Apply the operator to generate a new state.
       2.2.2 IF the new state is a goal state THEN quit
                 and return this state
             ELSE add the state to the end of Node-list.
```

รูป 2.6 อัลกอริทึมการค้นหาแนวกว้างก่อน

2.3.2 การค้นหาแนวลึกก่อน

การค้นหาแนวลึกก่อน (depth-first search) จะสร้างสถานะในแนวลึกทางด้านมุมซ้ายล่างก่อน (ดูรูป 2.7 ประกอบ) ถ้าสถานะตามแนวดิ่งถูกสร้างหรือกระจายจนหมดและยังไม่ได้คำตอบ ก็จะไล่กลับขึ้นด้านบนเพื่อหาเส้นทางอื่นที่จะเป็นไปได้การค้นหาแบบแนวกว้างก่อนกับการค้นหาแนวลึกก่อนนั้น ไม่สามารถบอกได้ว่าการค้นหาแบบไหนจะได้คำตอบเร็วกว่ากัน ขึ้นอยู่กับว่าคำตอบของเราอยู่บริเวณไหนในปริภูมิสถานะ (กรณีของตัวอย่างในรูป 2.7 นั้น การค้นหาแนวลึกก่อนพบคำตอบเร็วกว่า โดยสร้างสถานะทั้งสิ้น 7 ตัว ซึ่งคำตอบอยู่ที่บริเวณมุมซ้ายล่างของปริภูมิสถานะ)



รูป 2.7 การค้นหาแบบแนวลึกก่อนในปัญหาโลกของบล็อก

```
Algorithm: Depth-First Search

1. IF initial state = goal state THEN quit and return
       success
   ELSE UNTIL success or failure DO
   1.1 Generate a successor, E, of the initial state
       IF there are no more successors
       THEN return failure
       ELSE call Depth-First Search with E as the
            initial state.
   1.2 IF success is return THEN return success
       ELSE continue in this loop.
```

รูป 2.8 อัลกอริทึมการค้นหาแนวลึกก่อน

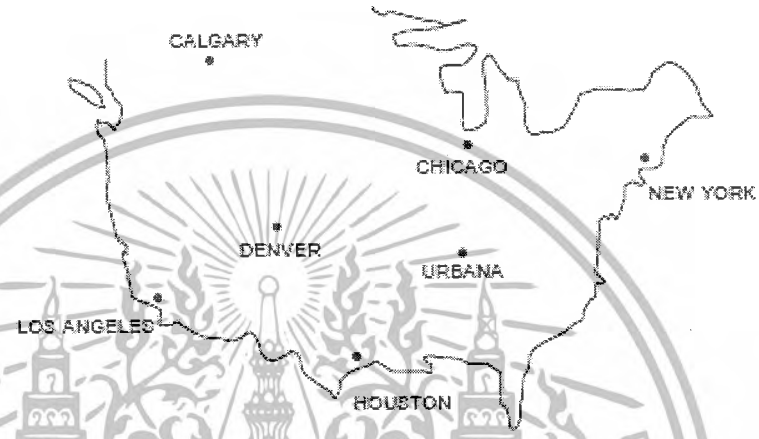
ข้อดีข้อเสียของอัลกอริทึมการค้นหาแนวลึกก่อน เมื่อเทียบกับการค้นหาแนวกว้างก่อน เป็นดังต่อไปนี้

ตาราง 2.1 เปรียบเทียบอัลกอริทึมการค้นหาแนวลึกก่อนและแนวกว้างก่อน

การค้นหาแนวลึกก่อน	การค้นหาแนวกว้างก่อน
1) ใช้หน่วยความจำน้อยกว่า เพราะว่าสถานะในเส้นทางค้นหาปัจจุบันเท่านั้นที่ถูกเก็บ (ในขณะใดๆ จะเก็บเส้นทางเดียว พอจะไปเส้นทางอื่นเส้นทางที่ผ่านมาก็ไม่จำเป็นต้องเก็บ)	1) ใช้หน่วยความจำมากเพราะต้องเก็บสถานะไว้ทุกตัวเพื่อหาเส้นทางจากสถานะเริ่มต้นไปหาคำตอบ
2) อาจจะติดเส้นทางที่ลึกมากโดยไม่พบคำตอบ เช่นในกรณีที่เส้นทางนั้นไม่มีคำตอบและเป็นเส้นทางที่ยาวไม่สิ้นสุด จะทำไม่สามารถไปเส้นทางอื่นได้	2) ไม่ติดเส้นทางที่ลึกมาก ๆ โดยไม่พบคำตอบ
3) ถ้าคำตอบอยู่ในระดับ n+1 สถานะอื่นทุกตัวที่ระดับ 1ถึงระดับ n ไม่จำเป็นต้องถูกกระจายจนหมด	3) ถ้าคำตอบอยู่ในระดับ n+1 สถานะทุกตัวที่ระดับ 1ถึงระดับ n จะต้องถูกกระจายจนหมด ทำให้มีสถานะที่ไม่จำเป็นในเส้นทางที่จะไปสู่คำตอบถูกกระจายออกด้วย
4) เมื่อพบคำตอบไม่สามารถรับประกันได้ว่าเส้นที่ได้เป็นเส้นทางที่สั้นที่สุดหรือไม่	4) ถ้ามีคำตอบจะรับประกันได้ว่าจะพบคำตอบแน่ๆ และจะได้เส้นทางสั้นที่สุดด้วย

2.4 การค้นหาแบบฮิวริสติก

การค้นหาประเภทฮิวริสติกนี้จะใช้ความรู้แบบหนึ่งที่เรียกว่าฮิวริสติกมาช่วยในการค้นหาให้มีประสิทธิภาพมากขึ้น โดยฮิวริสติกตัวนี้จะช่วยชี้แนะว่ากระบวนการค้นหาควรจะเลือกเส้นทางใดหรือสถานะใดเพื่อทำการค้นหาต่อไปให้ได้คำตอบอย่างมีประสิทธิภาพ พิจารณาปัญหาการเดินทางของพนักงานขาย (traveling salesman problem) ซึ่งแสดงในรูป 2.9



รูป 2.9 ปัญหาการเดินทางของพนักงานขาย

ในตัวอย่างของปัญหานี้มีเมือง 7 เมือง พนักงานขายต้องการเดินทางไปให้ได้ครบทั้ง 7เมือง และกลับมายังจุดเริ่มต้นโดยให้ได้ระยะทางโดยรวมสั้นที่สุด วิธีหนึ่งที่ทำได้คือ หาเส้นทางทั้งหมดที่เป็นไปได้ซึ่งจะมีด้วยกันทั้งสิ้น (7-1)!/2 (=360) แบบจากนั้นวัดแต่ละเส้นทางว่าใช้ระยะทางเท่าไร แล้วก็เลือกเส้นทางที่สั้นที่สุดวิธีการนี้ไม่สามารถคำนวณได้อย่างมีประสิทธิภาพในทางปฏิบัติ เมื่อจำนวนเมืองมีมากขึ้น เช่นถ้ามีเมือง 100 เมือง จะมีเส้นทางที่เป็นไปได้ทั้งสิ้น 4.67×10^{155}แบบ

ถ้าเราใช้สามัญสำนึกโดยคาดเดาอย่างมีเหตุผลว่า เมื่อเราต้องการระยะทางโดยรวมสั้นที่สุด เราก็น่าจะเลือกเมืองที่อยู่ใกล้มากที่สุดกับเมืองที่เราอยู่ในปัจจุบัน แล้วเดินทางไปเมืองนั้นก่อน เมื่อไปถึงเมืองนั้นแล้วค่อยทำในทำนองเดียวกันอีกว่าจะเดินไปยังเมืองที่ใกล้ที่สุดเมืองถัดไปทำเช่นนี้จนกระทั่งเดินทางครบทุกเมือง ก็น่าจะได้ระยะทางโดยรวมสั้นที่สุดแม้ว่าวิธีการเช่นนี้จะทำงานได้อย่างมีประสิทธิภาพ และคำตอบที่ได้มีแนวโน้มว่าจะดี แต่อย่างไรก็ดี คำตอบที่ได้โดยวิธีนี้อาจไม่เป็นเส้นทางที่สั้นที่สุดก็ได้วิธีการเช่นนี้ก็คือการนำความรู้แบบหนึ่งมาแก้ไขปัญหา ความรู้แบบนี้อาจไม่ใช่ความรู้ที่สมบูรณ์ แต่ก็พอที่จะนำมาแก้ไขปัญหาให้เราได้ และช่วยแนะให้เรารู้ว่าควรจะค้นหาเส้นทางอย่างไร เราเรียกว่าความรู้ที่ไม่สมบูรณ์หรือการคาดเดาอย่างมีเหตุผลแบบนี้ว่า ฮิวริสติก

การค้นหาแบบฮิวริสติกคือการค้นหาที่นำความรู้ประเภทนี้มาใช้ช่วยชี้แนะเส้นทางในการค้นหาคำตอบ โดยมีลักษณะเด่นดังนี้

1) เป็นเทคนิคที่ใช้เพิ่มประสิทธิภาพของกระบวนการค้นหา โดยอาจจะต้องย่อให้ขาดความสมบูรณ์ไปบ้าง คืออาจไม่พบคำตอบที่ถูกต้อง แม้ว่าในปริภูมิสถานะจะมีคำตอบนี้อยู่

2) การนำฮิวริสติกมาใช้จะต้องนำมาใช้ในรูปแบบที่วัดค่าได้อย่างง่าย ซึ่งมักทำโดยนิยามฮิวริสติกให้อยู่ในรูปแบบของฟังก์ชัน เราเรียกว่าฟังก์ชันฮิวริสติก (heuristic function) ซึ่งเป็นฟังก์ชันที่คำนวณค่าจากสถานะไปยังตัวเลขที่ชี้ว่าสถานะนั้นเข้าใกล้สถานะเป้าหมายมากเท่าไร (ยิ่งมากเท่าไร ยิ่งมีโอกาสที่จะเปลี่ยนเป็นสถานะเป้าหมายมากเท่านั้น) การค้นหาก็จะมุ่งไปเส้นทางที่มีค่าฟังก์ชันฮิวริสติกที่ดีกว่า

3) ฟังก์ชันฮิวริสติกนี้เป็นสิ่งที่ใช้ชี้แนะกระบวนการค้นหาว่าควรจะค้นหาไปในทิศทางใด ซึ่งกระบวนการค้นหาที่ใช้ฟังก์ชันฮิวริสติกสามารถออกแบบได้หลายชนิด ดังจะกล่าวต่อไป

4) ในบางกรณีที่เราสามารถนิยามฟังก์ชันฮิวริสติกได้อย่างสมบูรณ์แบบ การค้นหาก็จะสามารถมุ่งตรงไปยังสถานะเป้าหมายโดยไม่ผิดเส้นทางเลยแต่ถ้าฟังก์ชันฮิวริสติกไม่ดีก็อาจทำให้กระบวนการค้นหาหลงไปในทิศทางที่ผิดได้ ทำให้คำตอบที่ได้เมื่อใช้ฮิวริสติกไม่ใช่คำตอบที่ดีที่สุด

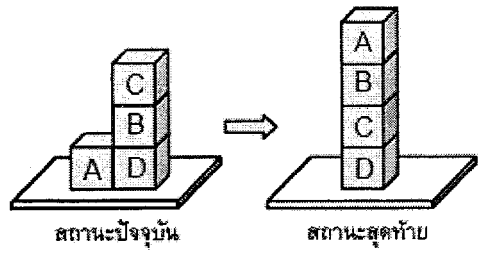
2.4.1 ตัวอย่างของฟังก์ชันฮิวริสติก

ในส่วนนี้จะขอยกตัวอย่างการนิยามฟังก์ชันฮิวริสติกสัก 3 ฟังก์ชันดังนี้

2.4.1.1 ฟังก์ชันฮิวริสติก h1 สำหรับปัญหาโลกของบล็อก

ฟังก์ชันฮิวริสติกตัวแรกที่จะยกมาให้ดูเป็นฟังก์ชันสำหรับคำนวณค่าฮิวริสติก (heuristic value) สำหรับสถานะใดๆ ของปัญหาโลกของบล็อก ขอเรียกฟังก์ชันนี้ว่า h1 ซึ่งนิยามดังนี้

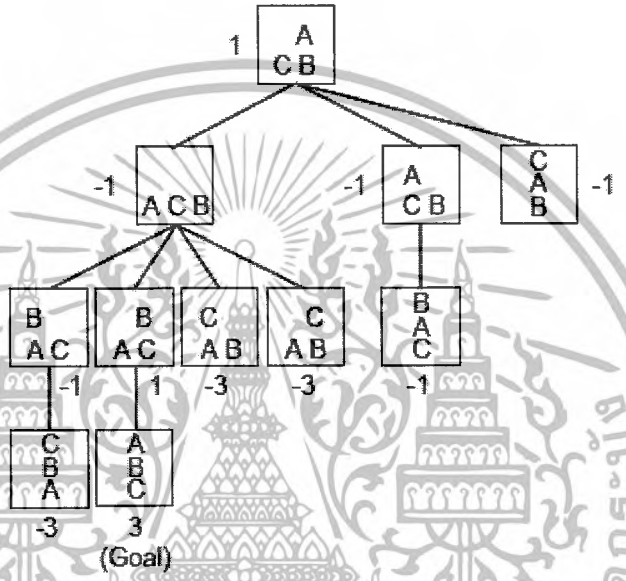
h1: บวกหนึ่งให้กับทุกบล็อกที่วางบนสิ่ง (บล็อกหรือโต๊ะ) ที่มันควรอยู่ และลบหนึ่งถ้าไม่ใช่



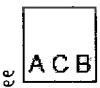
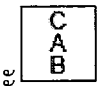
รูป 2.10 ตัวอย่างฟังก์ชันฮิวริสติก h1

'สิ่งที่มันควรอยู่' ในที่นี้คือสถานะสุดท้ายหรือคำตอบ พิจารณารูป 2.10 เราจะได้ว่าค่าฟังก์ชันจะเท่ากับค่าที่บล็อกแต่ละบล็อกได้รับเมื่อเทียบกับสถานะสุดท้ายว่า มันวางอยู่บนสิ่งเดียวกันหรือไม่ ซึ่งจะได้ว่า A ได้ -1 เพราะว่าในสถานะสุดท้าย A วางอยู่บน B แต่ในสถานะปัจจุบัน A วางอยู่บนโต๊ะ เมื่อตรวจสอบบล็อกทุกบล็อกจะได้ดังนี้คือ A = -1, B = -1,C = -1, D = 1 ซึ่งทำให้ได้ค่ารวมของฟังก์ชันเท่ากับ -2 หน่วย

ด้านล่างนี้แสดงค่าฮิวริสติกสำหรับสถานะต่างๆ ในรูป 2.5



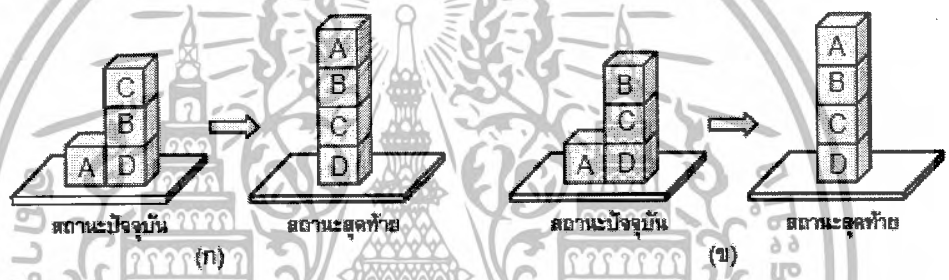
รูป 2.11 ค่าฮิวริสติก h1 สำหรับสถานะต่างๆในรูป 2.5

ในรูป 2.11 นี้ ตัวเลขที่กำกับที่แต่ละสถานะเป็นค่าฮิวริสติกที่คำนวณได้โดยฟังก์ชัน h1 จะเห็นว่าสถานะ  ที่เข้าใกล้คำตอบที่ค่าฮิวริสติกเท่ากับ 1 ส่วนสถานะ  ที่ต่างจากคำตอบมากก็มีค่าฮิวริสติกน้อยสุดเท่ากับ -3 ซึ่งแสดงให้เห็นว่าฟังก์ชัน h1 สามารถวัดความใกล้เคียงคำตอบได้ค่อนข้างดี อย่างไรก็ดีเมื่อพิจารณาที่สถานะลูกของสถานะเริ่มต้นในระดับแรก จะพบว่าสถานะทุกตัวมีค่าเท่ากับ -1 ซึ่งหมายความว่าฟังก์ชัน h1 ไม่สามารถแยกความแตกต่างของสถานะทั้งสามนี้ได้ แม้ว่าสถานะ  จะนำไปสู่คำตอบได้เร็วกว่าสถานะ  และสถานะ 

2.4.1.2 ฟังก์ชันฮิวริสติก h2 สำหรับปัญหาโลกของบล็อก

ดังจะเห็นได้ในตัวอย่างของฟังก์ชันฮิวริสติก h1 ที่กล่าวข้างต้นว่าไม่สามารถแยกความแตกต่างของบางสถานะได้ในที่นี้จึงขอยกตัวอย่างฟังก์ชันฮิวริสติก h2 ที่มีประสิทธิภาพในการวัดความเข้าใกล้คำตอบหรือความดีของสถานะได้อย่างสมบูรณ์แบบและมีประสิทธิภาพดีกว่า h1 ซึ่ง h2 มีนิยามดังนี้

