# การจำลอง Reinforcement Learning ในการค้นหาเส้นทางในระบบเครือข่าย

# Reinforcement Learning Simulation for Network Routing

โดย

นางสาวจิตราภรณ์ บุญลักษณานุสรณ์

รหัส 42067111



อาจารย์ที่ปรึกษา

ดร. โชติพัชร์ ภรณวลัย



รายงานนี้เป็นส่วนหนึ่งของวิชาโครงการพัฒนาระบบงาน

หลักสูตรวิทยาศาสตรมหาบัณฑิต สาขาวิชาเทคโนโลยีสารสนเทศ

ภาคเรียนที่ 1 ปีการศึกษา 2544

คณะเทคโนโลยีสารสนเทศ

สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง

| | |
|---|---|
| **ชื่อหัวข้อ** | การจำลอง Reinforcement Learning ในการค้นหาเส้นทางในระบบเครือข่าย |
| **นักศึกษา** | นางสาวจิตราภรณ์ บุญลักษณานุสรณ์ |
| **อาจารย์ที่ปรึกษา** | ดร. โชติพัชร์ ภรณวลัย |
| **ระดับการศึกษา** | วิทยาศาสตรมหาบัณฑิต สาขาวิชาเทคโนโลยีสารสนเทศ |
| **แขนงวิชา** | วิทยาการสารสนเทศ |
| **ปีการศึกษา** | 2544 |

## บทคัดย่อ


เนื่องจากในปัจจุบันระบบเครือข่ายมีประสิทธิภาพสูง และความซับซ้อนมากขึ้น จึงได้มีการนำเอาวิธีการทาง Artificial Intelligent ที่มีประสิทธิภาพต่างๆ เข้ามาแก้ปัญหาดังกล่าว เช่น Heuristic, Neural network, Genetic algorithm เป็นต้น

โครงการนี้จะเป็นการศึกษาทฤษฎีของ Reinforcement Learning ซึ่งเป็น Artificial Intelligent แขนงหนึ่งที่ใช้ค้นหาคำตอบจากการเรียนรู้จากสภาพแวดล้อมโดยตรง จากนั้นนำทฤษฎีของ Reinforcement Learning แบบ Q-routing มาประยุกต์ใช้งานในการค้นหาเส้นทางในระบบเครือข่ายที่มีการเปลี่ยนแปลงแบบไดนามิก

| | |
|---|---|
| **Title** | Reinforcement Learning Simulation for Network Routing |
| **Student** | Miss. Jittraporn Bunlaksananusorn |
| **Advisor** | Dr. Chotipat Pornavalai |
| **Level of Study** | Master of Science in Information Technology |
| **Major** | Information Science |
| **Academic Year** | 2001 |

## ABSTRACT

Nowsaday, Computer network system is effective and sophisticated.So there are a lot of Artificial Intelligence tactic to solve computer network problem such as Heuristic, Neural Network, Genetic Algorithm and so on.

This project describes Reinforcement Learning which is the problem faced by an agent that must learn behavior through trial-and-error interactions with dynamic environment.Then applied Q-routing algorithm to solve packet routing in dynamically changing networks.

# กิตติกรรมประกาศ

โครงการพัฒนาแบบจำลองการค้นหาเส้นทางในระบบเครือข่ายโดยใช้ Reinforcement Learning นี้ สำเร็จลุล่วงด้วยดี เนื่องจากได้รับคำแนะนำจาก ดร.โชติพัชร์ ภรณวลัย อาจารย์ที่ปรึกษา ซึ่งกรุณาให้ข้อคิดเห็นต่างๆ เพื่อเป็นแนวทางในการศึกษาและดำเนินการให้เป็นไปอย่างต่อเนื่อง เพื่อให้โครงการนี้สำเร็จตามวัตถุประสงค์ที่ได้ตั้งไว้

นอกจากนี้ต้องขอขอบคุณแรงบันดาลใจจากพ่อ และกำลังใจที่มีให้เสมอจากแม่ อีกทั้งเพื่อนๆที่คอยให้กำลังใจ และคอยให้คำปรึกษาในการเขียนโปรแกรม ซึ่งเป็นประโยชน์ต่อการพัฒนาโครงการและทำให้โครงการนี้ได้รับความสำเร็จเป็นอย่างดี

จิตราภรณ์ บุญลักษณานุสรณ์

# สารบัญ

หน้า

# สารบัญตาราง

หน้า

# สารบัญภาพ

หน้า

หน้า

ภาพที่

# บทที่ 1
# บทนำ

ปัจจุบันได้มีการนำเอาแนวความคิดทางด้าน Artificial Intelligence ได้รับความสนใจที่จะนำไปใช้ในโครงงานด้านต่างๆ เช่น นำไปประยุกต์ใช้กับการจดจำรูปแบบ (pattern recognition) การประมวลผลสัญญาณภาพ (image processing) ควบคุมการเคลื่อนไหวของหุ่นยนต์ (robot movement control) และอื่นๆ ซึ่ง Reinforcement Learning เป็นศาสตร์แขนงหนึ่งของ Artificial Intelligence ที่เป็นที่นิยมในปัจจุบัน ซึ่งจะเป็นการค้นหาคำตอบจากการเรียนรู้จากสภาพแวดล้อมโดยตรง ดังนั้นจึงเล็งเห็นประโยชน์ที่จะทำการศึกษาทฤษฎีและอัลกอริธึมต่างๆ ของ Reinforcement Learning เพื่อประยุกต์ใช้งานในการช่วยหาเส้นทางในระบบเครือข่าย โดยสร้างเป็นแบบจำลองในการค้นหาเส้นทางในระบบเครือข่ายที่สภาวะโหลดต่างๆ