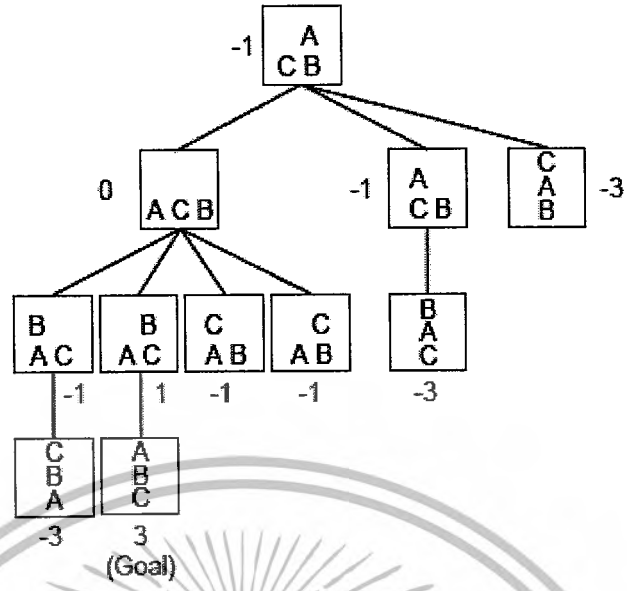
h2: สำหรับบล็อกแต่ละก้อนที่อยู่บนโครงสร้างที่ถูก บวก 1 แต้มให้กับบล็อกทุกก้อนที่อยู่ในโครงสร้างนั้นเพื่อเป็นคะแนนสำหรับบล็อกที่อยู่บนโครงสร้างที่ถูกนั้นและลบ 1 แต้มสำหรับบล็อกทุกก้อนที่อยู่บนโครงสร้างที่ผิด เพื่อเป็นคะแนนสำหรับบล็อกที่อยู่บนโครงสร้างที่ผิดนั้นค่าฮิวริสติกสำหรับสถานะคือคะแนนรวมของบล็อกทุกก้อนที่พิจารณาโดยที่โครงสร้างคือบล็อกที่เรียงตัวต่อกันตามแนวตั้ง (ไม่นับโต๊ะ)



รูป 2.12 ตัวอย่างฟังก์ชันฮิวริสติก h2

ฟังก์ชันฮิวริสติก h2 นี้จะพิจารณาถึงค่าความดีของสถานะโดยดูที่โครงสร้างที่รองรับบล็อกใดๆ ว่าต้องเป็นโครงสร้างที่ตรงกับโครงสร้างที่รองรับบล็อกเดียวกันในคำตอบ ทำให้การวัดค่าฮิวริสติกมีความแม่นยำยิ่งขึ้นพิจารณาตัวอย่างในรูป 2.12 (ก)โครงสร้างที่รองรับบล็อก 'C' ในสถานะปัจจุบันคือโครงสร้าง  ซึ่งต่างจากในคำตอบที่โครงสร้าง  เป็นโครงสร้างที่รองรับบล็อก 'C' ดังนั้นสำหรับบล็อก 'C'ที่สถานะปัจจุบันที่มีบล็อก 2 ก่อนอยู่ในโครงสร้างรองรับที่ผิดจึงได้แต้มเท่ากับ -2 เมื่อคำนวณคะแนนสำหรับบล็อกทุกก่อนก็จะได้คะแนนตามนี้คือ A = 0, B = -1, C = -2, D = 0 ซึ่งทำให้ได้ค่ารวมของฟังก์ชันเท่ากับ -3หน่วย (บล็อก A ได้ศูนย์แต้มเพราะว่าไม่มีโครงสร้างที่รองรับมันอยู่เลย เนื่องจากโครงสร้างไม่รวมถึงโต๊ะ) ในกรณีของสถานะปัจจุบันในรูป 2.10 (ข) จะได้คะแนนตามนี้คือ A = 0,B = 2, C = 1, D = 0 ซึ่งทำให้ได้ค่ารวมของฟังก์ชันเท่ากับ 3 หน่วย

เมื่อนำฟังก์ชันฮิวริสติก h2 ไปวัดค่าฮิวริสติกให้กับสถานะต่างๆในรูป 2.5 จะได้ผลดังรูป 2.13 ดังต่อไปนี้



รูป 2.13 ค่าฮิวริสติก h2 สำหรับสถานะต่างๆในรูป 2.5

จะเห็นว่าฟังก์ชันฮิวริสติก h2 สามารถแยกความแตกต่างระหว่างสถานะลูกทุกตัวของสถานะเริ่มต้นได้อย่างถูกต้องนอกจากนั้นฟังก์ชันนี้ยังบอกความใกล้กับคำตอบของสถานะทุกตัวได้อย่างสมบูรณ์แบบเมื่อดูเส้นทางจากสถานะเริ่มต้นจนถึงคำตอบ จะพบว่าในแต่ละระดับสถานะที่มีค่าฮิวริสติกสูงกว่าสถานะอื่นๆ ในระดับเดียวกันล้วนเป็นสถานะที่นำไปสู่คำตอบได้โดยตรงทั้งสิ้น

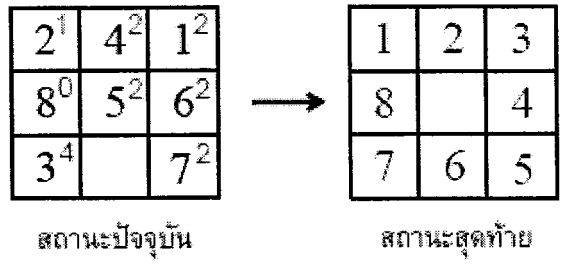
2.4.1.3 ฟังก์ชันฮิวริสติกสำหรับปัญหา 8-Puzzle

ตัวอย่างของฟังก์ชันฮิวริสติกสำหรับปัญหา 8-Puzzle แสดงในสมการด้านล่างนี้

$$h_{Man} = \sum_{i=1}^{8} d_x(c_i, g_i) + \sum_{i=1}^{8} d_y(c_i, g_i) \qquad (2.1)$$

โดยที่ c_i, g_i, d_x, d_yคือพิกัดของแผ่นป้าย i ที่สถานะปัจจุบัน พิกัดของแผ่นป้าย i ที่สถานะเป้าหมาย ระยะห่างระหว่าง c_iกับ g_iตามแกน x และระยะห่างระหว่าง c_iกับ g_iตามแกน y ตามลำดับ

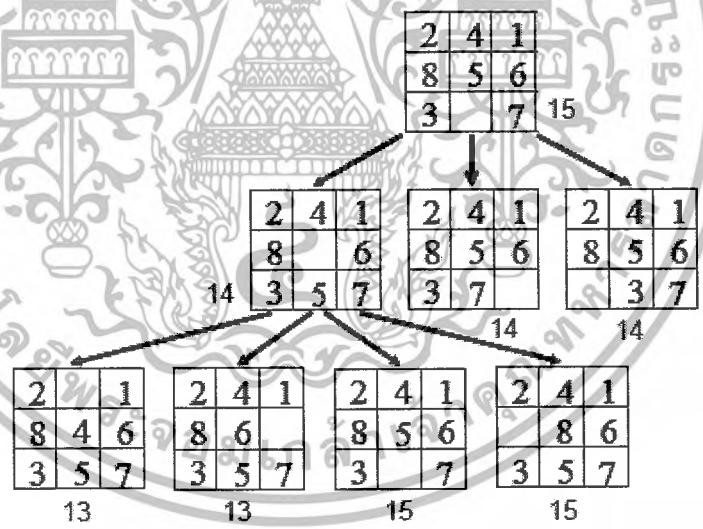
ฟังก์ชันนี้เรียกว่าฟังก์ชันแมนฮัตตันฟังก์ชันนี้สามารถแปรความหมายอย่างง่าย คือ การหาค่าจำนวนครั้งที่ต้องขยับแผ่นป้ายตั้งแต่แผ่นที่ 1 ถึงแผ่นที่ 8 จากตำแหน่งปัจจุบันตามแนวแกน x และ y ไปยังตำแหน่งที่ควรจะอยู่ตามคำตอบ ว่าต้องขยับทั้งสิ้นน้อยที่สุดรวมกี่ครั้ง โดยไม่ต้องมีเงื่อนไขว่าจะติดกับช่องว่างหรือไม่ (สามารถเลื่อนได้เลย แม้จะมีแผ่นป้ายอื่นวางอยู่แล้ว) สังเกตว่าฟังก์ชันนี้ยิ่งมีค่าน้อยยิ่งดี



รูป 2.14 ฟังก์ชันแมนฮัตตันสำหรับปัญหา 8-Puzzle

ตัวอย่างเช่นพิจารณารูป 2.14 ตัวเลขที่อยู่มุมขวาบนในแผ่นป้ายแต่ละแผ่นแสดงจำนวนครั้งที่ต้องเลื่อนแผ่นป้ายนั้นไปยังตำแหน่งที่มันควรอยู่เมื่อเทียบกับคำตอบ เช่นแผ่นป้าย '2' ที่สถานะปัจจุบันต้องเลื่อนทั้งหมด 1 ครั้ง (ตามแนวแกน x) จึงจะไปอยู่ในตำแหน่งเดียวกับในคำตอบแผ่นป้าย '3' ที่สถานะปัจจุบันต้องเลื่อนทั้งหมด 2+2 ครั้ง (2 ครั้งตามแนวแกน x และ 2 ครั้งตามแนวแกน y) จึงจะไปอยู่ในตำแหน่งเดียวกับในคำตอบ ดังนั้นค่าฮิวริสติกของสถานะปัจจุบันนี้มีค่าเท่ากับ 2+1+4+2+2+2+2+0 = 15 หน่วย

เมื่อนำฟังก์ชัน h_{Man} ไปวัดค่าฮิวริสติกให้กับสถานะต่างๆในรูป 2.2 จะได้ผลดังรูป 2.15 ต่อไปนี้



รูป 2.15 ค่าฟังก์ชันแมนฮัตตันสำหรับสถานะต่างๆในรูป 2.2

จากตัวอย่างของการนิยามฟังก์ชันฮิวริสติกทั้งสามด้านบน คงจะเห็นแนวทางการนิยามฟังก์ชันฮิวริสติกเหล่านี้ ซึ่งโดยมากเรามักนิยามฟังก์ชันเพื่อคำนวณความคล้าย (ความต่าง)กับคำตอบ และวัดเป็นตัวเลขไปใช้ในการค้นหาต่อไป ฟังก์ชันฮิวริสติกที่ดีต้องคำนวณง่ายไม่ยุ่งยากซับซ้อนมากจนกระทั่งเสียเวลาคำนวณมหาศาล ข้อสังเกตอีกอย่างก็คือการสร้างฟังก์ชันฮิวริสติกอาจทำได้โดยตัดเงื่อนไขของการใช้ตัวกระทำการออก อย่างเช่นในตัวอย่างของ 8-Puzzle นั้นตัวกระทำการเพื่อเลื่อนแผ่นป้ายมีเงื่อนไขว่า แผ่นป้ายจะเลื่อนได้ก็ต่อเมื่อแผ่นป้ายนั้นติดกับ

ช่องว่างการนิยามฟังก์ชันแมนฮัตตันนั้นเสมือนกับว่าแผ่นป้ายเลื่อนไปยังตำแหน่งต่างๆ ได้โดยไม่ต้องติดเงื่อนไขว่า แผ่นป้ายนั้นอยู่ติดกับช่องว่างหรือไม่แล้วนับการเลื่อนแผ่นป้ายทั้งหมดเป็นการนิยามฟังก์ชัน เป็นต้น

2.4.2 อัลกอริทึมปีนเขา

ฟังก์ชันฮิวริสติกที่นิยามขึ้นในข้างต้นนั้น สามารถนำมาช่วยกระบวนการค้นหาเพื่อให้ได้คำตอบอย่างรวดเร็วและมีประสิทธิภาพ วิธีการที่จะนำฟังก์ชันฮิวริสติกมาใช้มีหลายวิธีด้วยกันขึ้นอยู่กับว่าจะใช้ในลักษณะใด เช่นเลือกสถานะที่มีค่าฮิวริสติกดีขึ้น แล้วเดินไปยังสถานะนั้นเลยโดยไม่ต้องสนใจสถานะที่มีค่าฮิวริสติกแย่กว่าสถานะปัจจุบัน หรือว่าจะเก็บสถานะทุกตัวไว้แม้ว่าค่าฮิวริสติกจะแย่ลง แล้วพิจารณาสถานะเหล่านี้ทีหลัง เป็นต้น ในส่วนต่อไปนี้จะกล่าวถึงอัลกอริทึมต่างๆ ที่นำฟังก์ชันฮิวริสติกมาช่วยในการค้นหาคำตอบ โดยเริ่มจากอัลกอริทึมปีนเขา (hill-climbing algorithm)

รูป 2.16 อัลกอริทึมปีนเขา

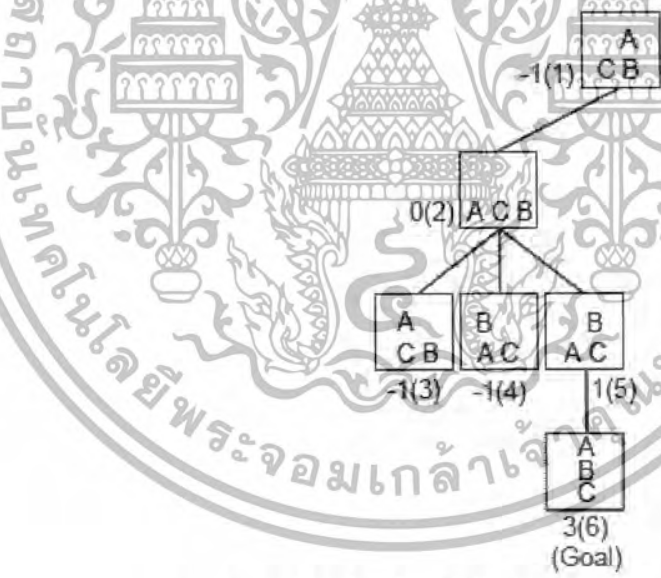
การค้นหาแบบนี้เปรียบเสมือนการปีนไปสู่ยอดเขาดังแสดงในรูป 2.16 ซึ่งอัลกอริทึมจะขึ้นในแนวดิ่งตลอด ถ้าเจอทางแยกเราก็จะไปแยกที่ตรงดิ่งขึ้นไปจนกระทั่งถึงยอดเขา ความสูงจากฐานภูเขาจนถึงตำแหน่งที่อยู่ในปัจจุบันก็จะเปรียบเสมือนค่าฮิวริสติก (ในกรณีนี้เราต้องการหาค่าสูงสุด) ซึ่งในที่นี้ยิ่งมากยิ่งดี อัลกอริทึมแสดงในรูป 2.17

```
Algorithm: Simple Hill-Climbing Search

1. Evaluate the initial state.
2. IF the initial state=goal state THEN
       return the initial state and quit
   ELSE current state := initial state.
3. UNTIL a goal state is found or there are no new
   operators left to be applied in the current state DO
     3.1 Select an operator that has not yet been
         applied to the current state and apply it to
         produce a new state.
     3.2 Evaluate the new state.
         IF  new state=goal state THEN
             return the new state and quit
         ELSE IF the new state is better than the
                 current state THEN
                 current state := new state
         ELSE IF the new state is not better than the
                 current state THEN
                 continue in this loop.
```

รูป 2.17 อัลกอริทึมปีนเขาอย่างง่าย

ตัวอย่างการใช้ฟังก์ชันฮิวริสติก h2 โดยอัลกอริทึมปีนเขาอย่างง่ายในรูป 2.17 กับปัญหาโลกของบล็อกแสดงในรูป 2.18



รูป 2.18 ตัวอย่างอัลกอริทึมปีนเขากับปัญหาโลกของบล็อก

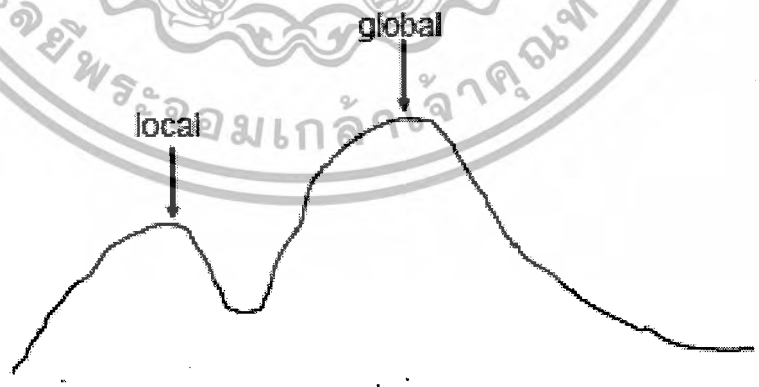
ตัวเลข h(i) ในรูปแสดงว่า สถานะที่ i มีค่าฮิวริสติกเท่ากับ h จากรูปจะเห็นได้ว่าเริ่มต้นจากสถานะที่ 1 ที่มีค่าฮิวริสติกเท่ากับ -1 อัลกอริทึมปีนเขาใช้ตัวกระทำการเพื่อสร้างสถานะลูกตัวแรกของสถานะที่ 1 แล้ววัดค่าฮิวริสติกได้ 0 ซึ่งมีค่าดีขึ้นถ้าสังเกตจากรูป 2.5 จะพบว่าสถานะที่ 1 มีสถานะลูกทั้งหมด 3 ตัว แต่ในกรณีของอัลกอริทึมปีนเขานี้เมื่อได้สถานะลูกตัวแรกซึ่งมีค่าฮิวริสติกดีขึ้น อัลกอริทึมจะไม่สร้างสถานะลูกที่เหลืออีก 2 ตัว และจะไม่มีการย้อนกลับมาที่สถานะ

ลูกทั้งสองนี้ แม้ว่าหลังจากนี้อัลกอริทึมจะค้นไม่พบคำตอบกล่าวคือเป็นการตัดทางเลือกทิ้งไปเลย ซึ่งการทำเช่นนี้แม้ว่าจะมีโอกาสไม่พบคำตอบ แต่ก็มีข้อดีที่เป็นการช่วยลดเวลาและปริภูมิที่ทำการค้นหาจะลดลงอย่างมาก

จากนั้นอัลกอริทึมมาที่สถานะที่ 2 แล้วเริ่มสร้างสถานะลูก ได้สถานะที่ 3 ที่มีค่าฮิวริสติก -1 ซึ่งแย่ลงในกรณีที่แย่ลงเช่นนี้ อัลกอริทึมจะไม่ไปยังสถานะลูกตัวนี้ และสร้างสถานะลูกตัวต่อไป โดยใช้ตัวกระทำการที่เหลือ ได้สถานะที่ 4 มีค่าฮิวริสติกเท่ากับ -1 ไม่ดีขึ้นเช่นกัน จึงสร้างสถานะลูกตัวถัดไป เป็นสถานะที่ 5 มีค่าฮิวริสติกเท่ากับ 1 เป็นค่าที่ดีขึ้นอัลกอริทึมจะมายังสถานะนี้และค้นพบคำตอบในที่สุด

อัลกอริทึมปีนเขานี้จะมีประสิทธิภาพมากดังเช่นแสดงในตัวอย่างนี้ซึ่งกระจายสถานะทั้งสิ้นเพียง 6 ตัวแล้วพบคำตอบ เปรียบเทียบกับอัลกอริทึมการค้นหาแนวกว้างก่อนซึ่งใช้สถานะทั้งสิ้นถึง 11 ตัว อย่างไรก็ดีอัลกอริทึมนี้จะมีประสิทธิภาพมาก ถ้าใช้ฟังก์ชันฮิวริสติกที่ดีมากๆ ดังเช่นฟังก์ชัน h2 ซึ่งเป็นฟังก์ชันที่สมบูรณ์มากในกรณีที่ฟังก์ชันฮิวริสติกไม่ดีนักอัลกอริทึมนี้ก็อาจหลงเส้นทางได้ และอาจไม่พบคำตอบแม้ว่าปริภูมิที่กำลังค้นหามีคำตอบอยู่ด้วยก็ตามสาเหตุของการหลงเส้นทางประการหนึ่งมาจากการเลือกสถานะลูก ซึ่งอัลกอริทึมจะไม่ได้พิจารณาสถานะลูกทุกตัว โดยเมื่อพบสถานะลูกตัวใดตัวหนึ่งที่ดีขึ้นก็จะเลือกเส้นทางนั้นในทันที อัลกอริทึมนี้สามารถดัดแปลงเล็กน้อยให้พิจารณาสถานะลูกทุกตัวให้ครบก่อน แล้วเลือกสถานะลูกตัวที่มีค่าฮิวริสติกสูงสุด เมื่อทำเช่นนี้ก็จะทำให้อัลกอริทึมได้พิจารณาเส้นทางที่ดีที่สุด ณ ขณะหนึ่งๆ ได้ดีขึ้นเราเรียกอัลกอริทึมที่ดัดแปลงนี้ว่าอัลกอริทึมปีนเขาชันสุด (steepest ascent hill-climbing)

2.4.3 อัลกอริทึมอบเหนียวจำลอง



รูป 2.19 ตัวอย่างปัญหาที่เกิดกับอัลกอริทึมปีนเขา

พิจารณาตัวอย่างในรูป 2.19 ซึ่งแสดงค่าฮิวริสติกด้วยความสูงจากฐานที่เกิดขึ้นระหว่างการค้นหาด้วยอัลกอริทึมปีนเขา ปัญหาที่เกิดขึ้นก็คืออัลกอริทึมปีนเขาจะค้นพบสถานะที่มีค่าดีสุดเฉพาะที่ (local optimum) เท่านั้น ไม่สามารถค้นพบค่าดีสุดวงกว้าง (global optimum) ได้

เนื่องจากเมื่อการค้นหามาตกที่สถานะดีสุดเฉพาะที่แล้ว พบว่าสถานะใหม่ที่สร้างขึ้นจะมีค่าฮิวริสติกแย่ลงทั้งหมด ทำให้การค้นหาหยุดที่สถานะนั้น ปริภูมิสถานะจำนวนมากที่มีลักษณะของค่าฮิวริสติกดังรูปด้านบนนี้ และในปัญหาจำนวนมาก เรามักพบปริภูมิสถานะที่มีสถานะดีสุดเฉพาะที่มากกว่าหนึ่งสถานะ ทำให้การค้นหาประสบความลำบากในการคนให้พบสถานะดีสุดวงกว้าง

อัลกอริทึมการอบเหนียวจำลอง (simulated annealing algorithm) ถูกออกแบบมาเพื่อให้สามารถหลุดออกได้จากสถานะดีสุดเฉพาะที่ โดยใช้แนวคิดอุณหพลศาสตร์ของกระบวนการอบเหนียว ซึ่งเป็นขั้นตอนการลดอุณหภูมิลงอย่างช้าๆ ระหว่างการหลอมเพื่อให้ได้โลหะที่อยู่ในสภาวะที่เหมาะที่สุด เป็นโลหะเหนียว ไม่เปราะแนวคิดอธิบายให้เขาใจได้ดีโดยใช้รูป 2.20

รูป 2.20 แนวคิดอัลกอริทึมการอบเหนียวจำลอง

เนื่องจากแนวคิดนี้เป็นการหาค่าต่ำสุด จึงใช้รูปที่กลับหัวกลับหางกับรูป 2.19 ปัญหาของค่าดีสุดเฉพาะที่ (ในรูปนี้คือค่าต่ำสุดเฉพาะที่) ก็คือเมื่อการค้นหาแบบปีนเขา (ในกรณีนี้คือกาค้นหาแบบลงเหว) มาตกที่สถานะนี้ การค้นหาก็จะหยุด วิธีที่จะแก้ปัญหาการตกที่สถานะนี้ได้ก็คือต้องดึงการค้นหาให้ขึ้นมาให้ได้เพื่อจะได้คนต่อไปจนพบค่าดีสุดวงกว้าง

อัลกอริทึมการอบเหนียวจำลองจะยอมให้การค้นหาวิ่งไปในทิศทางที่ไม่ดีได้ในช่วงเริ่มต้นของกระบวนการค้นหา เพื่อเป็นการสำรวจทั่วๆ แบบหยาบก่อน แล้วจึงค่อยๆ ค้นหาอย่างละเอียดเมื่อเวลาผ่านไปด้วยแนวคิดเช่นนี้ทำให้เราคาดหวังว่า ผลลัพธ์สุดท้ายที่ได้จะไม่ขึ้นกับสถานะเริ่มต้นมากนักเพื่อให้เขาใจถึงอัลกอริทึมนี้ ขออธิบายการอบเหนียวของโลหะ โดยย่อดังนี้

การอบเหนียวของโลหะเป็นการหลอมโลหะจนละลาย (ทำให้โลหะอยู่ในสถานะที่มีพลังงานสูง) แล้วค่อยๆ ลดอุณหภูมิลงทีละน้อยจนโลหะเปลี่ยนกลับมาอยู่ในสถานะของแข็ง จุดมุ่งหมายคือพยายามทำให้โลหะกลับมาเป็นของแข็งในสถานะสุดท้ายที่มีพลังงานต่ำสุดซึ่งโดยทั่วไปตามธรรมชาติ สสารจะพยายามเปลี่ยนตัวเองจากสถานะที่มีพลังงานสูงไปสู่สถานะพลังงานต่ำ แต่ก็มีความน่าจะเป็นที่สสารเปลี่ยนจากพลังงานต่ำไปพลังงานสูงอยู่บางความน่าจะเป็นนี้สามารถคำนวณได้โดยสมการต่อไปนี้

$$p=e^{-\Delta_{E/kT}} \quad (2.2)$$

โดยที่ ΔE เป็นระดับพลังงานที่เปลี่ยนไป (เป็นค่าบวก) T เป็นอุณหภูมิ และ k เป็นค่าคงที่ของโบลตซมันน (Boltzmann) การเปลี่ยนแปลงจากระดับพลังงานสูงไปต่ำนั้น มีความน่าจะเป็นที่จะเกิดในช่วงเริ่มต้นมากกว่าในช่วงปลายของกระบวนการอบเหนียวอัตราการลดอุณหภูมิในการอบเหนียวเรียกว่า หมายกำหนดการอบเหนียว (annealing schedule) ถ้าหมายกำหนดการอบเหนียวเร็ว กล่าวคือลดอุณหภูมิลงอย่างรวดเร็ว โลหะก็มีโอกาสเขาสู่สถานะสุดท้ายที่มีพลังงานสูงอยู่ เราจึงต้องพิจารณาหมายกำหนดการอบเหนียวไม่ให้เร็วเกินไป แต่ถ้าช้าไปก็ทำให้เสียเวลาโดยไม่จำเป็นเช่นกัน

แนวคิดของการอบเหนียวจำลองก็ได้จากการลอกเลียนการอบเหนียวของโลหะ โดยเป็นการค้นหาแบบปีนเขา (ลงเหว) แบบหนึ่ง ซึ่งการค้นหาสามารถไปในทิศทางที่ไม่ดีได้ โดยเฉพาะในช่วงต้นของกระบวนการค้นหา เพื่อสำรวจบริเวณที่น่าจะนำไปสู่คำตอบดีสุดเราได้สมการความน่าจะเป็นสำหรับการอบเหนียวจำลองดังนี้

$$p=e^{-\Delta_{E/T}} \quad (2.3)$$

โดยที่ ΔE เป็นค่าฮิวริสติกที่เปลี่ยนไป (เป็นค่าบวก) T เป็นอุณหภูมิ เนื่องจาก k ในสมการ (2.2) เป็นค่าคงที่ จึงรวมเข้าไปใน T ได้

สมการนี้เมื่อนำไปใช้รวมกับการค้นหาแบบปีนเขา ก็จะช่วยให้กระบวนการค้นหาสามารถหลุดออกจากค่าดีสุดเฉพาะที่ได้ตรงกับความต้องการของเรา ตัวอย่างเช่นเมื่อเราอยู่ที่สถานะปัจจุบันมีค่าฮิวริสติกเท่ากับ A และเมื่อสร้างสถานะลูกที่มีค่าฮิวริสติกเท่ากับ Bซึ่งแย่ลง การค้นหาอาจไปยังสถานะลูกนี้ได้ โดยคำนวณค่าความน่าจะเป็น (p) ตามสมการด้านบน ได้เป็น $p=e^{-|A-B|/T}$ ค่าที่ได้นี้จะอยู่ระหว่าง 0 ถึง 1 จากนั้นเราจะสุ่มตัวเลข (random(0,1)) ขึ้นมาหนึ่งตัว ถ้าค่า p มีค่ามากกว่า ก็จะรับสถานะลูกเป็นสถานะต่อไปได้ ค่า T เป็นพารามิเตอร์ของอัลกอริทึมนี้ที่เราสามารถปรับแต่งให้เหมาะสมสำหรับปัญหาหนึ่งๆ ที่เราสนใจ โดยช่วงเริ่มต้นกำหนดให้เป็นค่ามากแล้วค่อยๆ ลดลงเมื่อการค้นหาดำเนินต่อไป

เมื่อพิจารณาสมการนี้ เราจะได้คุณสมบัติที่เราต้องการดังนี้คือ เมื่อ T มาก (ในช่วงต้นของกระบวนการค้นหา) จะได้ p มีค่ามาก คือเรายอมให้การค้นหาไปในทิศทางที่ไม่ดีได้ง่ายหน่อยในช่วงต้นการค้นหา แต่เมื่อ T น้อย (ในช่วงปลายของกระบวนการค้นหา) จะได้ p มีค่าน้อยคือการค้นหาเริ่มเข้าสู่คำตอบ ก็ไม่ควรให้การค้นหากระโดดไปยังสถานะที่แย่ลงเมื่อพิจารณาΔE จะพบว่าเมื่อ ΔE มีค่ามากจะได้ว่า p มีค่าน้อย กล่าวคือการก้าวกระโดดจากสถานะที่ดีไปยังสถานะที่ไม่ดีที่การก้าวกระโดดนั้นเป็นก้าวใหญ่ (มีการเปลี่ยนแปลงค่าฮิวริสติกมาก) จะเกิดขึ้นได้ยากกว่าการก้าว

กระโดดที่เป็นก้าวเล็ก (มีการเปลี่ยนแปลงค่าฮิวริสติกน้อย) ซึ่งเป็นคุณสมบัติที่น่าพอใจเนื่องจากเราเชื่อว่าสถานะที่เป็นคำตอบมักจะเป็นหลุมลึกๆ ถ้าจะหลุดออกจากหลุมลึกได้ต้องก้าวใหญ่ ซึ่งไม่ควรให้เกิดขึ้นได้ง่าย ส่วนสถานะดีสุดเฉพาะที่มักเป็นหลุมตื้นๆ ดังนั้นจึงให้โอกาสหลุดลอดได้ง่ายหน่อย อัลกอริทึมการอบเหนียวจำลองแสดงในรูป 2.21

อัลกอริทึมจะมีลักษณะคล้ายกับอัลกอริทึมปีนเขา แต่สามารถเลือกสถานะที่มีค่าฮิวริสติกแย่ลงได้ดังนั้นเมื่อสิ้นสุดกระบวนการค้นหา สถานะตัวสุดท้ายที่พิจารณาอยู่อาจไม่ใช่สถานะที่มีค่าฮิวริสติกดีสุด ดังนั้นเราจึงจำเป็นต้องจำสถานะที่มีค่าฮิวริสติกดีสุดที่ผ่านมาไว้ โดยเก็บค่าไวในตัวแปรชื่อ BEST-SO-FAR ในตอนเริ่มกระบวนการค้นหา จะกำหนดค่าคงที่ตัวหนึ่งเป็นอุณหภูมิการอบเหนียว (T) และลดอุณหภูมิลงตามความเหมาะสม (ในขั้นตอนที่ 5.3 ในอัลกอริทึม) ในกรณีที่เราพิจารณาสถานะใหม่ (new state) ตัวหนึ่งอยู่ ถ้าพบว่าสถานะนี้มีค่าฮิวริสติกดีขึ้น ก็จะค้นหาต่อไปยังสถานะใหม่นี้ และปรับค่าตัวแปร BEST-SO-FAR แต่ถ้าหากว่าค่าฮิวริสติกแย่ลง เราจะคำนวณค่าความน่าจะเป็น ($e^{-\Delta E/T}$) ที่จะไปยังสถานะใหม่นี้ การประยุกต์ใช้อัลกอริทึมนี้ ผู้ใช้จำเป็นต้องกำหนดพารามิเตอร์ในอัลกอริทึมให้เหมาะสมสำหรับปัญหาที่กำลังพิจารณาอยู่ด้วยพารามิเตอร์นี้ได้แก่ ค่าอุณหภูมิเริ่มต้นและปริมาณการลดอุณหภูมิ

```
Algorithm: Simulated Annealing Search
1. Evaluate the initial state.
2. IF initial state=goal state THEN
     return the initial state and quit
   ELSE current state := initial state.
3. BEST-SO-FAR := current state
4. T := constant
5. UNTIL a goal state is found or there are no new
   operators left to be applied in the current state DO
   5.1 Select an operator that has not yet been applied
       to the current state and apply it to produce a
       new state.
   5.2 Evaluate the new state.
       IF new state=goal state THEN
          return new state and quit