## 1.1 วัตถุประสงค์

- เพื่อศึกษาและทำความเข้าใจเกี่ยวกับทฤษฎีของ Reinforcement Learning
- เผยแพร่แนวคิดและศาสตร์ทางด้าน Reinforcement Learning ซึ่งเป็นศาสตร์ที่ค่อนข้างเป็นที่รู้จักกันน้อยในปัจจุบัน
- สร้างโปรแกรมจำลองการทำงานของ Reinforcement Learning ในการค้นหาเส้นทางในระบบเครือข่ายโดยเปรียบเทียบกับทฤษฎีการค้นหาเส้นทางที่สั้นที่สุดของ Bellman Ford

## 1.2 ขั้นตอนในการศึกษา

- ศึกษาทฤษฎีและความรู้พื้นฐานเกี่ยวกับ Reinforcement Learning
- ศึกษาทฤษฎีของ Bellman Ford ในการค้นหาเส้นทางที่สั้นที่สุด
- นำทฤษฎีของ Reinforcement Learning มาประยุกต์ใช้งานเพื่อแก้ปัญหาการค้นหาเส้นทางในระบบเครือข่าย
- เขียนโปรแกรมจำลองการทำงานของ Reinforcement Learning ในการค้นหาเส้นทางในระบบเครือข่ายโดยเปรียบเทียบกับทฤษฎีของ Bellman Ford

- ทำการทดสอบโปรแกรมจำลอง พร้อมทั้งเปรียบเทียบผลลัพธ์ที่ได้จากการเรียนรู้แบบ Reinforcement Learning และจากการใช้ทฤษฎีของ Bellman Ford

**1.3 ประโยชน์ที่คาดว่าจะได้รับจากโครงการ**

- โปรแกรมจำลองจะทำให้สามารถเข้าใจหลักการทำงานของ Reinforcement Learning ได้ดียิ่งขึ้น
- สามารถเห็นประโยชน์ของการนำ Reinforcement Learning มาใช้ในการค้นหาเส้นทางของระบบเครือข่ายที่สภาวะโหลดต่างๆได้
- สามารถนำหลักการของ Reinforcement Learning ไปประยุกต์ใช้งานในด้านต่างๆได้

**1.4 สรุปเนื้อหาโดยสังเขป**

ในเอกสารเล่มนี้กล่าวถึง

บทที่ 1 กล่าวถึงวัตถุประสงค์ ขั้นตอนในการศึกษา และประโยชน์ที่คาดว่าจะได้รับ

บทที่ 2 กล่าวถึงความรู้พื้นฐานของ Reinforcement Learning และทฤษฎีที่เกี่ยวข้อง

บทที่ 3 กล่าวถึงการนำ Reinforcement Learning ไปประยุกต์ใช้งานในการค้นหาเส้นทางในระบบเครือข่าย

บทที่ 4 กล่าวถึงการใช้งานโปรแกรมจำลอง และผลที่ได้จากการทดลอง

บทที่ 5 กล่าวถึงบทสรุป และข้อเสนอแนะ

# บทที่ 2
# ความรู้พื้นฐานเกี่ยวกับ Reinforcement Learning

Reinforcement Learning (RL) เป็นแนวความคิดหนึ่งทางด้าน Artificial Intelligence ที่ใช้ค้นหาคำตอบจากการเรียนรู้จากสภาพแวดล้อมโดยตรง ซึ่งจะเป็นการเรียนรู้ว่า Agent ทำหน้าที่อะไร และเลือกกระทำแอ็คชั่น (action) ตามสภาวะ (state) ของสภาพแวดล้อม(environment)ที่ได้รับ จากนั้นจะได้รับสิ่งตอบแทนที่ได้จากการกระทำ actionนั้นๆ เรียกว่า "reward" โดยมีจุดมุ่งหมายที่จะพยายามให้ได้รับค่า reward ให้ได้มากที่สุด