       ELSE IF the new state is better than the current
                state THEN {
                current state := new state
                IF the new state is better than BEST-SO-FAR
                THEN BEST-SO-FAR := new state }
       ELSE IF the new state is not better than the
                current state THEN {
                ΔE := |(value of the current state)-
                      (value of the new state)|
                IF e^(-ΔE/T) > random(0,1) THEN
                   current state := new state }
   5.3 Revise T as necessary.
6. Return BEST-SO-FAR as the answer.
```

รูป 2.21 อัลกอริทึมการอบเหนียวจำลอง

2.4.4 อัลกอริทึมดีสุดก่อน

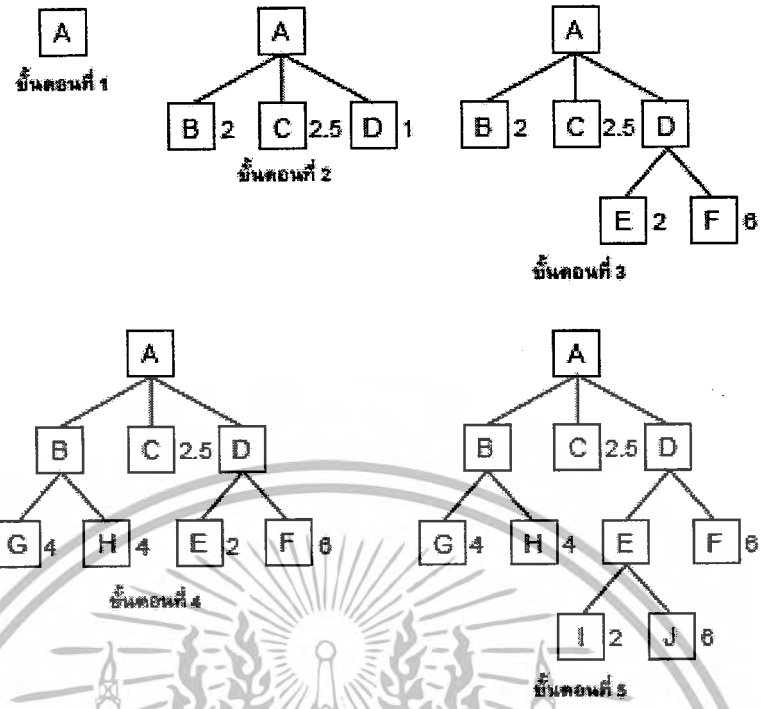
อัลกอริทึมดีสุดก่อน (best-first search) จะเก็บสถานะทุกตัวโดยไม่มีการตัดทิ้งไป ต่างจากอัลกอริทึมปีนเขาที่เมื่อเลือกเส้นทางหนึ่งแล้วตัวเลือกอื่นที่เป็นลูกของสถานะปัจจุบันจะถูกตัดทิ้งไปการไม่ตัดทิ้งจึงทำให้อัลกอริทึมนี้ไม่พลาดเส้นทางที่นำไปสู่คำตอบโดยในแต่ละขั้นตอนจะเลือกสถานะที่มีค่าฮิวริสติกดีสุด โดยพิจารณาสถานะทุกตัวที่ยังไม่ถูกกระจาย(สถานะที่ยังไม่ได้สร้างสถานะลูก) อัลกอริทึมแสดงในรูป 2.22

```
Algorithm: Best-First Search

1. OPEN := {initial state}
2. UNTIL a goal state is found or there are no states
   left on OPEN DO
   2.1 Pick the best node on OPEN.
   2.2 Generate its successors.
   2.3 FOR EACH successor DO
         IF the successor has not been generated THEN
           evaluate it, add it to OPEN, and record it
           parent.
         IF the successor has been generated before
         THEN change the parent if this new path is
              better than the previous one.
```

รูป 2.22 อัลกอริทึมดีสุดก่อน

ตัวแปร OPEN ในอัลกอริทึมเก็บสถานะทุกตัวที่ถูกสร้างขึ้นแล้วแต่ยังไม่ถูกกระจาย (ยังไม่ได้สร้างสถานะลูกของมัน) ขั้นตอนสุดท้าย (IF STATEMENT) มีไว้เพื่อปรับเส้นทางและต้นทุนใหม่ เนื่องจากว่าสถานะหนึ่งๆ อาจเขาถึงจากหลายเส้นทาง (ดังที่ได้กล่าวข้างต้นแล้วว่าปริภูมิสถานะโดยทั่วไปเป็นกราฟ) ในกรณีที่เราพบเส้นทางใหม่ที่นำมาสู่สถานะที่เคยสร้างแล้วและเส้นทางใหม่ดีกว่าหรือมีต้นทุนหรือจำนวนครั้งจากสถานะเริ่มต้นมายังสถานะนี้น้อยกว่าเส้นทางเดิม ก็ให้แก้ไขเส้นทางให้ถูกต้อง ตัวอย่างของการค้นหาแบบอัลกอริทึมดีสุดก่อนแสดงในรูป 2.23



รูป 2.23 ตัวอย่างของการค้นหาแบบอัลกอริทึมดีสุดก่อน

สมมติว่า 'A' เป็นสถานะเริ่มต้น สถานะลูกทุกตัวของ 'A' (คือ 'B', 'C' และ 'D') ถูกสร้างขึ้นในขั้นตอนที่ 2 จากนั้นเมื่อวัดค่าฮิวริสติกของสถานะทุกตัวได้ว่า 'D' มีค่าดีสุด (ในที่นี้ยิ่งน้อยยิ่งดี) จึงนำ 'D' มากระจายสถานะลูก ได้สถานะ 'E' และ 'F' ซึ่งมีค่าฮิวริสติก 2 และ 6 ตามลำดับ เมื่อเปรียบเทียบค่าฮิวริสติกของสถานะทุกตัวที่ยังไม่ได้กระจาย (เก็บไว้ในOPEN) จะได้ว่า 'B' มีค่าดีสุด (เท่ากับ 'E' แต่ในที่นี้กำหนดให้กรณีค่าเท่ากันให้นำสถานะที่สร้างก่อนมาทำก่อน) จึงนำ 'B' มากระจายต่อ และต่อจากนั้นในขั้นตอนที่ 5 สถานะ 'E' ถูกกระจายต่อไปตามลำดับ เป็นเช่นนี้จนกระทั่งพบคำตอบหรือไม่สามารถสร้างสถานะใหม่ได้อีก

2.4.5 อัลกอริทึม A*

อัลกอริทึม A* (A* Search) เป็นการขยายอัลกอริทึมดีสุดก่อนโดยพิจารณาเพิ่มเติมถึงต้นทุนจากสถานะเริ่มต้นมายังสถานะปัจจุบันเพื่อใช้คำนวณค่าฮิวริสติกด้วย ในกรณีของอัลกอริทึม A* เราต้องการหาค่าต่ำสุดของฟังก์ชัน f(s) ของสถานะ s นิยามดังนี้

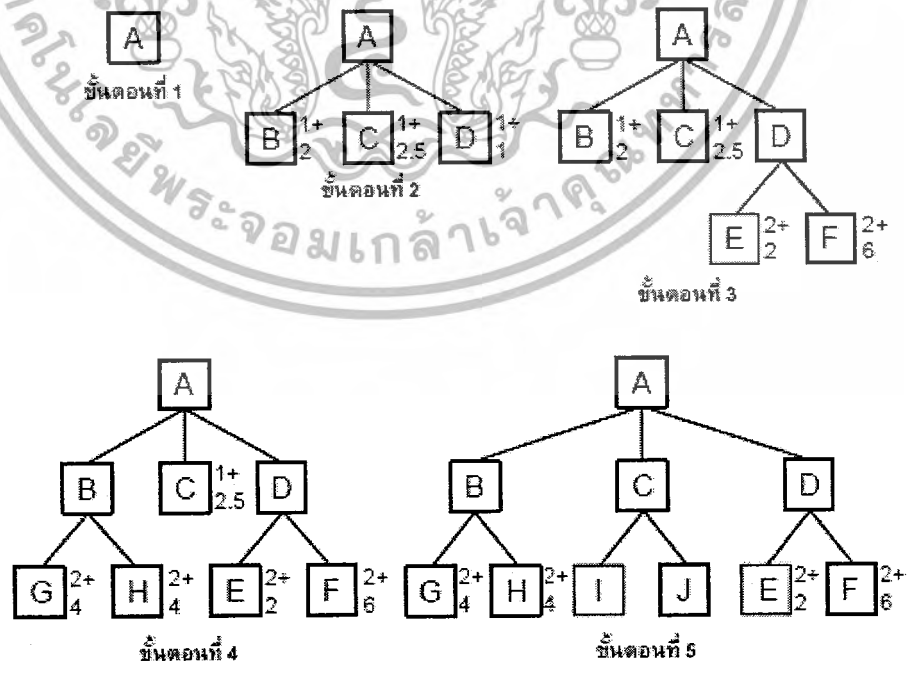
$$f'(s) = g(s) + h'(s) \qquad (2.4)$$

โดยที่ g คือฟังก์ชันที่คำนวณต้นทุนจากสถานะเริ่มต้นมายังสถานะปัจจุบัน h' คือฟังก์ชันที่ประมาณต้นทุนจากสถานะปัจจุบันไปยังคำตอบ ดังนั้น f' จึงเป็นฟังก์ชันที่ประมาณต้นทุนจากสถานะเริ่มต้นไปยังคำตอบ (ยิ่งน้อยยิ่งดี)

เรามองได้ว่าฟังก์ชัน h' คือฟังก์ชันฮิวริสติกที่เราเคยใช้ในการค้นหาอื่นๆ ก่อนหน้านี้ เช่นอัลกอริทึมปีนเขา อัลกอริทึมดีสุดก่อน เป็นต้น ในที่นี้เราใส่เครื่องหมาย ' เพื่อแสดงว่าฟังก์ชันนี้เป็นฟังก์ชันประมาณของฟังก์ชันจริงที่ไม่รู้ (เราทำได้แค่ประมาณว่า h' คือต้นทุนจากสถานะปัจจุบันไปยังคำตอบ เราจะรู้ต้นทุนจริงก็ต่อเมื่อเราได้ทำการค้นหาจริงจนไปถึงคำตอบแล้ว) ส่วน g เป็นฟังก์ชันที่คำนวณต้นทุนจริงจากสถานะเริ่มต้นมายังสถานะปัจจุบัน (จึงไม่ได้ใส่เครื่องหมาย ') เพราะเราสามารถหาต้นทุนจริงได้เนื่องจากได้ค้นหาจากสถานะเริ่มต้นจนมาถึงสถานะปัจจุบันแล้ว ส่วน f' ก็เป็นเพียงแค่ฟังก์ชันประมาณโดยการรวมต้นทุนทั้งสองคือ h' กับ g

อัลกอริทึม A* จะทำการค้นหาโดยวิธีเดียวกันกับอัลกอริทึมดีสุดก่อนทุกประการ ยกเว้นฟังก์ชันฮิวริสติกที่ใช้เปลี่ยนมาเป็น f' (ต่างจากอัลกอริทึมดีสุดก่อนที่ใช้ h') และด้วยการใช้ f' อัลกอริทึม A* จึงให้ความสำคัญกับสถานะหนึ่งๆ 2 ประการคือ (1) สถานะที่ดีต้องมี h' ดีคือต้นทุนเพื่อจะนำไปสู่คำตอบหลังจากนี้ต้องน้อย และ (2) ต้นทุนที่จ่ายไปแล้วกว่าจะถึงสถานะนี้ (g) ต้องน้อยด้วย เราจึงได้ว่า A* จะค้นหาเส้นทางที่ให้ต้นทุนโดยรวมน้อยสุดตามค่า f' ซึ่งต่างจากอัลกอริทึมดีสุดก่อนที่เน้นความสำคัญของสถานะที่ต้นทุนหลังจากนี้ที่จะนำไปสู่คำตอบต้องน้อย โดยไม่สนใจว่าต้นทุนที่จ่ายไปแล้วกว่าจะนำมาถึงสถานะนี้ต้องเสียไปเท่าไร

รูป 2.24 แสดงการค้นหาด้วยอัลกอริทึม A* กับสถานะในรูป 2.23 โดยสมมติให้ต้นทุนหรือระยะห่างระหว่างสถานะพ่อแม่ไปยังสถานะลูกเท่ากับ 1 หน่วย เช่นต้นทุนจริง (g) จาก 'A' ไปยัง 'B', 'C' หรือ 'D' มีค่าเท่ากับ 1 หน่วย



รูป 2.24 อัลกอริทึม A*

จากรูปจะเห็นได้ว่าในขั้นตอนที่ 4 สถานะ ‘C’ จะถูกเลือกมากระจายโดยอัลกอริทึม A* เนื่องจากมีค่า f’ น้อยสุดเท่ากับ 3.5 ซึ่งน้อยกว่า ‘E’ ที่มีค่าเท่ากับ 4 แม้ว่าค่า h’ ของ E จะน้อยกว่า ซึ่งต่างจากการสร้างสถานะของอัลกอริทึมดีสุดก่อน

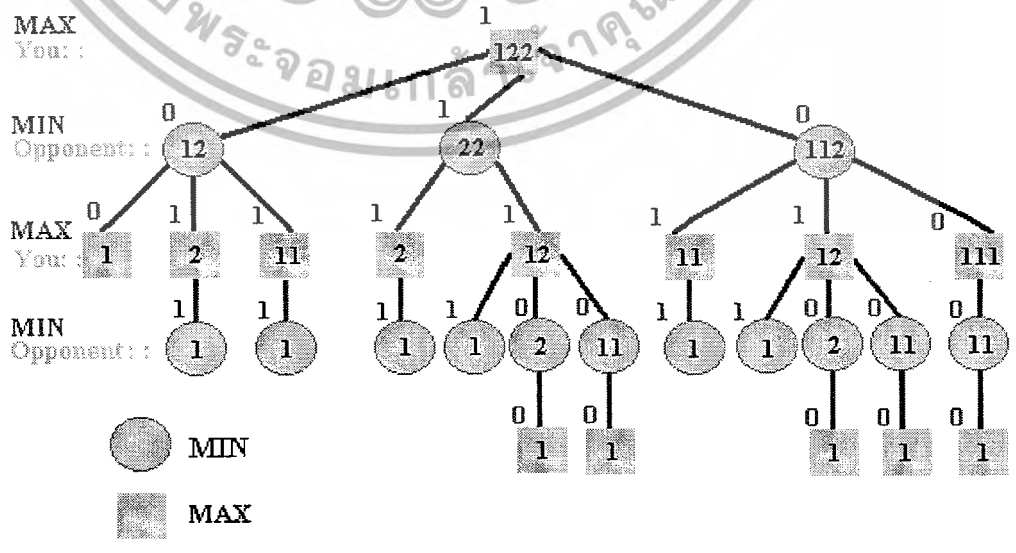
บทที่ 3
พื้นฐานความรู้เรื่องการเขียนโปรแกรมเกมส์หมากกระดาน

3.1 แผนภูมิต้นไม้ของเกม (Game Tree)

ในการพิจารณาเกมประเภทที่มีผู้เล่นสองฝ่ายที่ผลัดกันเล่น ซึ่งเกมส์นี้ปกติแล้วสามารถแทนได้ด้วยแผนภูมิต้นไม้ที่มีโหนดต่างๆแทนสถานะปัจจุบันของเกมส์และเส้นโค้ง (arc) หลายๆเส้นพุ่งออกจากโหนดแทนการเล่นเกมส์แต่ละแบบ แผนภูมินี้ประกอบด้วยการเปลี่ยนแปลงที่เกิดขึ้นจากการเล่นทุกๆแบบที่เป็นไป ได้ของผู้เล่นเริ่มต้นจากโหนดตั้งต้น เส้นโค้งแต่ละเส้นจะไปสิ้นสุดที่โหนดลูกๆ ซึ่งหมายถึงผู้เล่นฝ่ายตรงข้ามแล้วถัดมาก็เป็นตาผู้เล่นฝ่ายแรกบ้าง อย่างนี้เรื่อยไปตามแต่ที่เกมส์จะเป็นไปได้ และที่ปลายสุดของโหนดลำดับสุดท้ายของแผนภูมิจะแทนผลลัพธ์ของเกมส์เมื่อเกมส์สิ้นสุดลง (ชนะ,แพ้,เสมอ) และจะมีคะแนนแสดงไว้ที่โหนดลำดับสุดท้ายนี้ทุกๆโหนด ยกตัวอย่างเช่น เราอาจจะให้คะแนนเท่ากับ 1 เมื่อชนะ, 0 เมื่อเสมอ และ –1 เมื่อแพ้ เป็นต้น

3.1.1 ตัวอย่าง : เกมเลือกหยิบแท่งไม้

เกมส์นี้จะมีอุปกรณ์เป็นแท่งไม้จำนวนหนึ่งถูกแบ่งออกเป็นหลายๆกลุ่ม กลุ่มหนึ่งๆจะประกอบด้วยแท่งไม้อย่างน้อยหนึ่งแท่ง เราสามารถแทนกลุ่มของแท่งไม้เหล่านี้ได้ด้วยลำดับของตัวเลขได้เช่น (1,3,5,7) หรือ (2,2,3,9,110) ผู้เล่นสามารถเลือกหยิบแท่งไม้จากกลุ่มใดก็ได้เพียงหนึ่งกลุ่ม จำนวนแท่งที่หยิบไม่จำกัดถ้าไม่เกินที่กลุ่มนั้นมีอยู่แต่ไม่ต่ำกว่า 1 แท่ง ดังนั้น (1,3,5,7) ก็อาจกลายเป็น (1,1,3,5) ถ้าผู้เล่นดึงแท่งไม้จากกลุ่มสุดท้ายออกมา 6 แท่งกติกาคือผู้เล่นที่หยิบแท่งไม้ออกเป็นแท่งสุดท้ายเป็นฝ่ายแพ้ เกมส์ (1,2,2) แทนได้ด้วยรูป 3.1



รูป 3.1 Game Tree for (1,2,2)

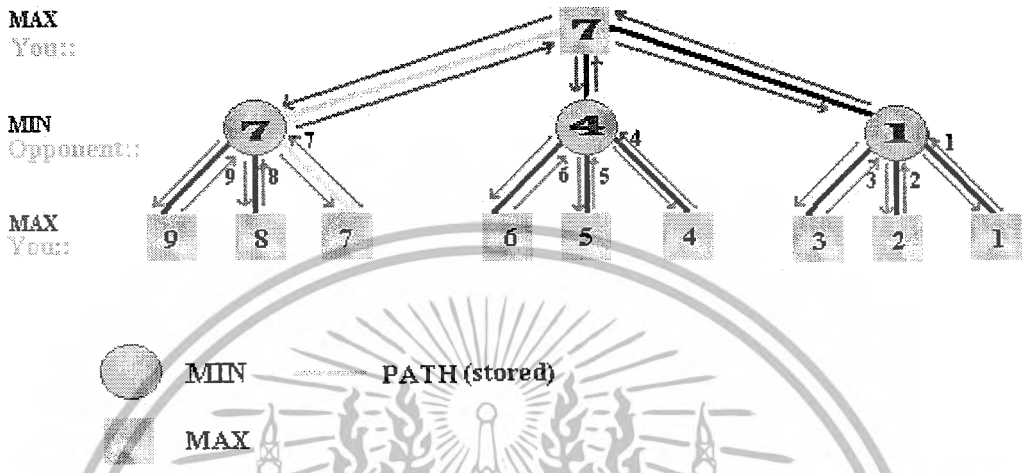
ในรูป 3.1 ตัวเลขที่โหนดแรกระบุว่าในตอนเริ่มต้นแท่งไม้มีทั้งสิ้น 5 แท่ง แบ่งเป็น 3 กลุ่ม คือ 1,2,2 ถ้าคุณเป็นผู้เล่นคนแรก คุณก็อาจจะดึงแท่งไม้มาได้ 1 หรือ 2 แท่ง หลังจากนั้นก็ถึงคราวฝ่ายตรงข้ามบ้างและจำนวนตัวเลขที่โหนดจะบอกถึงแท่งไม้ที่เหลืออยู่ เมื่อฝ่ายตรงข้ามดึงเอาแท่งไม้ออกไป 1 หรือ 2 แท่งบ้าง สถานะขณะนั้นก็จะถูกแสดงในโหนดลำดับถัดไป เกมส์จะเป็นเช่นนี้เรื่อยไปจนกว่าจะมีแท่งไม้เหลือเพียงแท่งเดียว

คราวนี้เราจะนำแผนภูมิที่ได้มาวิเคราะห์หาวิธีเล่นที่ดีที่สุดนั่นคือทำให้ฝ่ายตรงข้ามแพ้ นั่นคือเราควรเล่นโดยทำให้ฝ่ายเราได้คะแนนสูงสุด และบังคับให้ฝ่ายตรงข้ามได้คะแนนต่ำสุด ดังนั้นนโหนด MAX จะเป็นโหนดสถานะของผู้เล่นฝ่ายเราและโหนด MIN จะเป็นโหนดสถานะของฝ่ายตรงข้าม เนื่องจากเป้าหมายของเกมส์นี้คือผู้เล่นคนสุดท้ายเป็นฝ่ายแพ้ ดังนั้นคะแนนจะเป็น 0 ถ้าโหนดระดับล่างสุดเป็นโหนด MAX และคะแนนจะเท่ากับ 1 ถ้าโหนดระดับล่างสุดเป็นโหนด MIN จากนั้นเราค่อยมาดูคะแนนที่โหนดระดับกลางๆ โดยอาศัยคะแนนจากโหนดระดับล่างสุด ไล่ขึ้นไปทีละระดับ ที่โหนด MAX เราจะคิดคะแนนจากคะแนนสูงสุดของโหนดลูกของมันทั้งหมด เพราะเราย่อมเล่นเกมส์ในวิธีที่จะทำให้เราชนะมากที่สุด (ลองคิดถึงเกมส์อื่นๆที่มีระดับคะแนนมากกว่า 3 ระดับ เช่น หมากกระดาน ซึ่งตัดสินคะแนนจากสภาพหมากบนกระดาน แน่นอนว่าเราไม่สามารถแสดงโหนดลูกทั้งหมดได้เพราะติดที่เรามีขีดจำกัดของ เวลา จึงต้องกำหนดความลึกไว้เป็นค่าหนึ่งๆ แล้วคิดคะแนนจากสภาพหมากที่ระดับความลึกนั้น) ส่วนที่โหนด MIN เราเลือกคะแนนต่ำสุดของงโหนดลูก เพราะผู้เล่นฝ่ายตรงข้ามที่เก่งๆ ย่อมเล่นเกมส์ในวิธีที่ทำให้คะแนนของเราต่ำที่สุดเท่าที่จะเป็นไปได้ สังเกตว่าการคำนวณคะแนนจากตัวเกมส์จริงๆทำที่สถานะที่ลึกสุดของแผนภูมิเท่านั้น ในรูป 3.1 โหนดแรกจะมีคะแนนเท่ากับ 1 แสดงว่าผู้เล่นฝ่ายแรกจะเป็นผู้ชนะเสมอ (สำหรับเกมส์ (1,2,2)) หากผู้เล่นนั้นเลือกเล่นวิธีที่โหนดลูกของมันมีคะแนนเท่ากับ 1 ทุกครั้ง

3.2 แผนภูมิต้นไม้แบบ Minimax

แผนภูมิชนิดนี้ก็คือแผนภูมิที่ได้กล่าวถึงแล้ว ผู้เล่นคนแรกจะอยู่ที่ระดับที่เป็นเลขคี่ ส่วนผู้เล่นอีกฝ่ายก็อยู่ที่โหนดที่มีระดับเป็นเลขคู่ ในเกมส์ที่มีผู้ เล่นสองคน ผู้เล่นฝ่ายแรกเล่นเกมส์แบบคะแนนสูงสุด ผู้เล่นฝ่ายที่สองเล่นแบบคะแนนต่ำสุด การค้นหาแบบมินิแมกซ์ ทำเพื่อหาความเป็นไปได้ทั้งหมดของเกมส์และผลของมันจนถึงระดับความลึกที่ต้องการ คะแนนที่ความลึกสูงสุด (หรือโหนดปลายสุด) จะถูกคำนวณไว้แต่ต้นแล้ว (เหลือแต่การจะค้นหาว่าลำดับการเล่นวิธีใดที่ดีที่สุดของเราเท่านั้น) การค้นหาจะเป็นการไล่ให้คะแนนจากระดับล่างขึ้นไปจนถึงระดับบนพร้อมทั้ง เปรียบเทียบกับคะแนนจากโหนดอื่นๆ ที่อยู่ในระดับเดียวกันและมีโหนดพ่อแม่ (parent node) ร่วมกัน

เนื่องจากเราทราบผลลัพธ์ล่วงหน้าแล้ว (กับผู้เล่นฝ่ายตรงข้ามที่เก่งๆ) ผลก็จะเหมือนกับผลลัพธ์ของเกมส์โดยผู้เล่นที่เก่งสุดๆสองฝ่ายเล่นกัน โดยเริ่มจากสถานะเริ่มต้นที่กำหนดให้ ความซับซ้อนของอัลกอริทึมจะเท่ากับจำนวนโหนดทั้งหมดในแผนภูมิ

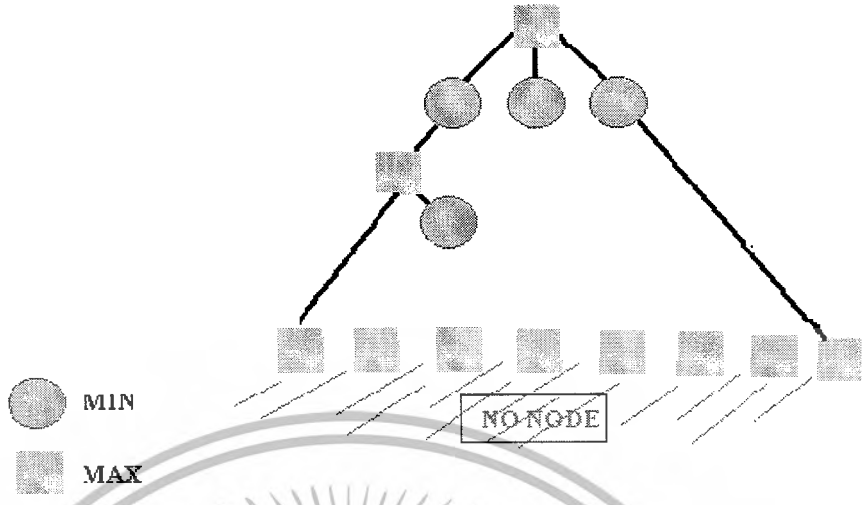


รูป 3.2 Minimax Search

รูป 3.2 แสดงการทำงานของอัลกอริทึมมินิแมกซ์ แผนภูมิต้นไม้ที่ได้จะประกอบไปด้วยการเล่นที่เป็นไปได้ทั้งหมด โหนดราก(โหนดแรกสุด) แทนผู้เล่นคนแรกและโหนดลูกๆแทนผู้เล่นฝ่ายตรงข้าม โหนดปลายสุดจะระบุคะแนน เราต้องการค้นหาการเล่นวิธีที่ดีที่สุดของผู้เล่นคนแรกดังนั้นเราจึงเลือก คะแนนสูงสุดของโหนดลูกๆของมัน ส่วนที่โหนดของผู้เล่นฝ่ายตรงข้ามเราเลือกคะแนนต่ำสุดของโหนดลูกๆของมัน สำหรับโหนด MIN คะแนนเริ่มแรกก่อนการเปรียบเทียบจะเท่ากับ +infinity และลดลงเรื่อยๆเมื่อโปรแกรมทำงานต่อไป ส่วนที่โหนด MAX คะแนนจะเริ่มที่ –infinity และเพิ่มขึ้นเรื่อยๆเมื่อเวลาผ่านไป

3.3 แผนภูมิต้นไม้ที่มีขนาดใหญ่จนต้องกำหนดขอบเขตการค้นหาล่วงหน้า

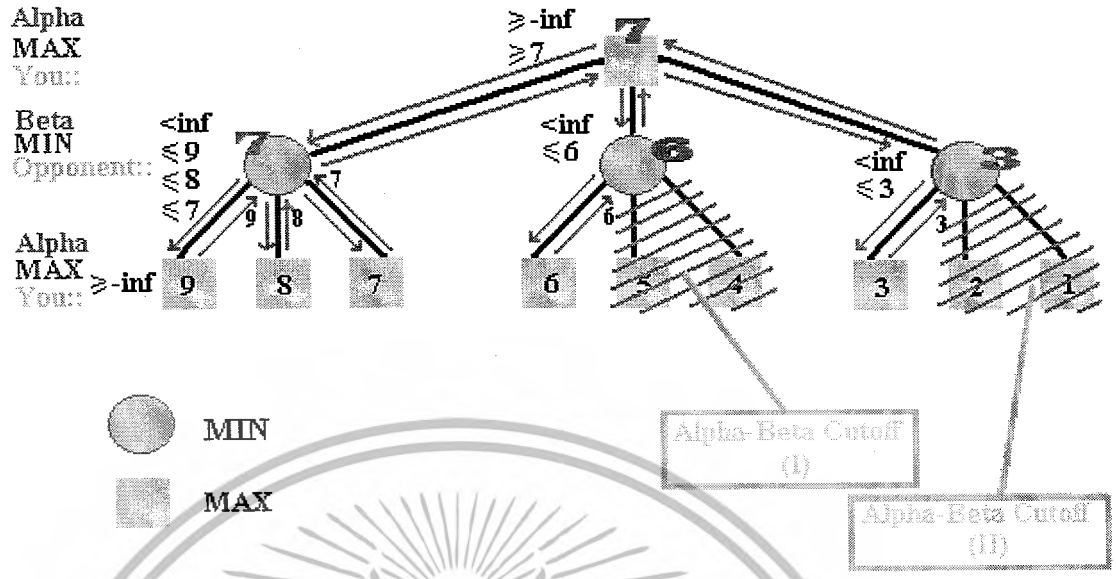
ในแผนภูมิต้นไม้ใหญ่ๆย่อมเป็นไป ไม่ได้ที่จะค้นหาทุกๆโหนด วิธีที่ดีที่สุดคือตัดการค้นหาของแผนภูมิให้เหลือระดับที่ไม่มากนัก และคิดเสียว่าเป็นการประมาณที่ดีของแผนภูมิต้นไม้ทั้งหมด (ที่ไม่มีใครรู้) และคิดคะแนนให้กับโหนดที่ระดับความลึกสูงสุดของการค้นหา ความแตกต่างในตอนนี้คือคะแนนที่ได้ไม่ใช่คะแนนที่ถูกต้องแน่นอนอีกต่อไปแต่ เป็นคะแนนจากการคาดการณ์อย่างมีเหตุผล คะแนนตรงนี้จะได้มาจากการคำนวณโดยสิ่งที่เรียกว่า evaluation function ซึ่งมาจากผู้เขียนโปรแกรมโดยอาศัยความรู้ความเข้าใจในเกมส์บวกกับประสพการณ์ เรายังคงใช้อัลกอริทึมแบบมินิแมกซ์เพื่อคำนวณคะแนนได้เช่นกัน



รูป 3.3 แผนภูมิต้นไม้ขนาดใหญ่

3.4 การค้นหาโดยใช้วิธีการของ Alpha-Beta pruning เข้าช่วย

การค้นหาแบบ Alpha-Beta เป็นวิธีการค้นหาซึ่งลดจำนวนโหนดจากการค้นหาแบบมินิแมกซ์ลงได้จำนวนหนึ่ง ทำให้สามารถลดเวลาในการค้นหา โดยตัดการค้นหาส่วนหนึ่งซึ่งเสียไปกับการเล่นแบบต่างๆที่อัลกอริทึมเห็นว่า ไม่ดีอย่างชัดเจนสำหรับผู้เล่นฝ่ายที่เรากำลังสนใจ วิธีการของอัลฟาเบต้าคือเก็บวิธีการเดินที่ดีที่สุดของแต่ละฝ่ายเอาไว้ตลอด การค้นหา เช่นเดียวกับอัลกอริทึมแบบมินิแมก คะแนนเริ่มด้วย +infinity สำหรับโหนด MIN และลดลงตามเวลา ส่วนโหนด MAX คะแนนเริ่มด้วย –infinity และเพิ่มขึ้นตามเวลา ประสิทธิภาพของอัลฟาเบต้าที่โหนดต่างๆ ขึ้นกับลำดับของโหนดลูกๆของมันว่ามีลำดับของคะแนนอย่างไร หากเราโชคดี ที่โหนด MIN เราอาจได้ลำดับโหนดจากคะแนนต่ำไปหาสูง และที่โหนด MAX เราอาจจะได้ลำดับของโหนดที่มีคะแนนจากสูงไปหาต่ำ โดยทั่วไปแล้วในสถานการณ์ที่ดีที่สุดของการค้นหาแบบอัลฟาเบต้านี้จะให้จำนวน โหนดปลายสุดเท่ากับมินิแมกซ์สำหรับแผนภูมิต้นไม้แต่มีความลึกมากกว่าเป็นสองเท่า



รูป 3.4 Alpha-beta search

เริ่มแรกเราอยู่ที่โหนดราก ยังมีค่าเริ่มต้นของคะแนนเท่ากับ –inf หรือ alpha = -infinity ซึ่งเป็นค่าที่ยังนำไปใช้ไม่ได้แน่นอน โปรแกรมเรียกใช้ evaluatemax และการค้นหาจะลงมาตามเส้นทางซ้ายสุด ซึ่ง beta = -infinity จากนั้นจะเรียกใช้ evaluatemin และ ลงมาที่โหนดระดับล่างสุดซึ่งก็มี alpha = +infinity เช่นกัน เมื่อเรียก evaluatemax ครั้งนี้ จะได้ค่า return ออกมาทันทีเท่ากับ 9 ค่า alpha=9 นี้จะถูกส่งกลับไปเก็บไว้เป็น beta ของ โหนด MIN และเมื่อวนลูปเพื่อเปรียบเทียบหาค่า beta ที่ต่ำสุด แล้ว สุดท้ายโหนด MIN โหนดแรกจะได้ค่า beta = 7 และ ถูก return เป็นค่า alpha ของโหนดราก จากนั้นก็วนลูป หาค่าสูงสุดของalpha ที่โหนดรากต่อไป สิ่งที่น่าสนใจคือ alpha-beta ลดการค้นหาได้อย่างไร คำตอบก็คือ ในการพยายามหาค่า alpha ครั้งต่อไปเพื่อนำมาเปรียบเทียบกับค่าแรก(= 7) ซึ่งคราวนี้ได้ alpha = 6 หากค่าคะแนนในโหนดล่างสุด จำนวนสามโหนดถูกเรียงลำดับจากมากไปน้อยอยู่แล้ว ทำให้โหนด MIN มี ค่า beta =6 เมื่อนำไปเปรียบเทียบกับค่า alpha = 7 ของโหนดราก แล้ว เห็นได้ว่า beta มีค่าน้อยกว่า ก็ออกจากลูปได้ทันที ซึ่งเรียกส่วนที่ตัดทิ้งไปนี้ว่า Alpha-Beta Cutoff ส่วน การเรียงลำดับของคะแนนไว้ตั้งแต่แรกนั้นเรียกว่า move ordering

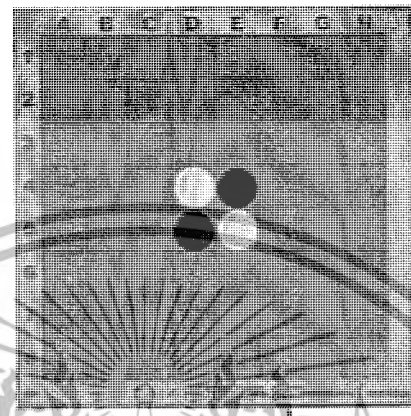
อัลกอริทึมของอัลฟาเบต้าประกอบไปด้วยฟังก์ชันสองฟังก์ชันคือ evaluatemin และ evaluatemax ถ้าเรากำลังอยู่ที่โหนด MIN เราก็จะใช้ฟังก์ชัน evaluatemin เช่นเดียวกับถ้าเราอยู่ที่โหนด MAX เราก็จะใช้ฟังก์ชัน evaluatemax B คือค่าคะแนนที่ใช้ในตอนเริ่มต้นของอัลกอริทึม เช่น +infinity

3.5 เกมส์โอเทลโล่

3.5.1 การเริ่มต้นเกมส์

1) ผู้เล่นทำการแบ่งจำนวนหมากฝ่ายละ 32 ตัว และทำการเลือกสีที่ต้องการเล่น

2) นำหมากของแต่ละฝ่ายมาวางในตำแหน่งเริ่มต้น ดังรูป 3.5



รูป 3.5 แสดงตำแหน่งเริ่มต้นเกมส์

3) การเดินหมากในเกมส์โอเทลโล่ ผู้เล่นจะต้องทำการหนีบหมากของฝ่ายตรงข้ามเสมอ (ดูรูป 3.6 และ 3.7 ประกอบ) โดยหมากที่ถูกหนีบจะถูกพลิกให้เป็นสีของฝ่ายที่หนีบ

รูป 3.6 หมากดำต้องการเดินในตำแหน่งดังรูป

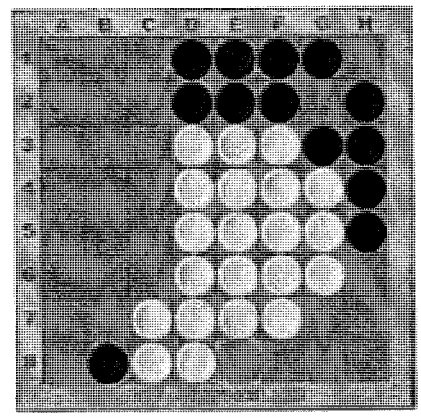
รูป 3.7 หลังจากลงตำแหน่งดังกล่าว หมากขาวที่อยู่ระหว่างหมากดำจะถูกพลิกเป็นสีดำ

3.5.2 กฎ และกติกา

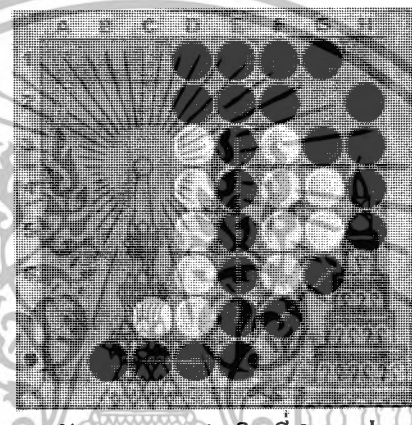
1) ผู้เล่นผลัดกันเดิน โดยฝ่ายที่เล่นสีดำเดินก่อนเสมอ

2) กรณีที่ผู้เล่นฝ่ายใดฝ่ายหนึ่งไม่มีตาเดิน ผู้เล่นฝ่ายนั้นจำเป็นต้องผ่าน เพื่อให้ผู้เล่นอีกฝ่ายหนึ่งเดิน จนกว่าผู้เล่นฝ่ายที่ผ่านจะมีตาเดินจึงสามารถเดินต่อไปได้

3) การเดินหมาก หมากที่ถูกหนีบจะถูกพลิกเปลี่ยนสีทุกแนว และไม่จำกัดจำนวน



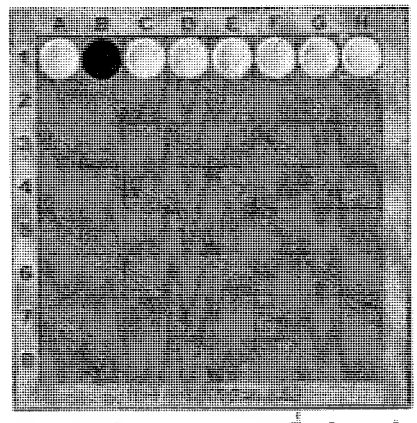
รูป 3.8 หมากดำต้องการเดินในตำแหน่ง E8



รูป 3.9 หลังจากหมากดำเดินที่ตำแหน่ง E8 แล้ว

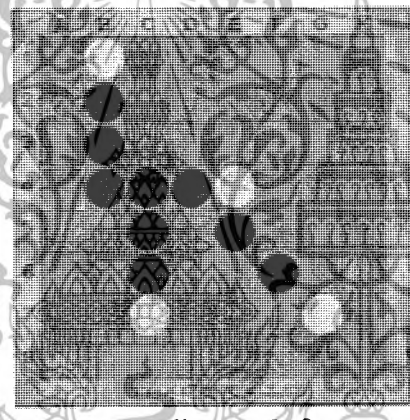
4) ไม่สามารถหนีบข้ามหมากสีของตนเองได้

รูป 3.10 หมากขาวต้องการเดินในตำแหน่ง H1

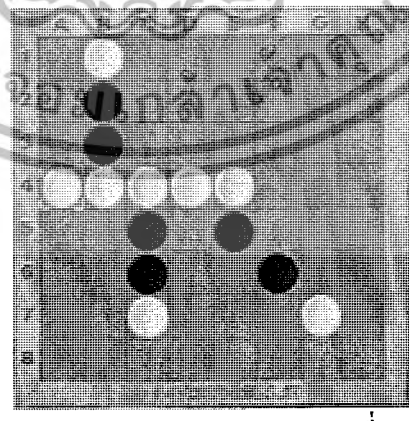


รูป 3.11 หมากดำจะถูกพลิกเฉพาะหมากที่อยู่ระหว่าง C1-H1 เท่านั้น

5) ในการเดินหมากแต่ละครั้ง หมากที่ถูกพลิกจะพลิกเฉพาะตัวที่ถูกหนีบจากตำแหน่งที่ผู้เล่นเดินในตาเดินล่าสุดเท่านั้น



รูป 3.12 หมากขาวต้องการเดินในตำแหน่ง A4