## 2.1 องค์ประกอบของ Reinforcement Learning

Reinforcement Learning มีองค์ประกอบที่สำคัญ 4 องค์ประกอบ ได้แก่

2.1.1 Policy เป็นนโยบายเพื่อใช้กำหนดว่าให้ Agent ทำหน้าที่อะไร

2.1.2 Reward เป็นสิ่งตอบแทนที่ได้จากการกระทำ action

2.1.3 Value เป็นการบอกว่าสิ่งใดดีหรือไม่ดีสำหรับ Agent

2.1.4 Model เป็นโมเดลที่ใช้เลียนแบบพฤติกรรมของสภาพแวดล้อม เพื่อช่วยในการวางแผนการทำงานของ Agent

## 2.2 ลักษณะการเรียนรู้

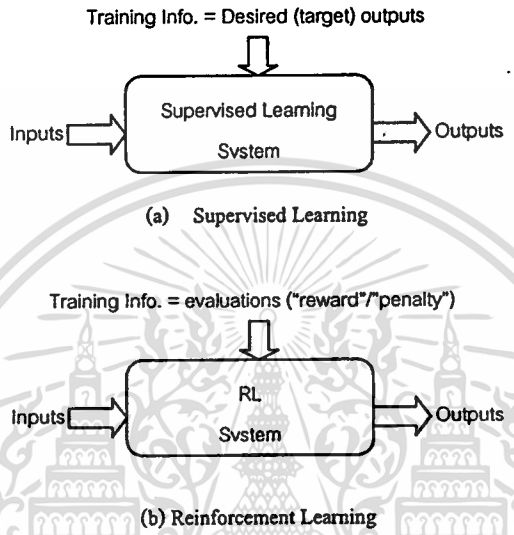
ลักษณะการเรียนรู้สามารถแบ่งได้เป็น 3 ประเภท คือ

2.2.1 การเรียนรู้แบบมีผู้สอน (supervised learning) จะสอนด้วยการป้อนแบบข้อมูลอินพุท (input pattern) และแบบข้อมูลเอาต์พุทที่ต้องการ (desired output pattern) หรือเป้าหมาย (target value) ให้เรียนรู้ตามกฎการเรียนรู้ที่มีการวัดค่าผิดพลาด (error) โดยการคำนวณจากผลต่างระหว่างค่าเอาต์พุท (actual output) กับค่าเอาต์พุทที่ต้องการ แล้วนำค่าผิดพลาดนี้ไปใช้ในการคำนวณย้อนกลับ (feedback) เพื่อปรับให้ได้ค่าข้อมูลเอาต์พุทเท่ากับหรือใกล้เคียงกับค่าเอาต์พุทที่ต้องการ ดังรูปที่ 2.1 (a)

2.2.2 การเรียนรู้แบบไม่มีผู้สอน (unsupervised learning) เป็นการสอนด้วยการป้อนแบบข้อมูลอินพุทเข้าไปอย่างต่อเนื่องเพียงอย่างเดียวจะไม่มีการส่งค่าใดๆ ให้คำนวณย้อน

กลับ (no feedback) ซึ่งแบบข้อมูลเข้าทั้งหมดที่ใช้ในการเรียนรู้ไม่สามารถคาดการณ์ล่วงหน้าได้ว่าข้อมูลฝึกจะมีลักษณะใดบ้าง

2.2.3 การเรียนรู้แบบReinforcement Learning แสดงดังรูปที่ 2.1(b) ซึ่งสิ่งที่ต้องการใน RL จะเป็นค่า reward ที่ประเมินได้จากการกระทำ action




**รูปที่ 2.1** ลักษณะการเรียนรู้

คุณสมบัติสำคัญของ RL ที่ต่างจากการเรียนรู้แบบอื่น คือ มีการใช้ training information ที่มีการประเมิน action ที่เลือกมากกว่าการสอนให้ agent เลือก action ที่ถูกต้อง

Feedback สามารถแบ่งออกได้เป็น 2 ประเภท คือ

1) Evaluative feedback เป็นการบอกว่า action ที่เลือกดีอย่างไร (How good) โดยที่ไม่รู้ว่า action ที่เลือกเป็น action ที่ดีที่สุดหรือแย่ที่สุด เป็นวิธีที่ใช้ใน Reinforcement Learning
2) Instructive feedback feedback จาก environment จะเป็นตัวชี้ว่า action ที่ถูกต้องนั้นควรเป็นอะไร โดยที่ไม่ขึ้นกับ action ที่เลือก เป็นวิธีที่ใช้ใน supervised learning

ยกตัวอย่าง เช่น ถ้ามี action ที่เป็นไปได้ 100 action และ Agent ทำการเลือก action หมายเลข 30 evaluative feedback จะทำการประเมินและให้คะแนนเป็น 0.9 สำหรับการเลือก action นั้น ขณะที่ instructive feedback จะระบุว่า action ที่ถูกต้องคือ action หมายเลข 67 ซึ่งต่างจากวิธีการของ evaluative ที่จะต้องมีการเปรียบเทียบกับ action อื่นๆ ด้วย

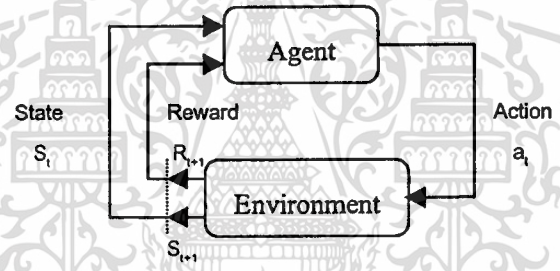
ในการที่ agent จะได้ค่า reward ให้มากที่สุดสามารถทำได้โดยการเลือก action ซึ่งจะแบ่งออกเป็น 2 แบบ คือ

1) Exploitation เป็นการเลือก action ที่เป็น greedy action คือ action ที่มีค่า action value ที่มีค่าสูงสุด
2) Exploration เป็นการเลือก action ที่เป็น non-greedy action ซึ่งจะช่วยในการปรับปรุงการประเมินค่า action value ให้ดีขึ้น

การทำ exploitation เป็นการทำเพื่อให้ได้ค่า reward ที่มีค่าสูงสุดในการ play แต่ละครั้ง แต่ exploration เป็นการทำเพื่อหวัง total reward ในระยะยาว ในระหว่างการ exploration ค่า reward ที่ได้จะมีค่าน้อย แต่จะมีค่ามากขึ้นในระยะยาวภายหลังจากที่ค้นพบ action ที่ดีกว่า

**2.3 หลักการทำงานของ Reinforcement Learning**

จากรูปที่ 2.2 แสดงอินเตอร์เฟสระหว่าง Agent กับสภาพแวดล้อมที่มีการเปลี่ยนแปลงตลอดเวลา




รูปที่ 2.2 อินเตอร์เฟสระหว่าง Agent กับสภาพแวดล้อม

ในแต่ละช่วงเวลา Agent จะได้รับค่า state จากสภาพแวดล้อม ($S_t$) และทำการเลือกกระทำ action ($a_t$) จากนั้นในเวลาต่อมาที่เวลา t+1 Agent จะได้รับค่า reward ($r_{t+1}$) และสภาพแวดล้อมก็จะเปลี่ยน state ใหม่เป็น $S_{t+1}$

การที่ Agent ทำการแม็พค่า state ที่ได้ไปเลือก action ที่เป็นไปได้ เรียกว่า "Agent Policy ($\pi_t$)" โดยที่ $\pi_t(s,a)$ คือโอกาสที่จะเลือกกระทำ action $a$ ที่เวลา t ($a_t$) ถ้า state $s_t = s$ โดยที่เป้าหมายหลักของ Agent คือ พยายามให้ได้ค่า reward มากที่สุด ดังนั้นจึงต้องมีการกำหนดค่า reward ที่มีค่าเป็นตัวเลขให้กับ Agent เช่น การสอนให้หุ่นยนต์เดินจะต้องกำหนดค่า reward ในแต่ละช่วงเวลาตามผลที่ได้จากการเคลื่อนไหวของหุ่นยนต์

ยกตัวอย่าง เช่น Recycling Robot มีหน้าที่ในการเก็บกระป๋อง โดยมี sensor ในการตรวจจับกระป๋อง การตัดสินใจในการหากระป๋องจะขึ้นกับระดับของแบตเตอรี่ ซึ่ง action ที่ agent จะกระทำมี 3 แบบ คือ

1) เดินค้นหากระป๋อง
2) อยู่กับที่และรอให้มีคนมาทิ้งกระป๋องเอง
3) เดินกลับไป recharge แบตเตอรี่

โดย state จะพิจารณาจากสถานะของระดับแบตเตอรี่ และกำหนดให้ค่า reward มีค่าเป็น 0 ตลอดเวลา และมีค่าเป็นบวกเมื่อ robot สามารถเก็บกระป๋องได้ และมีค่าเป็นลบเมื่อระดับของแบตเตอรี่ลดลง

**2.4 Return**

ในแต่ละช่วงเวลา Agent จะได้รับค่า reward เป็นลำดับ คือ $r_{t+1}$, $r_{t+2}$, $r_{t+3}$, ... และเนื่องจาก Agent ต้องการได้รับ reward ให้มากที่สุด ดังนั้นค่า Expected Return ($R_t$) ที่ต้องการจะเป็นผลรวมของค่า reward ตามสมการ 2.1

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + r_{t+4} + \ldots + rT \qquad \ldots(2.1)$$

เมื่อ

$T$ คือ final time step

ถ้าการ interact กันระหว่าง Agent กับสภาพแวดล้อมมีการแบ่งออกเป็นลำดับย่อย (subsequence) ที่เรียกว่า "episode" โดยที่ episode จะสิ้นสุดที่ terminal stateจะเรียกลักษณะงานแบบนี้ว่า "episodic task" ซึ่งจะมีค่า Expected Return ตามสมการที่ 2.1

แต่ในบางกรณีการ interact กันระหว่าง Agent กับสภาพแวดล้อมไม่มีการแบ่งออกเป็น episode จะเรียกลักษณะงานแบบนี้ว่า "continuous task" ซึ่งทำให้ค่า final time step ($T$) = $\infty$ ดังนั้นค่า Expected Return คำนวณได้จากสมการ

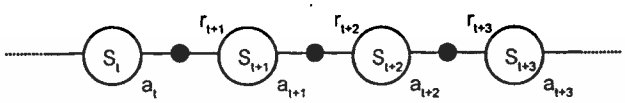
$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad \ldots(2.2)$$

เมื่อ

$\gamma$ คือ discount rate มีค่า $0 \leq \gamma \leq 1$

**2.5 Markov Decision Process (MDP)**

คุณสมบัติของสภาพแวดล้อม(environment) ใน RL จะถูกโมเดลให้เป็น MDP โดย Agent จะใช้ state ของสภาพแวดล้อมในการตัดสินใจเลือกกระทำ action ซึ่งแต่ละ state จะมีความสัมพันธ์กับ state ก่อนหน้า แสดงดังรูปที่ 2.3