รูป 3.13 หลังจากหมากขาวเดินที่ A4

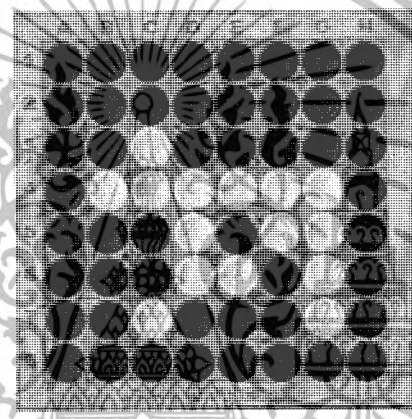
6) หมากที่ถูกหนีบในแต่ละตาเดินจะต้องถูกพลิก แม้ว่าผู้เล่นจะไม่ต้องการพลิกก็ตาม

7) หากผู้เล่นลืมพลิกหมากในบางตำแหน่ง หรือ พลิกผิด ผู้เล่นฝ่ายตรงข้ามสามารถโต้แย้ง เพื่อให้ทำการแก้ไขได้ แต่หากผู้เล่นฝ่ายตรงกันข้ามได้ทำการเดินในตาต่อไปแล้ว จะสายเกินไปไม่สามารถกลับมาแก้ไขได้อีก

8) เมื่อได้ทำการเดินหมากลงบนกระดานในตำแหน่งใดๆแล้วก็ตามจะไม่สามารถเปลี่ยนไปยังตำแหน่งอื่นได้อีกตลอดทั้งเกมส์

9) หากผู้เล่นฝ่ายใดฝ่ายหนึ่งตัวหมากที่มีหมดแต่ยังมีตาเดินต่อ ให้ผู้เล่นฝ่ายตรงข้ามแบ่งตัวหมากให้ฝ่ายนั้น เพื่อทำการเดินต่อจนจบเกมส์

10) เกมส์จะสิ้นสุดลงเมื่อทั้งสองฝ่ายไม่มีตาเดิน โดยฝ่ายที่มีจำนวนหมากบนกระดานมากกว่าจะเป็นฝ่ายชนะ

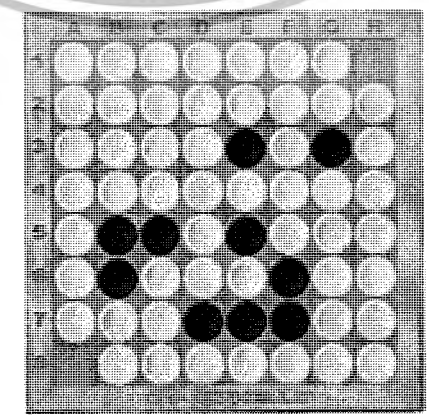


รูป 3.14 จบเกมส์หมากตำชนะ 49-15

3.5.3 หมายเหตุ

1) เกมส์อาจสิ้นสุดลงได้โดยการเสมอ หากทั้งสองฝ่ายมีจำนวนหมากเท่ากัน

2) ในบางครั้ง เกมส์สามารถสิ้นสุดลงได้โดยที่ยังเหลือช่องว่างอยู่บนกระดาน



รูป 3.15 จบเกมส์หมากขาวชนะ 10-52

3) ตามกติกาการแข่งขันสากลจะนับช่องว่างที่เหลือบนกระดานเพิ่มเป็นคะแนนให้กับฝ่ายที่ชนะ ดังนั้นรูป 3.15 หมากขาวจะชนะ 10-54

3.6 องค์ประกอบพื้นฐานของโปรแกรม

3.6.1 Board Representation, Input Output Functions

เป็นก้าวแรกของการสร้างโปรแกรมโอเทลโล่ เมื่อพูดถึงโปรแกรมส่วนแรกก็คือต้องมีหน้าตาให้ผู้เล่นชมว่าขณะนั้นๆ หมากในกระดานเป็นอย่างไร (Output) ขณะเดียวกันก็ต้องมีส่วนรับข้อมูลจากผู้เล่นว่าเดินอะไร (Input) เพื่อความง่ายไม่ยุ่งยากจึงใช้คำสั่งจาก Keyboard ในลักษณะของ Console Application เมื่อได้ข้อมูลที่เปลี่ยนแปลงก็นำข้อมูลนั้นไปยังส่วนที่เก็บข้อมูลในโปรแกรม (Board Representation) เพื่อให้คิดคำนวณพิจารณาตาเดินที่เป็นไปได้ทั้งหมดและในจำนวนนั้นเลือกตาที่ดีที่สุดแล้วส่งกลับมายัง Output แสดงผลให้เห็นเป็นกระดาน

Board Representation หมายถึงการเก็บข้อมูลตัวหมากแต่ละตัวเพื่อนำไปแสดงผลออกทางหน้าจอ หรือนำไปคำนวณเพื่อหาตาเดินที่ดีที่สุด สามารถเลือกได้หลายวิธี วิธีที่ดีที่สุดคือ Bitboard คือการเก็บข้อมูลในรูปของตัวเลข Integer โดยที่แต่ละ bit ของตัวเลขนั้นจะเป็นการแทนข้อมูลของตัวหมากต่างๆ บนกระดาน แต่เนื่องจากความยุ่งยาก และเพื่อความง่ายในการทำความเข้าใจจึงใช้วิธี Array Board ซึ่งเข้าใจง่ายที่สุดโดยใช้ board[64] เพื่อเก็บข้อมูลทั้งหมดของกระดาน

3.6.2 Initialization Functions

คือ Function ที่กำหนดความตัวแปรต่างๆ ในช่วงเริ่มต้นเกมส์โดยรวมๆก็มี 2 กลุ่มหลักๆ คือ

1) ตัวแปรกลุ่มที่ต้องกำหนดใหม่เมื่อเริ่มเกมส์ใหม่ หรือค่าคงที่ที่จำเป็นในส่วนต่างๆ ของโปรแกรมโดยทั่วไปก็เป็นตัวแปรเกี่ยวกับข้อมูลทั้งหมดของกระดานเช่น จำนวนเบี้ยของแต่ละฝ่ายเมื่อเริ่มต้นเล่นมีฝ่ายละ 2 ตัว และกำหนดฝ่ายเดินก่อน

2) กลุ่มตัวแปรที่ใช้ใน Evaluation

3.6.3 Move Generation Function

เมื่อเราเก็บข้อมูลของกระดานหมากฮอสทุกอย่างอยู่ในรูปของ array[64] และใช้ตัวเลข Integer แทนตัวหมาก และช่องว่าง จากนั้นเราต้องทราบว่าในตำแหน่งดังกล่าว ฝ่ายใดเป็นฝ่ายเดินและมีตาเดินอะไรได้บ้าง เราต้องเตรียมตัวแปรจำนวนหนึ่งเพื่อเก็บข้อมูลของตาเดินแต่ละตา

3.6.4 Makemove and Unmakemove Functions

หมายถึงให้โปรแกรมเดินหน้าและย้อนกลับ ใน Makemove() ก็คือให้เดินไปตามตาเดินนั้นๆ ส่วน Unmakemove() คือ ย้อนตาเดินที่เพิ่งจะเดินไป ในส่วนของ Makemove() มีองค์ประกอบการทำงานหลักอยู่สองส่วนคือ

1) การเก็บข้อมูลทั้งหมดของตาเดินนั้นๆ ก่อนที่จะมีการเปลี่ยนแปลงใดๆ เพื่อที่จะได้ย้อนกลับได้อย่างถูกต้อง ซึ่งตรงนี้จะใช้ใน Search Function
2) ทำการเปลี่ยนแปลงข้อมูลในตำแหน่งต่างๆ ที่เราได้จากตาเดินนั้นๆ ใน board[35] ที่เราเก็บข้อมูลไว้

ส่วน Unmakemove() ก็เช่นกันแต่ทำในลักษณะตรงข้ามกับ Makemove() โดยนำข้อมูลของตาเดินนั้นๆ มาจากที่เก็บไว้เดิม แล้วทำย้อนกลับซึ่งตรงนี้จำเป็นมากในการวิเคราะห์หาตาเดิน

3.6.5 Static Evaluation Function

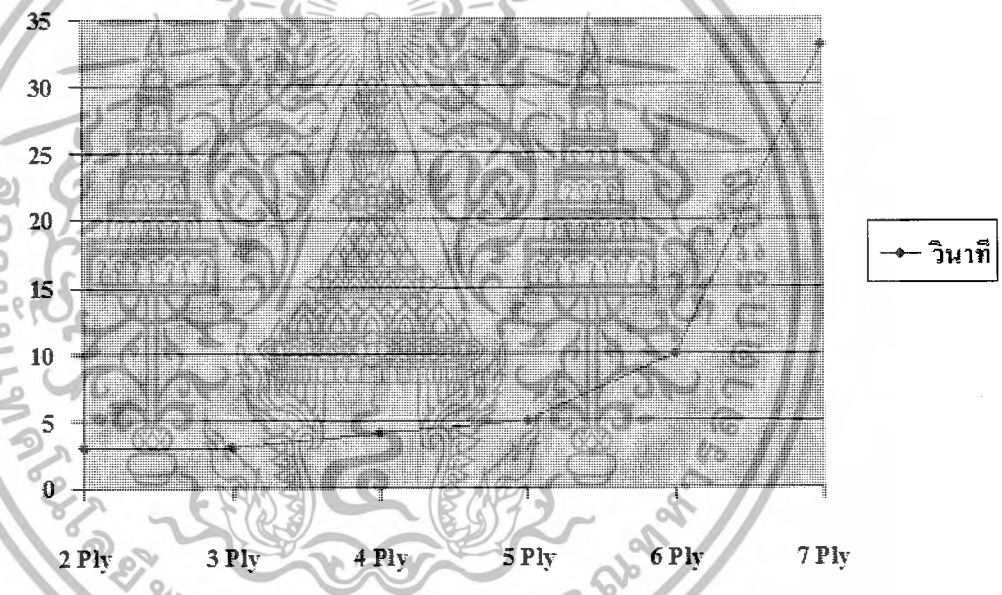
ความหมายก็คือ ภายใต้เทคโนโลยีปัจจุบัน ในเมื่อเราไม่สามารถให้คอมพิวเตอร์คิดตั้งแต่ตาแรกจนถึงตาสุดท้ายได้ เมื่อความลึกถึงระดับที่เราต้องการ เราจึงต้องมี Function นี้เพื่อประเมินอย่างคร่าวๆ ถึงสภาพการณ์ว่าฝ่ายใดเป็นต่อ หรือเป็นรองประมาณเท่าไร แล้วส่งค่ากลับขึ้นมาเป็นตัวเลขเพื่อการเปรียบเทียบกับตาเดินตาอื่นๆ โดยทั่วไปหลักการอย่างง่ายที่ควรมีเป็นองค์ประกอบใน Evaluation Function คือ

1) Material หรือจำนวนตัวใครมีตัวมากว่ากัน อันนี้เป็นสิ่งที่เข้าใจง่ายที่สุด ฝ่ายตัวมากกว่าย่อมดีกว่าฝ่ายตัวน้อย
2) ตำแหน่งของเบี้ย เนื่องจากตำแหน่งต่างๆบนกระดานจะทำให้มีโอกาสชนะที่แตกต่างกันออกไป

บทที่ 4
ผลการทดลองและการวิเคราะห์

4.1 เปรียบเทียบเวลาที่ใช้ในระดับความลึกที่แตกต่างกัน

ทำการทดลองโดยการให้คอมพิวเตอร์เป็นผู้เล่นทั้งสองฝ่าย โดยให้ฝ่ายหนึ่งเป็น min และอีกฝ่ายหนึ่งเป็น max ทำการจับเวลาของการเล่นจบหนึ่งเกม ซึ่งผู้เล่นฝ่าย min จะทำการค้นหาที่ระดับความลึก 2 Ply เสมอ ส่วนผู้เล่นฝ่าย max จะเปลี่ยนระดับการค้นหาที่แตกต่างกันทั้งหมด 6 ระดับด้วยกัน ผลการทดลองที่ได้แสดงดังรูปต่อไปนี้

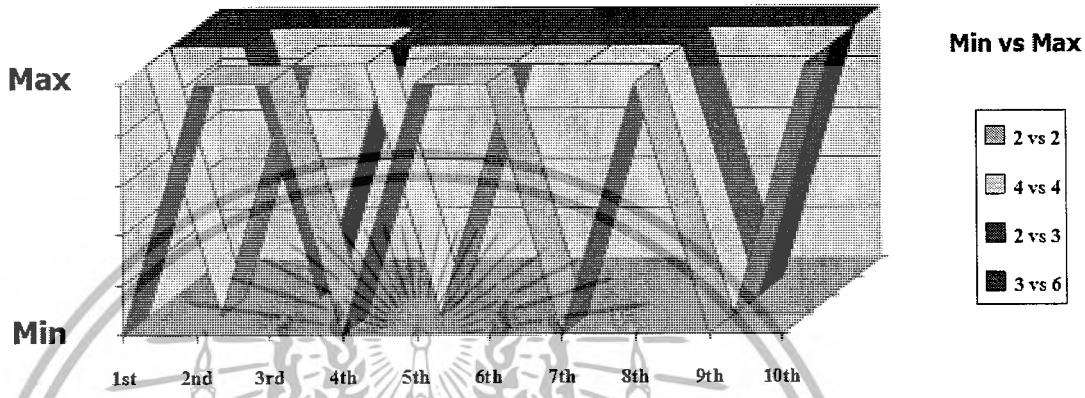


รูป 4.1 กราฟผลการทดลองเปรียบเทียบเวลาที่ใช้

จากการทดลองจะพบว่าในการค้นหาที่ระดับความลึกที่แตกต่างกันนั้น การค้นหาที่ระดับความลึกที่มากกว่าย่อมที่จะต้องใช้เวลาสูงกว่าการค้นหาที่ระดับความลึกน้อยกว่าเนื่องจากวิธีการค้นหาของ Minimax นั้นจะเป็นการค้นหาในลักษณะของ depth first search และจะต้องทำการประเมินค่าที่ node ปลายสุดของ tree ทั้งหมดก่อน และการเพิ่มขึ้นของจำนวน node เมื่อทำการค้นหาที่ระดับความลึกเพิ่มขึ้นจะเป็นในลักษณะของเอ็กซ์โพเนนเชียล เวลาที่ใช้ในการค้นหาจึงเป็นไปในลักษณะเดียวกัน

4.2 เปรียบเทียบผลลัพธ์ในระดับความลึกที่แตกต่างกัน

ทำการทดลองโดยการให้คอมพิวเตอร์เป็นผู้เล่นทั้งสองฝ่าย โดยให้ฝ่ายหนึ่งเป็น min และอีกฝ่ายหนึ่งเป็น max นำผลที่ได้จากการเล่นระหว่างทั้งสองฝ่ายมาเปรียบเทียบผลลัพธ์ที่ระดับความลึกที่แตกต่างกัน ซึ่งในการทดลองแบ่งการทดลองเป็น 4 รูปแบบด้วยกันคือ 2 vs 2 , 4 vs 4 , 2 vs 3 , 3 vs 6 และทดลองรูปแบบละ 10 ครั้ง ผลการทดลองที่ได้แสดงดังรูปต่อไปนี้



รูป 4.2 กราฟผลการทดลองเปรียบเทียบผลลัพธ์

จากการทดลองจะพบว่าในการค้นหาที่ระดับความลึกที่แตกต่างกัน โอกาสที่จะทำให้ชนะมากกว่าจะขึ้นอยู่กับระดับความลึกที่ทำการค้นหา โดยที่ฝ่ายที่ทำการค้นหาที่ระดับความลึกมากกว่าย่อมมีโอกาสชนะสูงกว่าฝ่ายที่ทำการค้นหาที่ระดับความลึกน้อยกว่า เนื่องจากวิธีการของ Minimax นั้นเป็นการมองตาเดินล่วงหน้าโดยที่จะเลือกเส้นที่ดีที่สุดของแต่ละฝ่าย ดังนั้นการค้นหาที่ระดับความลึกสูงๆย่อมทำให้โอกาสที่จะชนะสูงขึ้นตามไปด้วย เหมือนกับการที่มนุษย์คิดตาเดินล่วงหน้าเอาไว้ในใจนั่นเอง

4.3 สรุปผลการทดลอง

จากการทดลองสร้างเกมต่างๆแล้วทดสอบเล่นดูพบว่า เกมประเภทสลับกันเล่นนั้นถึงแม้ว่าจะเหมาะกับการนำวิธี Minimax มาใช้ก็ตาม แต่ประสิทธิ์ภาพของ AI นั้นก็ขึ้นอยู่กับความลึกที่ทำการค้นหาลึกลงไป ซึ่งยิ่งค้นหาที่ความลึกมากเท่าไหร่ผลลัพธ์ที่ได้ก็จะมีโอกาสชนะมากขึ้นเท่านั้น แต่ก็ต้องแลกกับเวลาที่ใช้ในการค้นหาที่เพิ่มมากขึ้น นอกจากนี้การนำวิธี Alpha-Beta เข้ามาช่วยนั้นจะทำให้ช่วยลดภาระในการค้นหาลงไปได้มาก โดย Alpha-Beta จะตัดตาเดินที่ไม่จำเป็นต้องคิดออกไป ทำให้ช่วยลดจำนวนครั้งที่จะต้องใช้ในการประมวลผลไปได้มาก

ดังนั้นการนำวิธีการของปัญญาประดิษฐ์มาใช้แทนผู้เล่นที่เป็นมนุษย์ นอกเหนือจากการค้นหาคำตอบที่มีประสิทธิภาพแล้ว สิ่งที่สำคัญก็คือการทำให้ผู้เล่นอีกฝ่ายที่เป็นมนุษย์นั้นรู้สึกเหมือนกับ

ว่ามีมนุษย์มาเล่นด้วยนั่นเอง ใช่ว่าการสร้างปัญญาประดิษฐ์ที่เก่งที่สุดจะดีเสมอไป แต่จะต้องคำนึงถึงปัจจัยอื่นๆที่ทำให้คล้ายคลึงกับความเป็นมนุษย์มากที่สุด เช่น มนุษย์ย่อมมีความผิดพลาด หรือ มนุษย์สามารถมีการเรียนรู้เพื่อเพิ่มพูนความสามารถ เป็นต้น และการที่จะทำให้ AI มีความสามารถเช่นนั้นได้จะต้องนำวิธีการอื่นของปัญญาประดิษฐ์มาประยุกต์ใช้ร่วมกับวิธีที่ศึกษามานั่นเอง เช่น Machine Learning หรือ Neuron Network เป็นต้น

4.4 เครื่องมือ XNA Game Studio Express