**รูปที่ 2.3** Markov Decision Process

ในการตัดสินใจของ Agent จะยึด state ของ environment เป็นหลัก โดยปกติ state ของ environment จะมีคุณสมบัติของ Markov

Agent จะมีหน้าที่ในการตัดสินใจเลือก action ให้เป็นไปตาม state signal ของ environment ซึ่ง signal state จะหมายความรวมถึง immediate sensation เช่น การที่ Agent มองเห็นและสามารถบอกได้ว่าสิ่งที่มองเห็นคืออะไร และยังสามารถแยกแยะตำแหน่งของสิ่งของได้ ซึ่งทั้งหมดนี้ state จะถูกสร้างและ maintain โดยขึ้นกับ immediate sensation กับค่า state ก่อนหน้าหรือความจำของ state ก่อนหน้า

การที่ Agent รู้ state signal ไม่ได้หมายความว่า Agent จะรู้เกี่ยวกับ environment หรือสิ่งต่างๆที่ใช้ในการตัดสินใจ (decision making) เช่น ถ้า Agent กำลังเล่นไพ่ เราไม่ได้คาดหวังที่จะให้ Agent รู้ว่าไพ่ใบต่อไปในสำรับจะเป็นอะไร โดยในกรณีนี้จะมี hidden state information ใน environment

เพื่อให้การคำนวณง่ายขึ้นจะสมมติให้ state และค่า reward มีจำนวนจำกัด (finite number) ถ้า state signal มีคุณสมบัติของ Markov Environment จะมีการตอบสนองที่เวลา t+1 โดยจะขึ้นกับ state และ action ที่เวลา t โดยมีความน่าจะเป็นของ state และค่า reward ต่อไปตามสมการที่ 2.3

$$\Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \qquad \ldots(2.3)$$

ถ้าจำนวนของ state และ action มีจำนวนจำกัด จะเรียกว่า"finite MDP" ซึ่งมีค่า transition probabilities ตามสมการ 2.4

$$P^{a}_{ss'} = Pr\{s_{t+1} = s' | s_t = s, a_t = a\} \quad ...(2.4)$$

เมื่อ

$s, s' \in S, a \in A(s)$

$P^{a}_{ss'}$ คือ ความน่าจะเป็นในการเปลี่ยน state จาก $s$ เป็น $s'$ เมื่อเลือกกระทำ action $a$ และมีโอกาสที่จะได้รับค่า reward ตามสมการ 2.5

$$R^{a}_{ss'} = E\{r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'\} \quad ...(2.5)$$

เมื่อ

$s, s' \in S, a \in A(s)$

$R^{a}_{ss'}$ คือ ความน่าจะเป็นที่จะได้รับค่า expected reward จากการเปลี่ยน state จาก $s$ เป็น $s'$ เมื่อเลือกกระทำ action $a$

จากตัวอย่างของ Recycling Robot เป็นตัวอย่างของ MDP โดย action ที่ agent จะกระทำ คือ

1) เดินค้นหากระป๋อง
2) อยู่กับที่และรอให้มีคนมาทิ้งกระป๋องเอง
3) เดินกลับไป recharge แบตเตอรี่

จะเห็นว่า action ที่ดีที่สุดเพื่อให้ได้จำนวนกระป๋องมากที่สุด คือ การให้ robot เดินหากระป๋อง แต่การเดินจะทำให้ระดับแบตเตอรี่ลดลง สิ่งที่ agent จะต้องใช้ในการตัดสินใจคือระดับพลังงานของแบตเตอรี่ ซึ่งสามารถแบ่งออกเป็น 2 ระดับ คือ high และ low

ดังนั้นจะได้ State $S$ = { high, low } และเซตของ action set ของ agent จะมีค่าเป็น

$A$(high) = { search, wait }

$A$(low) = { search,wait,recharge }

ถ้าระดับพลังงานเป็น "high" ทำให้ agent สามารถเดิน หากระป๋องได้สำเร็จโดยปราศจากความเสี่ยงที่แบตเตอรี่จะหมด

ระดับพลังงานในช่วงการเดินหากระป๋องเริ่มต้นจะมีระดับพลังงานเป็น high ด้วยความน่าจะเป็น $\alpha$ และระดับพลังงานจะลดลงเป็น low ด้วยความน่าจะเป็น 1-$\alpha$ ในทางตรงข้ามระดับพลังงานในช่วงการเดินหากระป๋องที่มีระดับพลังงานเป็น low ด้วยความน่าจะเป็น $\beta$ และแบตเตอรี่จะถูกใช้จนหมดด้วยความน่าจะเป็น 1-$\beta$ ซึ่ง robot จะถูกช่วยเหลือโดยการ recharge แบตเตอรี่ให้เป็น high

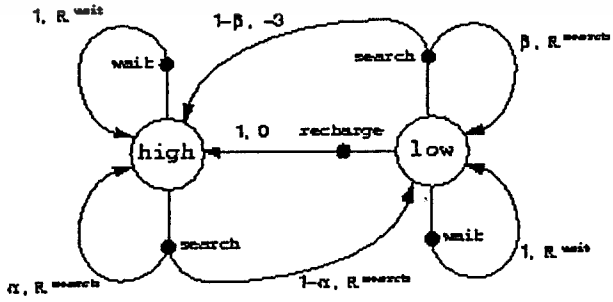
ค่า reward ที่ได้จะมีค่าตามจำนวนกระป๋องที่เก็บได้ โดย robot จะได้รับค่า reward เป็น –3 เมื่อ robot ถูกช่วยเหลือโดยการ recharge แบตเตอรี่ให้เป็น high โดยให้ $R^{\text{SEARCH}}$ คือ ค่า reward ที่ได้จากการเดินหากระป๋อง , $R^{\text{WAIT}}$ คือ ค่า reward ที่ได้จากการรอให้คนอื่นมาทิ้งกระป๋อง ซึ่งค่า $R^{\text{SEARCH}} > R^{\text{WAIT}}$ สามารถเขียนให้อยู่ในรูปของ finite MDP โดยมีค่า transition probability และค่า expected reward ได้ตามตารางที่ 2.1

| $s = s_t$ | $s' = s_{t+1}$ | $a = a_t$ | $\mathcal{P}^a_{ss'}$ | $\mathcal{R}^a_{ss'}$ |
|---|---|---|---|---|
| high | high | search | $\alpha$ | $\mathcal{R}^{\text{search}}$ |
| high | low | search | $1-\alpha$ | $\mathcal{R}^{\text{search}}$ |
| low | high | search | $1-\beta$ | $-3$ |
| low | low | search | $\beta$ | $\mathcal{R}^{\text{search}}$ |
| high | high | wait | 1 | $\mathcal{R}^{\text{wait}}$ |
| high | low | wait | 0 | $\mathcal{R}^{\text{wait}}$ |
| low | high | wait | 0 | $\mathcal{R}^{\text{wait}}$ |
| low | low | wait | 1 | $\mathcal{R}^{\text{wait}}$ |
| low | high | recharge | 1 | 0 |
| low | low | recharge | 0 | 0. |

ตารางที่ 2.1 transition probability และค่า expected reward ของ Recycling Robot

สามารถแสดงการเปลี่ยนแปลงของ finite MDP ในตารางที่ 2.1 ได้ด้วย transition graph ดังแสดงในรูปที่ 2.4 ซึ่งจะมีโหนด 2 โหนด คือ state node (แทนด้วยวงกลมสีขาว) และ action node (แทนด้วยวงกลมสีดำ)

เริ่มต้นที่ state $s$ โดยการเลือก action $a$ จะเป็นการเปลี่ยนจาก state node $S$ ไปเป็น action node $(s,a)$ เมื่อ environment มีการ response ในการ transition ไปเป็น next state node โดยผ่าน action node $(s,a)$ ซึ่งลูกศรแต่ละเส้นจะแทนด้วย $(s,s',a)$ โดยที่ $s'$ คือ state ถัดไป




**รูปที่ 2.4** Transition graph ของ Recycling Robot

**2.6 Value Functions**

เป็นฟังก์ชันของ state (state-action pairs) ที่ใช้ในการประเมินว่า Agent ได้รับผลอย่างไรใน state นั้นๆ หรือ Agent ได้รับผลอย่างไรที่เลือกกระทำ action นั้นจาก state ที่ได้รับ ซึ่งผลลัพธ์ที่กล่าวถึงก็คือ ค่า reward ที่จะได้ในอนาคต (futute reward) ค่า value function สามารถแบ่งออกได้เป็น 2 แบบ คือ

**2.6.1 State-value function : $V^{\pi}(s)$**

ภายใต้ policy $\pi$ State-value function เป็นค่า Expected return ที่ได้รับตั้งแต่ state $s$ จนถึง state ปัจจุบันนิยามได้ตามสมการ

$$V^{\pi}(s) = E_{\pi}\{R_t \mid s_t = s\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\} \quad ...(2.6)$$

เมื่อ

$E_{\pi}$ คือ Expected value ที่ Agent ปฏิบัติตาม policy $\pi$ ซึ่งสามารถเขียนให้อยู่ในรูปของสมการBellman ได้ดังนี้

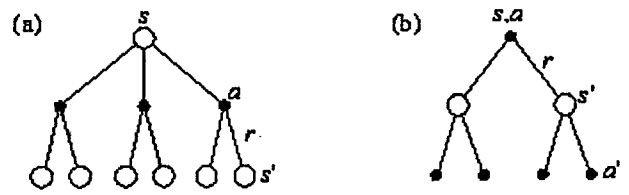
$$V^{\pi}(s) = \sum_{a} \pi(s,a) \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')] \quad ...(2.7)$$

**2.6.2 Action-value function : $Q^{\pi}(s,a)$**

ภายใต้ policy $\pi$ Action-value function เป็นค่า Expected return ที่ได้รับตั้งแต่ state $s$ จนถึง state ปัจจุบัน โดยการเลือกกระทำ action $a$ นิยามได้ตามสมการ

$$Q^{\pi}(s,a) = E_{\pi}\{R_t \mid s_t = s, a_t = a\} = E_{\pi}\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\} \quad ...(2.8)$$

backup diagram เป็นไดอะแกรมที่ใช้บอกความสัมพันธ์ในการอัพเดท หรือ backup operation แสดงดังรูปที่ 2.5 ซึ่งเป็นหัวใจสำคัญของ RL โดยโอเปอเรชั่นเหล่านี้จะมีการ transfer ค่ากลับไปยัง state หรือ state-action pair จาก state ถัดไป