การนำ เครื่องมือ XNA Game Studio Express เข้ามาช่วยในการพัฒนาเกมนั้นช่วยให้การพัฒนาเกมนั้นง่ายขึ้น โดยที่ไม่ต้องต้องสนใจเรื่องการจัดการ Windows, Graphics Card, Graphics Mode, Input, Message pump สิ่งที่ต้องสนใจมีอย่างเดียวคือการเขียน code เกม และการจัดการด้าน resource ก็ทำได้ง่าย โดยมีเครื่องมือที่ชื่อ "Content Pipeline" ที่ทำให้การส่ง content เข้าไปในเกมทำได้ง่ายทั้งการ import, compiling, และ loading ในเกม อีกทั้งยังมี "Starter Kits" ให้ คือเป็นเกมที่สมบูรณ์ทั้ง resource และ code ที่ใช้เป็น project template ได้เลย นักพัฒนาเกมสามารถเริ่มสร้างเกมโดยการปรับแต่ง template ดังกล่าวได้ทันที

บทที่ 5
บทสรุป

5.1 ปัญหาและอุปสรรค

1) ช่วงแรกของการพัฒนาเกมเกิดปัญหาในการเริ่มใช้เครื่องมือ XNA Game Studio Express เนื่องจากผู้จัดทำไม่คุ้นเคยกับภาษา C# มาก่อน แต่เมื่อได้เริ่มศึกษาอย่างจริงจังแล้วทำให้สามารถใช้งานได้คล่องขึ้นและ รู้สึกว่าง่ายต่อการใช้งาน
2) AI ที่ได้ทำการพัฒนานั้นอาจจะยังไม่สามารถนำไปใช้ในการพัฒนาเกมเพื่อทางธุรกิจได้ เนื่องจากยังต้องใช้ความสามารถของซีพียูเป็นอย่างมาก ซึ่งยังจะต้องทำการพัฒนาวิธีการให้ดียิ่งขึ้นต่อไป

5.2 แนวทางการศึกษาต่อ

1) ทำการศึกษาเพิ่มเติมเกี่ยวกับการพัฒนาเกมที่มีความซับซ้อนมากขึ้นไป เช่น หมากฮอส หรือ หมากรุก
2) ศึกษาและพัฒนาในส่วนของ Evaluation function ให้มีประสิทธิภาพและประสิทธิผลดียิ่งขึ้น โดยอาจจะใช้วิธีอื่นเพิ่มเติม เช่น Machine Learning หรือ Neuron Network เป็นต้น
3) พัฒนาโปรแกรมโอเทลโล่ที่สร้างด้วยเครื่องมือ XNA Game Studio ให้สมบูรณ์ยิ่งขึ้นโดยการเพิ่มความสามารถของ AI และการตั้งค่าอื่นๆของโปรแกรม

5.3 สรุป

จากการศึกษาและทดลองจะพบว่าเกมส์กระดานโดยทั่วไปจะนิยมใช้วิธีการของ Minimax และ Alpha-Beta pruning เป็นหลักการพื้นฐานเนื่องจากเป็นวิธีการที่เหมาะสม แต่ก็ยังสามารถนำหลักการของ AI อื่นๆเข้ามาประยุกต์ใช้ร่วมกับวิธีการของ Minimax และ Alpha-Beta pruning ได้ โดยจะต้องคำนึงถึงความเป็นไปได้และความเหมาะสมกับเกมที่เลือกพัฒนา ซึ่งวิธีการที่ใช้ในการศึกษาครั้งนี้จะสามารถพัฒนาต่อไปให้ดีขึ้นได้อีก โดยอาจจะประยุกต์และดัดแปลงวิธีการอื่นจากที่ได้ศึกษามาเพื่อให้สอดคล้องกับเกมที่ตนสนใจ เนื่องจากในการพัฒนาเกมไม่ได้มีแค่เพียงเกมกระดานเท่านั้น แต่ยังมีเกมอีกมากมายที่สามารถนำวิธีการของ AI มาใช้เพื่อให้ตัวเกมมีความน่าสนใจมากยิ่งขึ้น นอกเหนือจากแนวคิดของ AI แล้ว การที่จะพัฒนาเกมนั้นยังต้องอาศัยความคิดที่แปลกใหม่ และจะต้องสามารถนำมาใช้ได้จริง

บรรณานุกรม

Buckland, M. 2005. **Programming game AI by example :** Wordware Publishing, Inc.

Carter, C. 2009. **Microsoft XNA Game Studio 3.0 Unleashed** : Sams.

Hall, J. 2008. **XNA Game Studio Express :** Thomson.

Luger, G. F. 2002. **Artificial Intelligence: Structures and Strategies for Complex Problem Solving.** Fourth Edition : Addison Wesley.

Pfeifer, R. and Scheier, C. 1999. **Understanding Intelligence** : The Mit Press.

Rich, E. and Knight, K. 1991. **Artificial Intelligence. (International Edition)** : McGraw-Hill.

Russell, S. and Norvig, P. 1995. **Artificial Intelligence: A Modern Approach** : Prentice Hall.

XNA Community Thailand. 2009. **XNA Game Studio** [Online]. Available : http://www.thaixna.com/.

ภาคผนวก ก
คู่มือการใช้งานโปรแกรม

ก.1 การใช้งานโปรแกรมโอเทลโล่ที่พัฒนาด้วย Visual C#

ก.1.1 เมนูต่างๆ

ภายในโปรแกรมจะประกอบไปด้วยเมนูต่อไปนี้

1) เมนู Game ประกอบด้วยเมนูย่อยที่เกี่ยวข้องกับการจัดการภายในเกม ซึ่งจะมีเมนูย่อยดังนี้

- New Game ใช้ในการเริ่มเกม
- Resign Game ใช้ในการหยุดเกมเพื่อจะทำการเริ่มใหม่
- Options ใช้ในการตั้งค่าต่างๆภายในเกม เช่น สี , ผู้เล่น , ระดับความยาก เป็นต้น
- Statistics แสดงสถิติของเกม
- Exit ทำการออกจากเกม

2) เมนู Move ประกอบด้วยเมนูย่อยที่เกี่ยวข้องกับการจัดการการเดินของหมาก ซึ่งมีเมนูย่อยดังนี้

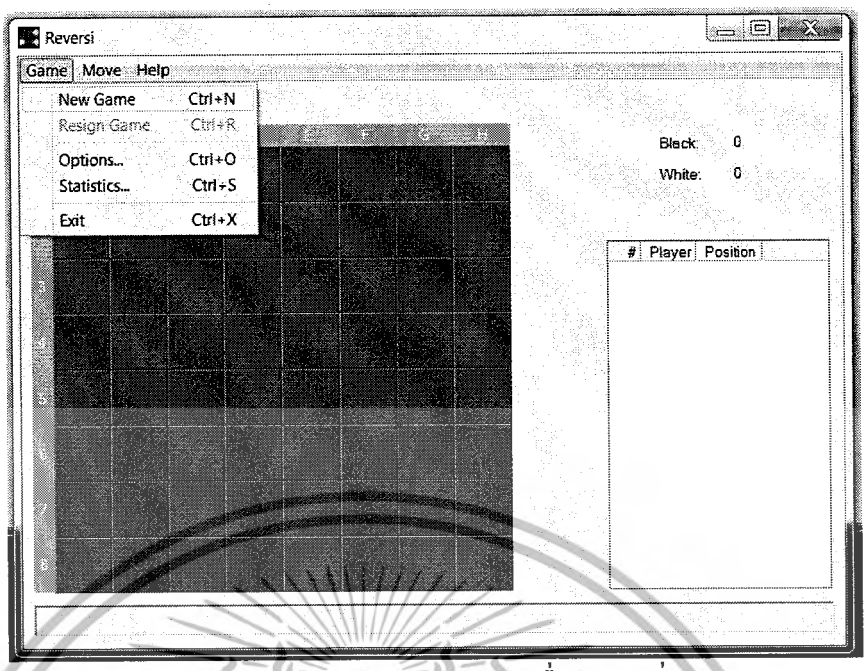
- Undo Move ใช้ในการย้อนกลับตาเดินที่ผ่านมา
- Redo Move ใช้ในการกลับไปตาเดินข้างหน้า
- Undo All Moves ใช้ในการย้อนกลับการเดินทั้งหมดของเกม
- Redo All Moves ใช้ในการกลับไปตาเดินข้างหน้าล่าสุด
- Resume Play กลับเข้าสู่การเล่นอีกครั้ง

3) เมนู Help ประกอบด้วย

- Help Topics ใช้ในการดูข้อมูลต่างๆที่เกี่ยวข้องกับเกม
- About แสดงชื่อผู้จัดทำ

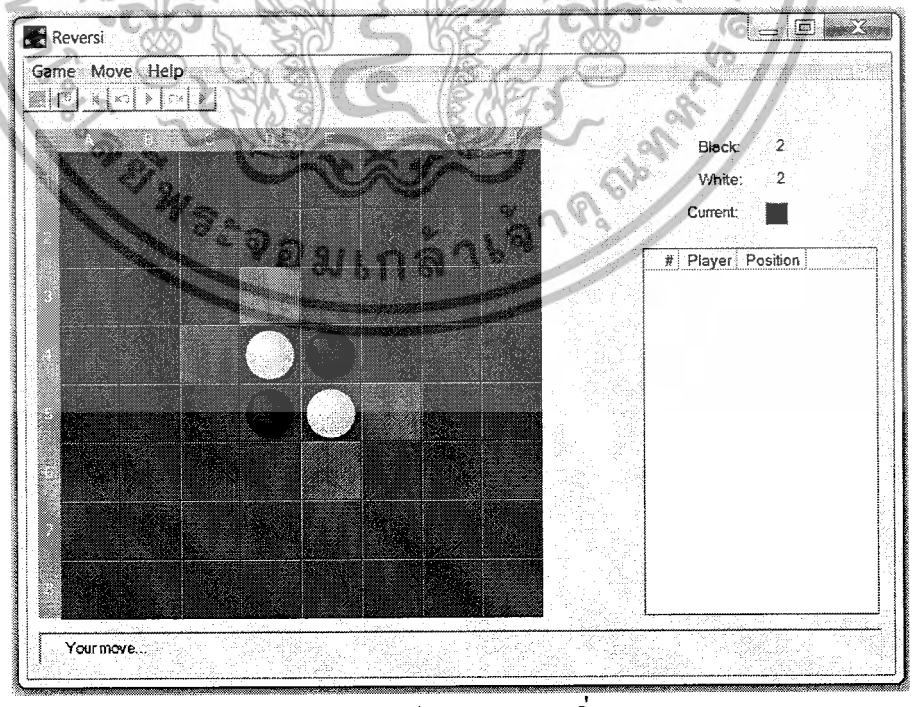
ก.1.2 การใช้งานโปรแกรม

1) เมื่อทำการเปิดโปรแกรมขึ้นมาให้ทำการเลือกเมนู Game และเลือกเมนูย่อย New Game เพื่อทำการเริ่มเกมดังรูป ก.1



รูป ก.1 เลือกเมนูย่อย New Game เพื่อทำการเริ่มเกม

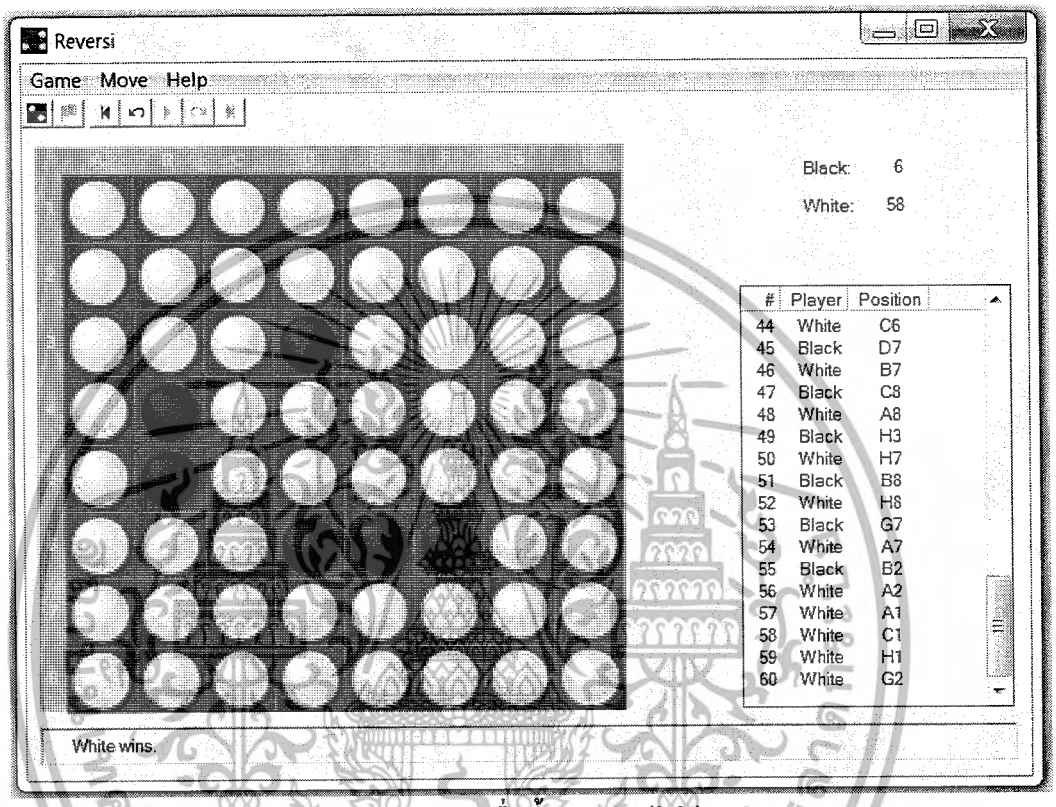
2) เมื่อเริ่มเกมแล้วภายในกระดานจะวางหมากฝ่ายละ 2 ตัว และทางแถบขวาของโปรแกรมจะแสดงสถานะของเกมว่าแต่ละฝ่ายมีจำนวนหมากอยู่กี่ตัว และขณะนี้เป็นตาของฝ่ายไหนเป็นผู้เล่น ถัดมาจะเป็นหน้าต่างแสดงการวางหมากของแต่ละฝ่ายที่ได้ทำการวางลงทั้งหมด ดังรูป ก.2



รูป ก.2 กระดานเริ่มเกม

3) การวางหมากทำได้โดยใช้เมาส์ชี้ในช่องที่สามารถลงได้ โดยช่องที่สามารถลงได้จะแสดงด้วยสีที่ได้ทำการตั้งค่าเอาไว้ในเมนูย่อย Options

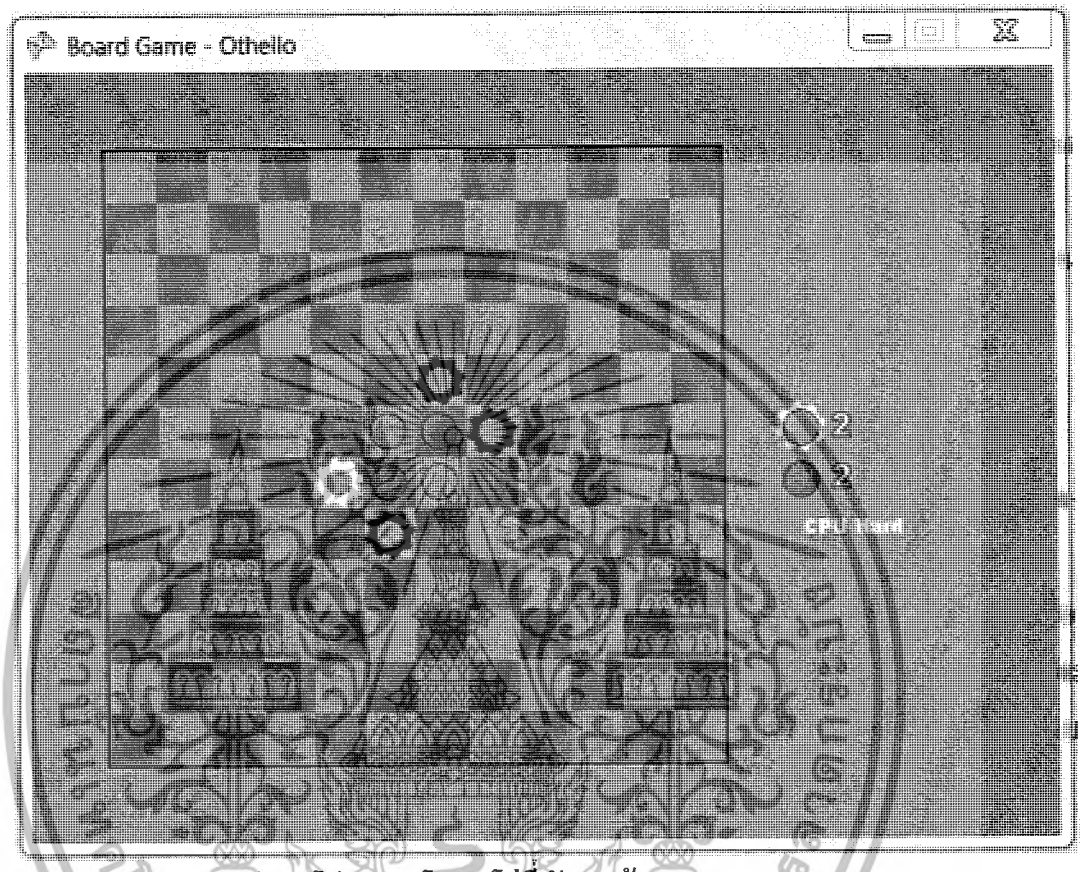
4) เกมจะจบลงเมื่อทั้งสองฝ่ายไม่มีตาเดิน โดยฝ่ายที่มีจำนวนหมากบนกระดานมากกว่าจะเป็นฝ่ายชนะ หรือเสมอกันหากมีจำนวนเท่ากัน



รูป ก.3 เกมจบลงเมื่อทั้งสองฝ่ายไม่มีตาเดิน

ก.2 การใช้งานโปรแกรมโอเทลโล่ที่พัฒนาด้วย XNA Studio

เมื่อทำการเปิดโปรแกรมขึ้นมาก็จะเป็นการเริ่มเกมทันที โดยบนกระดานจะแสดงช่องที่สามารถวางหมากได้โดยใช้เฟืองที่กำลังหมุนอยู่ดังรูป ก.4



รูป ก.4 โปรแกรมโอเทลโล่ที่พัฒนาด้วย XNA Studio

ฝ่ายผู้เล่นจะเป็นฝ่ายเริ่มต้นก่อนเสมอ สามารถเลือกช่องลงบนกระดานโดยการกดคีย์ <- หรือ -> บนคีย์บอร์ด และกด spacebar เพื่อวางหมาก ในแถบด้านขวาจะแสดงจำนวนหมากบนกระดานของแต่ละฝ่ายและถัดมาจะเป็นระดับของคอมพิวเตอร์ซึ่งจะสามารถปรับได้โดยการกด Page Up หรือ Page Down บนคีย์บอร์ด

ภาคผนวก ข

XNA Game Studio Express

ข.1 ทฤษฎีและความรู้พื้นฐานเกี่ยวข้องกับ XNA

Microsoft XNA Game Studio Express เป็นชุดเครื่องมือที่ใช้ในการทำเกมโดยทำงานภายใต้ Microsoft Visual C# 2005 Express Edition. XNA Game Studio Express ประกอบด้วย Xna Framework ที่เป็นตัวคอยจัดการกับ libraries ที่อยู่บน Microsoft .NET Framework 2.0 ซึ่ง Xna Framework ถูกออกแบบมาเพื่อการพัฒนาเกมโดยเฉพาะโดยมีรายละเอียดที่สำคัญดังต่อไปนี้

ข.1.1 ความหมายและวัตถุประสงค์ของ XNA

XNA ย่อมาจาก Xbox/DirectX New generation Architecture เป็นชุดคำสั่งที่ใช้พัฒนาเกมจากบริษัทไมโครซอฟต์ XNA เป็นเครื่องมือสำหรับพัฒนาเกมที่ต่างจาก DirectX เนื่องจาก XNA ไม่ได้เป็นframework เหมือนกับ DirectX แต่ยังรวมเอาเครื่องมือสำหรับพัฒนาเกมเข้าไว้ด้วย ซึ่งรวมถึง IDE (Integrated Development Environment) ที่พัฒนามาจาก Microsoft Visual Studio ด้วย เครื่องมือเหล่านี้ทำให้การพัฒนาเกมสะดวกขึ้นมากโดยมีวัตถุประสงค์ดังนี้

1) Cross-platform Game Development Tools คือเกมที่สร้างขึ้นบน XNA จะสามารถเล่นได้ทั้งใน Windows และ Xbox ไมโครซอฟต์ต้องการให้นักพัฒนาเกมสามารถสร้างเกมใน Windows ได้อย่างสะดวกและรวดเร็วรวมทั้งใช้ต้นทุนต่ำในการ พัฒนาและยังสามารถพัฒนาเป็นเวอร์ชั่นสำหรับ Xbox ได้ โดยที่โค้ดจะมีความเหมือนกันประมาณ 95% ส่วนที่ต่างกันก็คือการใช้ features ที่เฉพาะเจาะจงสำหรับแพลตฟอร์มนั้นๆ ดังนั้นเกมที่พัฒนาส่วนใหญ่จะไม่มี code ที่แตกต่างกัน

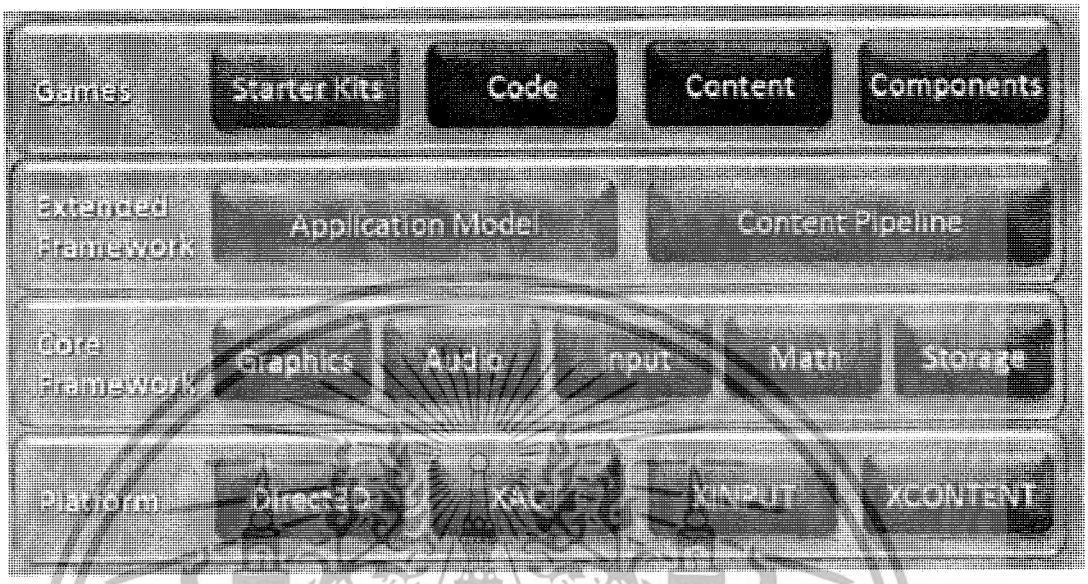
2) Simplify Game Development คือทำให้การพัฒนาเกมสามารถทำได้ง่ายโดยจะมีการจัดการด้านทรัพยากรให้ทำได้ง่ายและสะดวกซึ่งมีเครื่องมือที่มีชื่อเรียกว่า "Content Pipeline" ที่ทำให้การส่ง content เข้าไปในเกมทำได้ง่ายทั้งการ import, compiling, และ loading ในเกม

3) มี "Starter Kits" ให้คือเป็นเกมที่สมบูรณ์ทั้ง resource และโค้ดที่ใช้เป็น project template ได้ทำให้นักพัฒนาเกมสามารถเริ่มสร้างเกมโดยการปรับแต่ง template ดังกล่าวได้ทันที

ข.1.2 องค์ประกอบของ XNA