รูปที่ 2.5 backup diagram ของ (a) $V^{\pi}$ และ (b) $Q^{\pi}$

**2.7 Optimal Value Functions**

วิธีในการแก้ปัญหา RL คือ การหา policy ที่จะได้รับค่า rewards จำนวนมากในระยะยาว ซึ่ง policy $\pi$ จะดีกว่า policy $\pi'$ ก็ต่อเมื่อ $V^{\pi}(s) \geq V^{\pi}(s')$ โดยจะมีเพียงอย่างน้อย 1 policy เท่านั้นที่ดีกว่าหรือมีค่าเท่ากับ policy อื่น เรียก policy นี้ว่า "optimal policy ($\pi^*$) " โดยที่มีการแชร์ค่า state-value function เดียวกัน เรียกว่า "optimal state-value function ($V^*$)" ซึ่งนิยามได้ตามสมการ

$$V^*(s) = \max_{\pi} V^{\pi}(s) \qquad \ldots(2.9)$$

และมีการแชร์ค่า action value function เดียวกัน เรียกว่า "optimal action-value function ($Q^*$)" ซึ่งนิยามได้ตามสมการ

$$Q^*(s,a) = \max_{\pi} Q^{\pi}(s,a) \qquad \ldots(2.10)$$

สำหรับ state-action pair $(s,a)$ ฟังก์ชันนี้จะให้ค่า expected return จากการเลือก action $a$ ใน state $s$ ภายใต้ optimal policy ซึ่งสามารถเขียน $Q^*$ ให้อยู่ในรูปของ $V^*$ ได้ ตามสมการ

$$Q^*(s,a) = E\{r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a\} \qquad \ldots(2.11)$$

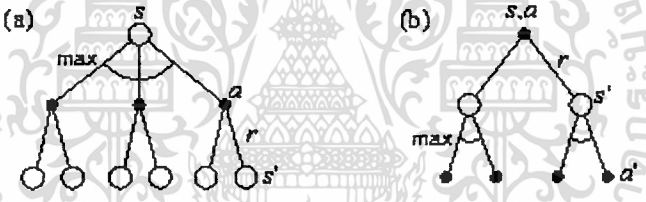
จะพบว่าภายใต้ optimal policy ค่า state-value จะมีค่าเท่ากับค่า expected return ในการเลือก action ที่ดีที่สุดจาก state นั้นๆ สามารถแสดงให้อยู่ในรูปของสมการ Bellman optimality equation ได้ตามสมการ 2.12

$$V^*(s) = \max_{a \in A(s)} Q^{\pi^*}(s,a) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^*(s')] \quad \ldots(2.12)$$

และสามารถแสดงค่า optimal action-value function ในรูปของสมการ Bellman optimality equation ได้ตามสมการ 2.13

$$\begin{aligned} Q^*(s,a) &= E\{r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a\} \\ &= \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma \max_{a'} Q^*(s', a')] \end{aligned} \quad \ldots(2.13)$$

backup diagram ของ optimal state-value function และ optimal action-value function จะเหมือนกับ bakcup diagram ของ $V^\pi$ และ $Q^\pi$ แต่แตกต่างกันที่เป็นการเลือก state หรือ state-action pair ที่ให้ค่า value มากที่สุด แสดงดังรูปที่ 2.6




รูปที่ 2.6 แสดง backup diagram ของ (a) $V^*$ และ (b) $Q^*$

## 2.8 อัลกอริธึมที่ใช้ในการแก้ปัญหา Reinforcement Learning

### 2.8.1 Dynamic Programming (DP)

การคำนวณหาค่า state-value function ($V^\pi$) สำหรับ policy $\pi$ ใน DP เรียกว่า "Policy evaluation" ซึ่งมักจะเรียกว่าเป็น "prediction problem" จะได้ค่า state-value function ตามสมการที่ 2.7 โดย $\pi(s,a)$ เป็นความน่าจะเป็นในการเลือก action $a$ ใน state $s$ ภายใต้ policy $\pi$ ในการคำนวณจะใช้วิธี iterative method ซึ่งจะเป็นการพิจารณาค่า sequence ของ approximation value function $V_0$ , $V_1$ , $V_2$ ,... โดยการหาค่า $V_1$ , $V_2$ ,... สามารถคำนวณได้จาก Bellman equation ตามสมการ

$$V_{k+1}(s) = \sum_a \pi(s,a) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V_k(s')] \quad \ldots(2.14)$$

จะเห็นว่า sequence $\{V_k\}$ จะ converge เข้าสู่ $V^\pi$ ถ้า k-> $\infty$ เรียก algorithm นี้ว่า "iterative policy evaluation"

ในการ approximation ค่า $V_{k+1}$ จาก $V_k$ จะเป็นการ apply operation เหมือนกันในแต่ละ state $S$ โดยจะทำการ replace ค่าเก่าของ state ก่อนหน้า เรียกโอเปอเรชั่นแบบนี้ว่า "full backup" ในแต่ละครั้งของการ iteration จะมีการ backup ค่าของทุกๆ state 1 ครั้งก่อนแล้วจึงทำการ approximation new value function $V_{k+1}$