XNA ประกอบด้วย 2 ส่วนหลัก คือ XNA Game Studio Express ซึ่งเป็น IDE ที่ประกอบไปด้วยเครื่องมือที่ใช้ในการพัฒนาเกมทั้งหมดและ XNA Framework ที่รวมเอาไลบรารี่ที่ใช้ในการ

พัฒนาเกมซึ่งเป็น dll ที่เขียนด้วยภาษา C# ทำให้การทำงานของ XNA ต้องทำงานภายใต้ภาษา C# โดยมีการแบ่งเป็นเลเยอร์ต่างๆดังรูป ข.1



รูป ข.1 เลเยอร์ของ XNA Framework

ข.1.3 หน้าที่ของเลเยอร์หลักใน XNA Framework

โดยมีการแบ่งเลเยอร์ต่างๆออกเป็น 4 เลเยอร์ที่เป็นอิสระต่อกันทำให้การพัฒนาเกมสามารถทำได้ง่ายขึ้น โดยแต่ละเลเยอร์ มีหน้าที่ดังต่อไปนี้

1) Platform เป็นชั้นล่างสุดของโครงสร้างของ XNA เป็น low-level native and managed API สำหรับ XNA framework ทั้งหมดจะเรียกใช้ส่วนนี้ให้ทำหน้าที่ในการติดต่อกับฮาร์ดแวร์หรือตัวกลาง เช่น Direct3D, XACT, XInput และ XContent

2) Core Framework เป็นส่วนที่ทำหน้าที่หลัก (core functionality) เช่น Managed DirectX ส่วนนี้แบ่งการทำงานเป็นกลุ่ม คือ Graphics, Audio, Input, Math และ Storage ส่วนของ XNA อื่นๆคือการขยายการทำงานจากส่วนนี้

3) Extended Framework เป็นส่วนที่ช่วยอำนวยความสะดวกสำหรับการพัฒนาเกม ในเวอร์ชั่นแรกของ XNA ส่วนนี้ประกอบด้วย Application Model และ Content Pipeline

4) Game คือตัวเกมซึ่งประกอบด้วยส่วนของโค้ดและ content รวมทั้ง starter kits, templates และ game component

ข.1.4 หน้าที่ของส่วนต่างๆภายในเลเยอร์ของ XNA Framework

1) Application Model เป็นส่วนที่ทำให้นักพัฒนาเกมไม่ต้องกังวลถึงแพลตฟอร์มที่พัฒนาส่วนนี้จะทำหน้าที่จัดการเรื่อง window, message pump, timer or clock, รวมทั้งการจัดการ Graphics Component และ Graphics Device นักพัฒนาจึงไม่ต้องสนใจความแตกต่างของระบบระหว่าง Windows และ XBox และสามารถพัฒนาเกมขึ้นจาก Game Components ได้ด้วย (เรียกส่วนนี้ว่า Componet Model) ซึ่งทำให้การพัฒนาเกมทำได้ง่าย

2) Graphics XNA พัฒนาส่วนกราฟฟิกจาก DirectX 9.0 โดยตัดเอาส่วนของ Fixed-function pipeline ออกเพื่อต้องการให้เกมใน Windows เป็นแบบ full compaitible กับ Xbox 360 ซึ่งมีแต่ shader-driven programmable pipeline ให้ได้ใช้ XNA ต้องการทำให้การใช้งานในส่วนของ shaders และ effects สามารถทำได้โดยง่ายขึ้นซึ่งมี Basic Effect ที่เป็นการตัวจัดการ

3) Audio ด้านของเสียง XNA สร้างขึ้นจาก XACT ซึ่งเป็น cross-platform audio API สำหรับ Windows กับ Xbox ของไมโครซอฟต์เอง XACT ใช้หลักการคล้ายกับ shaders ใน Direct3D คือ นักพัฒนาเสียงจะสร้างเสียงไว้เป็น package ซึ่งมีรายละเอียดบรรจุครบถ้วน เช่น volume, looping, channel mixing แล้วโปรแกรมเมอร์เรียกใช้จากชื่อ package โดยไม่ต้องคำนึงถึงรายละเอียดด้านฮาร์ดแวร์

4) Input ส่วนนี้สร้างขึ้นจาก XInput ซึ่งก็เป็น cross-platofrm API สำหรับ common controller โดยทำงานแบบ immediate mode API ไม่ต้องมีการ initiialization ทำให้ไม่ต้องข้องเกี่ยวกับรายละเอียดการคุมฮาร์ดแวร์ โดยเรียกคำสั่ง GetState จาก GamePad, Keyboard และ Mouse ก็สามารถทำงานได้ทันที

5) Storage เป็นส่วนสำหรับเก็บข้อมูลเกม (save games, high scores etc) ใน Xbox 360 จะเก็บสถานะของเกม (Game state) ไว้กับ profile และ storage device (hard drive หรือ memory unit) XNA มีส่วนนี้ emulate ให้ทำงานบน Windows เหมือนกับ Xbox จึงสามารถใช้โค้ดร่วมกันได้

6) Math XNA มีความสามารถทางคณิตศาสตร์ที่จำเป็นสำหรับเกม เช่น Vector2, Vector3, Vector4, Matrix,Plane,Ray เป็นต้น รวมถึง bounding volume เช่น BoundingBox, BoundingSphere และ BoundingFrustum ซึ่งสามารถคำนวณ intersection และ containment test ได้ โดย Math ทั้งหมดเป็น right-handed เพื่อให้ไม่ต้องทำ function ให้เป็นสองชุดจึงสะดวกต่อการสร้าง content ตลอดจนการใช้ middleware แต่ก็สามารถเรียกใช้ Left-handded API ได้ในกรณีที่ต้องการ

ข.1.5 หน้าที่ของ Content Pipeline

เป็นส่วนที่สำคัญมากใน XNA ซึ่งช่วยในการจัดการทรัพยกรต่างๆได้ง่ายขึ้นโดยมีหน้าที่สำคัญดังนี้การ import,compile และ load game asset ซึ่งได้แก่ textures, 3D models, shaders และ sounds

Content Pipeline ประกอบด้วย dll หลายตัว ดังนี้

1) Microsoft.Xna.Framework.Content.Pipeline.dll ฟังก์ชั้นพื้นฐานสำหรับ content pipeline
2) Microsoft.Xna.Framework.Content.Pipeline.EffectImporter.dll ใช้สำหรับ import และ compile shaders
3) Microsoft.Xna.Framework,Content.Pipeline.FBXImporter.dll ใช้สำหรับ import และ compile Filmbox (.fbx) 3D model รวมทั้ง skining และ bones
4) Microsoft.Xna.Framework.Content.Pipeline.TextureImporter.dll ใช้สำหรับ import texture files ในรูป DirectX format, .png, .jpg, .bmp และ .tga รวมทั้ง 2D sprites
5) Microsoft.Xna.Framework.Content.Pipeline.XImporter.dll ใช้สำหรับ import และ compile .X 3D model dll เหล่านี้จะไม่ถูกใช้ในการ run เกม แต่จะใช้สำหรับ build และ compile เท่านั้น content ที่ได้จากการ compile จะอยู่ในรูป .xnb (XNA binary) ซึ่งทำให้การ distribute เกมทำได้สะดวก ทำให้ได้รูปแบบและความครบถ้วนของ content เนื่องจากทุกอย่างจะอยู่ใน .xnb ทั้งหมดและแปลงเป็นรูปแบบที่ใช้ได้ทันที การใช้ content pipeline compile content เป็น .xnb จะสามารถ run ใน Xbox 360 ได้ เนื่องจากXboxจะ load ได้เฉพาะ content ใน .xbn เพียงอย่างเดียว
6) เมธอดที่สำคัญใน XNA Game Class เกมที่พัฒนาโดยใช้ XNA จะมี class หลักคือ Game Class ซึ่งประกอบด้วย game engine, graphic device และ window setting รวมทั้งการเรียกใช้ input และ sound Game Class นี้มี method ที่สำคัญดังนี้
   6.1) initialize() กำหนดค่าเริ่มต้นของเกม และโหลด game content
   6.2) LoadGraphicsContent() ใช้สำหรับโหลด graphics content ทั้งหมด
   6.3) UnloadGraphicsContent() ใช้สำหรับยกเลิกการโหลด graphcis content
   6.4) Update() ถูกเรียกก่อนแสดงผลเพื่อให้เกม update สถานะต่างๆ ซึ่งหาก Graphics Hardware ทำงานช้า Update จะถูกเรียกใช้มากกว่า Draw เพื่อให้การดำเนินเกมเป็นไปอย่างราบรื่นแม้จะแสดงผลไม่ทันก็ตาม
   6.5) Draw() แสดงผลของเกมทางจอภาพ

ข.2 ขั้นตอนการทำงานของ XNA Framework

ขั้นตอนการทำงาน ของ XNA พอสรุปได้ดังนี้

Game1() - ประกาศตัวแปร อยู่ที่ไฟล์ Game1.cs

Initialize() - การเขียน ชื่อย่อ ,ไตเติ้ล เกม อยู่ที่ไฟล์ Game1.cs

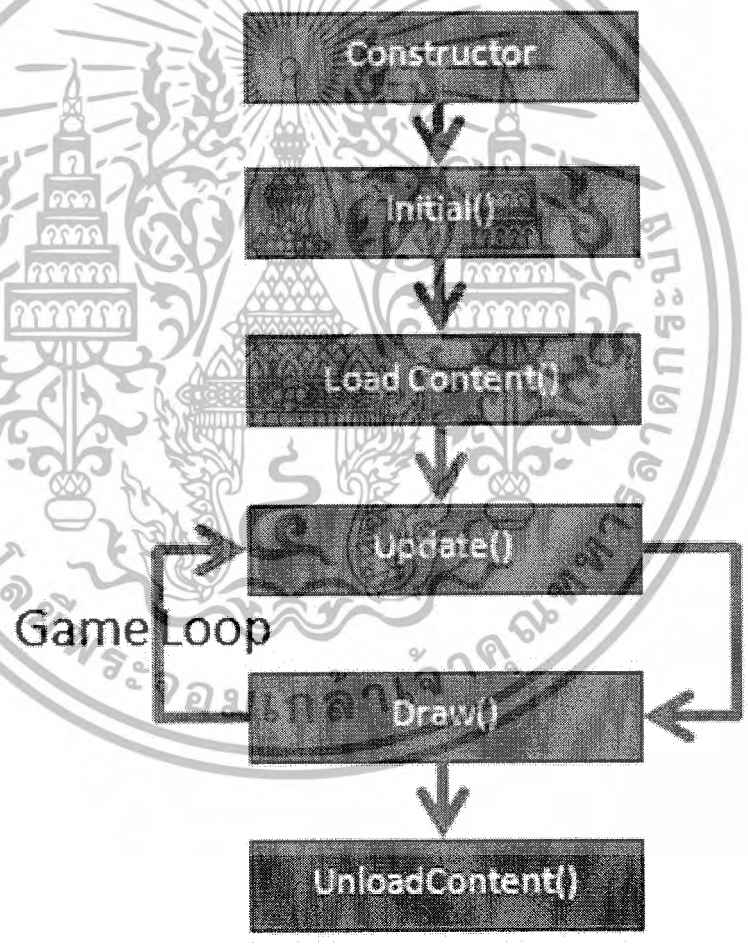
LoadContent() - โหลดไฟล์รูปภาพ เสียง โมเดลต่างๆจากโฟลเดอร์ Content อยู่ที่ Game1.cs

Run() - ทุกครั้งที่เกมเริ่มทำงานและวนลูป อยู่ที่ไฟล์ Program.cs

Update() - ตรวจสอบ input ของผู้ใช้ คำนวณ และ ให้เกมแสดงผล อยู่ที่ไฟล์ Game1.cs

Draw() - วาดรูปต่างๆ อยู่ที่ไฟล์ Game1.cs

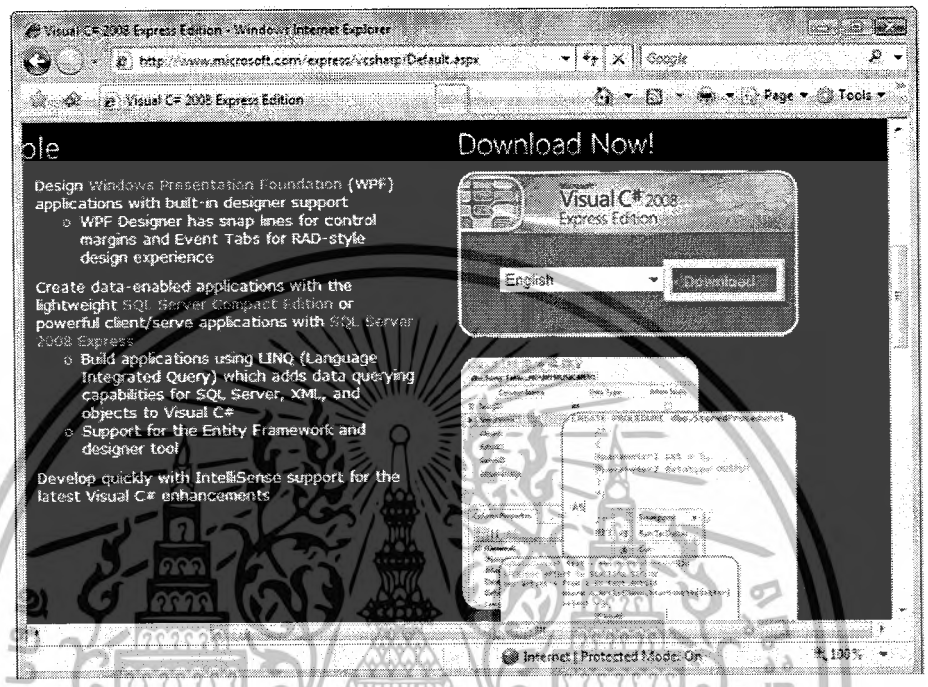
UnloadContent() - คืนทรัพยากร กราฟิก Game1.cs



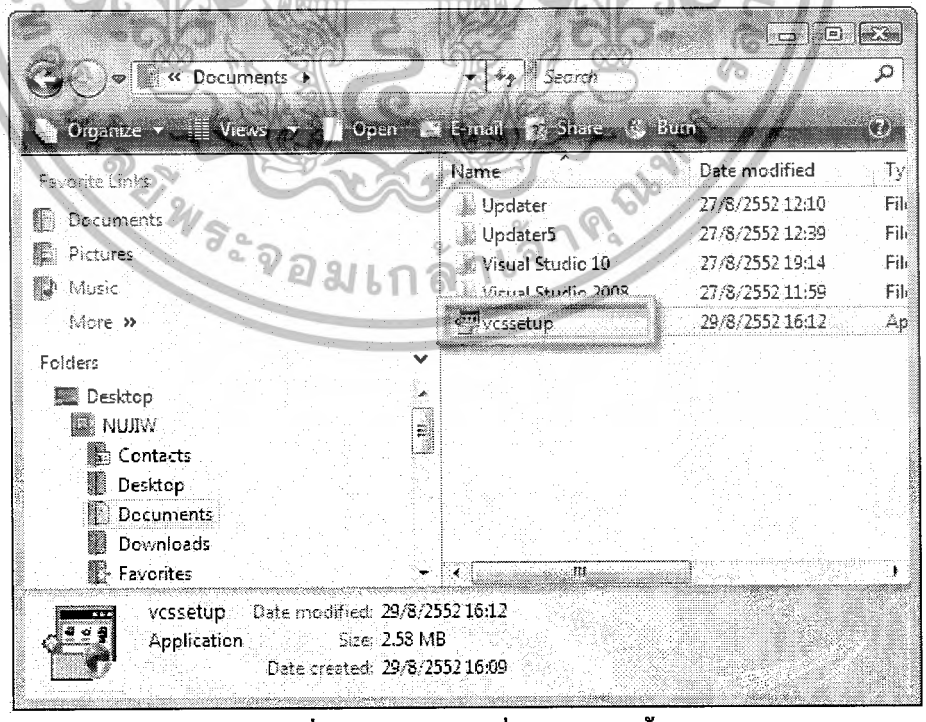
รูป ข.2 ขั้นตอนการทำงานของ XNA Framework

ข.3 การติดตั้ง Microsoft Visual C#

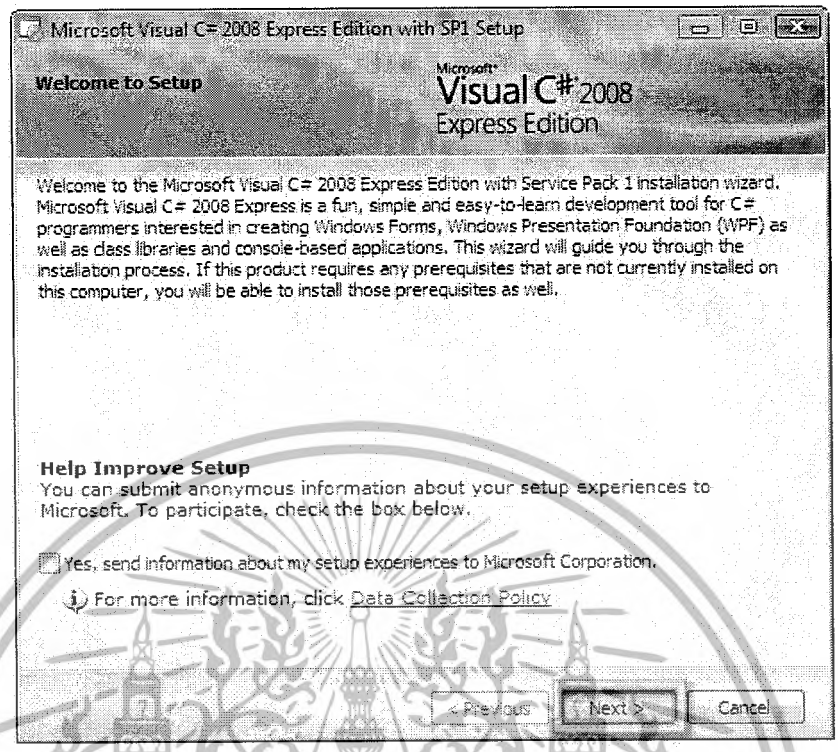
ดาวน์โหลดได้ที่ URL : http://www.microsoft.com/express/vcsharp/ การติดตั้งก็จะเหมือนกับการติดตั้งโปรแกรมทั่วๆไป



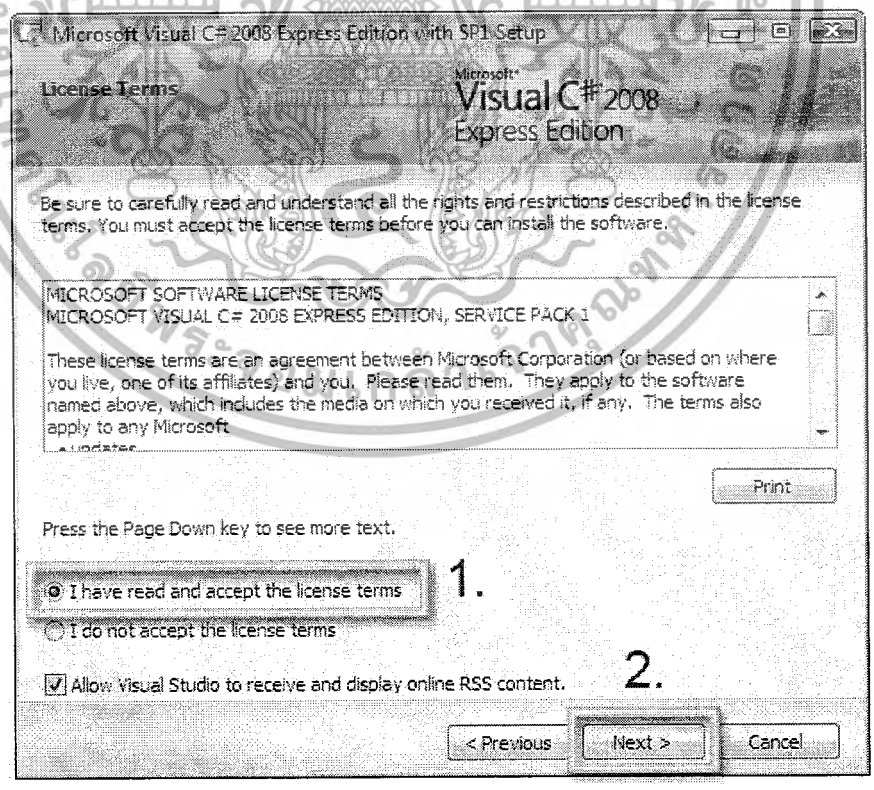
รูป ข.3 ทำการ Download Visual C#



รูป ข.4 คลิกที่ไฟล์ vcssetup เพื่อทำการติดตั้งโปรแกรม

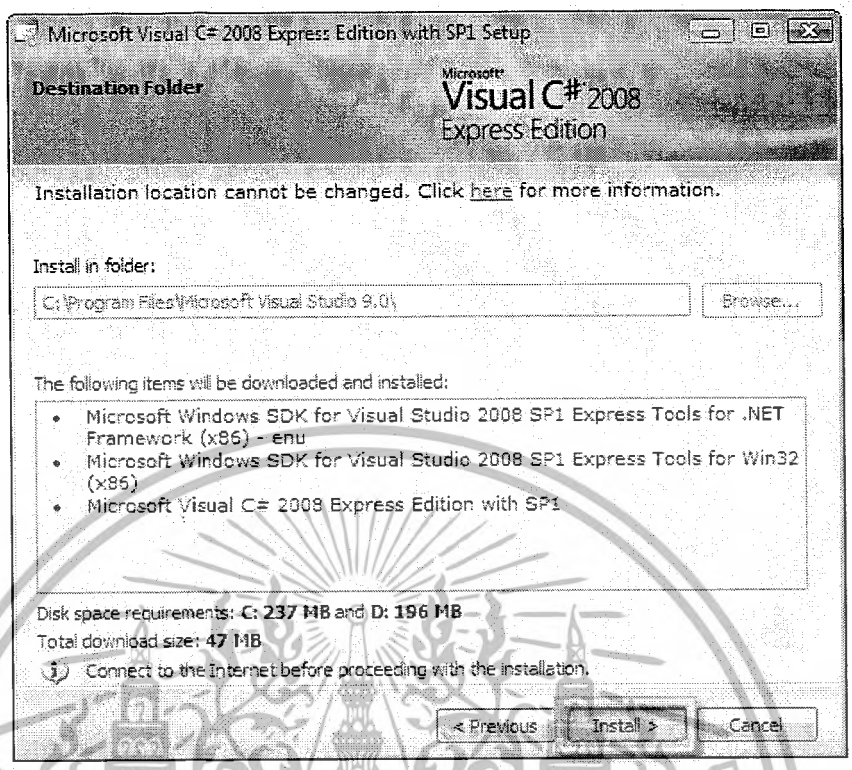


รูป ข.5 คลิกที่ [Next >] เพื่อดำเนินการต่อไป

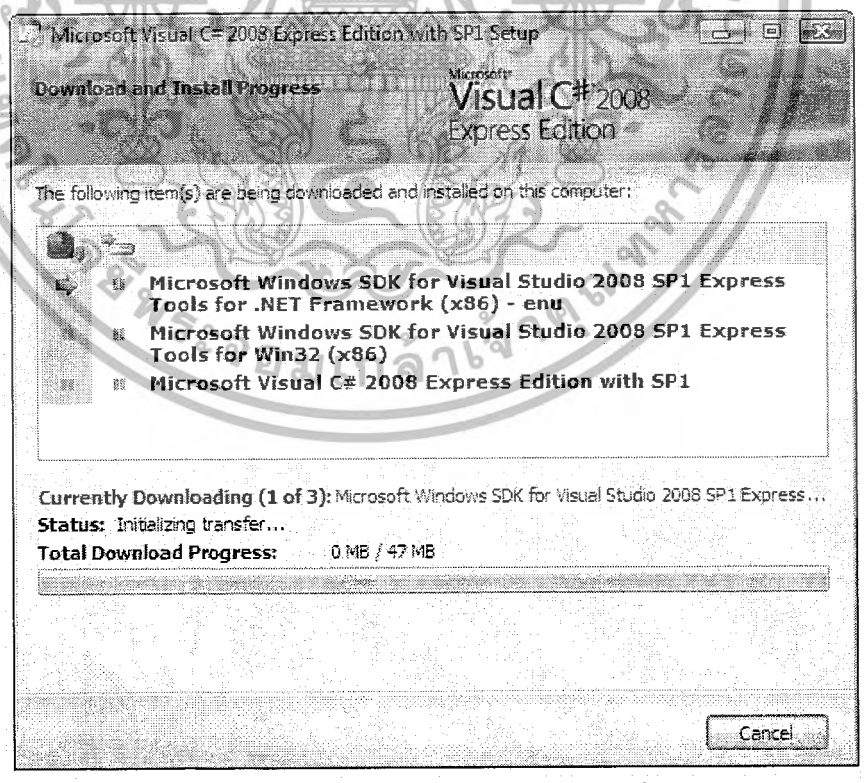


รูป ข.6 คลิกเลือก I have read and accept the license.

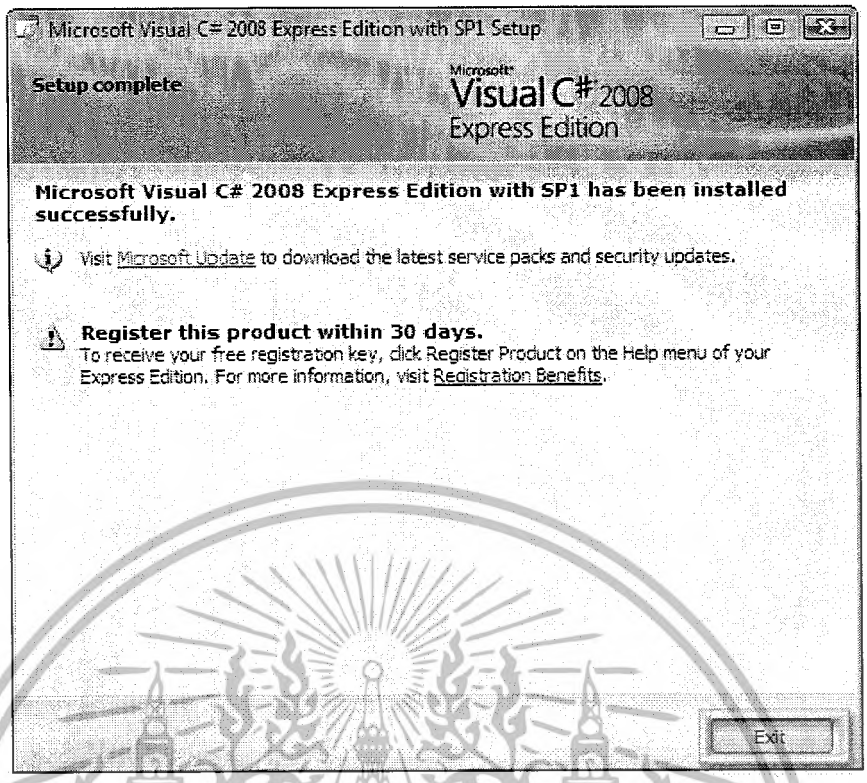
แล้วจึง คลิก [Next >]



รูป ข.7 คลิกที่ [Install >] คอมพิวเตอร์ต้องเชื่อมต่อกับอินเตอร์เน็ท



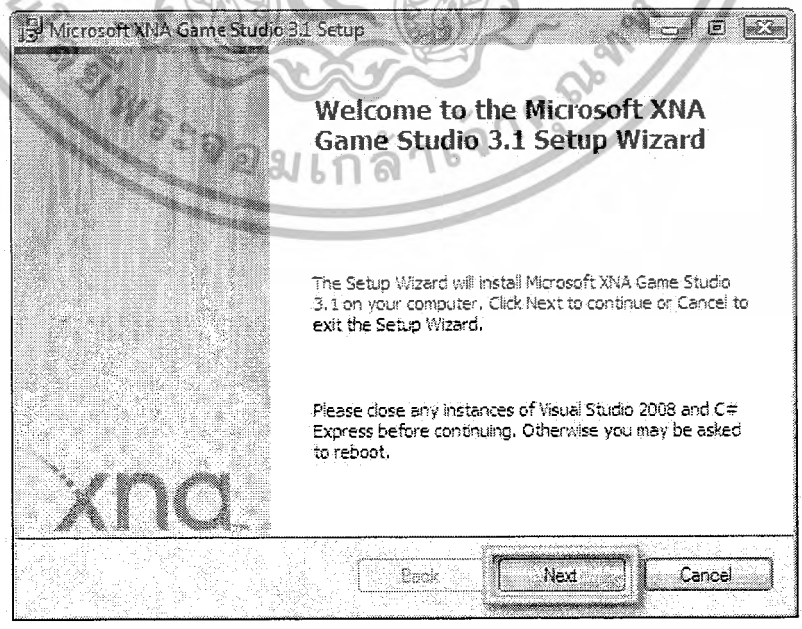
รูป ข.8 ดาวน์โหลดไฟล์จากไมโครซอฟท์เพิ่มเติม



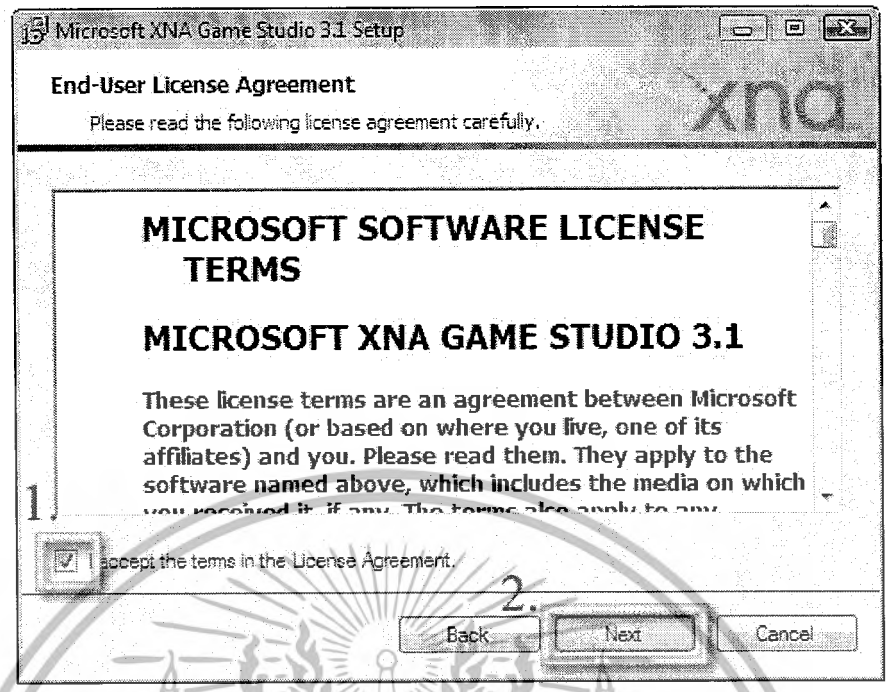
รูป ข.9 คลิกที่ [Exit] เพื่อจบขั้นตอนการติดตั้งโปรแกรม

ข.4 การติดตั้ง XNA Studio

คลิกที่ไฟล์ XNAGS31_ssetup เพื่อทำการติดตั้งโปรแกรม (ขณะติดตั้งต้องปิดโปรแกรม Visual C# ก่อน)



รูป ข.10 คลิกที่ [Next >] เพื่อดำเนินการต่อไป



รูป 2.11 คลิกเลือก I accept the terms in License Agreement. แล้วจึง คลิก [Next >]

รูป 2.12 คลิกที่ [Finish] เพื่อจบขั้นตอนการติดตั้งโปรแกรม