วิธีการของ full backup มีหลายประเภทขึ้นกับ state หรือ state-action pair ที่กำลังถูก backup และขึ้นกับความแม่นยำในการ estimate ค่าของ state ถัดไป การ backup ใน DP algorithm จะเป็น full backup เพราะมันจะ base on ทุกๆ next state ที่เป็นไปได้มากกว่าที่จะเป็นค่า sample next state

ขั้นตอนในการ implement Iterative policy evaluation สามารถทำได้ตามรูปที่ 2.7

Input $\pi$, the policy to be evaluated

Initialize $V(S) = 0$ ,for all $S \in S+$

Repeat

$\Delta \leftarrow 0$

For each $S \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Until

$\Delta < \theta$

Output

$V \approx V^\pi$

**รูปที่ 2.7** Iterative policy evaluation

การที่ต้องมีการคำนวณหาค่า value function สำหรับ policy ก็เพื่อต้องการหา policy ที่ดีกว่า สมมติว่าในการพิจารณาค่า value function $V^\pi$ สำหรับ policy $\pi$ ซึ่งในบาง state จะต้องการรู้

ว่าควรเปลี่ยน policy หรือไม่ โดยการเลือก $a \neq \pi(s)$ โดยการพิจารณาจาก $V^{\pi}(s)$ แต่จะไม่รู้ว่ามันจะดีขึ้นหรือแย่ลงถ้ามีการเปลี่ยนแปลง policy ใหม่

สิ่งที่จะช่วยได้ คือ การพิจารณาในการเลือก action $a$ ใน state $s$ ตาม policy $\pi$ ค่า value ที่ได้จะมีค่าตามสมการ 2.15

$$\begin{aligned} Q^{\pi}(s,a) &= E_{\pi}\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V_k(s')] \end{aligned} \quad \ldots(2.15)$$

โดยมีเงื่อนไขว่าค่าที่ได้จะมีค่ามากกว่าหรือน้อยกว่า $V^{\pi}(s)$ ถ้ามีค่ามากกว่าจะทำการเลือก action $a$ ทุกครั้งที่เกิด state $s$ ซึ่งเป็นการแสดงว่า policy ใหม่ดีกว่า policy เก่า

ถ้า $Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s)$ แสดงว่า policy $\pi'$ ดีกว่า $\pi$ เรียกวิธีการแบบนี้ว่า "policy improvement theorem" ดังนั้นจะได้ค่า expect return ที่มากกว่าหรือเท่ากันในทุก state $s \in S$ ตามสมการ 2.16

$$V^{\pi'}(s) \geq V^{\pi}(s) \quad \ldots(2.16)$$

เมื่อมี policy และ value function ก็จะสามารถทำการ evaluate ในการที่จะเปลี่ยน policy ได้ง่ายขึ้นโดยการเลือก action ในแต่ละ state ที่ให้ค่า $Q^{\pi}(s,a)$ มากที่สุด หรือกล่าวได้ว่า policy ใหม่เป็น greedy policy , $\pi'$ ได้ตามสมการ 2.17

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^{\pi}(s,a) \\ &= \arg\max_a E\{r_{t+1} + \gamma V^{\pi}(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \arg\max_a \sum_{s'} P^{a}_{ss'}[R^{a}_{ss'} + \gamma V^{\pi}(s')] \end{aligned} \quad \ldots(2.17)$$

โดย $argmax_a$ หมายถึง ค่าของ $a$ ที่ทำให้เกิดค่าที่มากที่สุด

กระบวนการที่ทำการปรับปรุง policy เดิมไปเป็น policy ใหม่โดยการทำ greedy เรียกว่า "policy improvement"

สมมติว่า new greedy policy ($\pi'$) เป็น policy ที่ดี แต่ไม่ดีกว่า policy เดิม ($\pi$) ดังนั้น $V^{\pi} = V^{\pi'}$ จากสมการที่ 2.17 จะได้

$$V^{\pi'}(s) = \max_a \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma V^{\pi'}(s')] \quad \ldots(2.18)$$

ซึ่งจะเหมือนกับสมการ Bellman Optimal equation ดังนั้น $V^{\pi'} = V^*$ และ policy $\pi$ และ $\pi'$ จะเป็น optimal policy

ในการปรับปรุง policy โดยใช้ $V^{\pi}$ เพื่อหา policy $\pi'$ ที่ดีขึ้น จะได้ sequence ของ policy และ value function ดังนี้

$$\pi_0 \xrightarrow{E} V^{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} \cdots \xrightarrow{I} \pi^* \xrightarrow{E} V^*$$

โดย $\xrightarrow{E}$ คือ policy evaluation และ $\xrightarrow{I}$ คือ policy improvement

แต่ละ policy ก็จะถูก improve จาก policy ก่อน นอกจากว่า policy นั้นจะเป็น optimal policy อยู่แล้ว เนื่องจาก finite MDP มี policy เป็นแบบ finite number ดังนั้นกระบวนการนี้ก็จะ converge สู่ optimal policy และ optimal value function

วิธีในการหา optimal policy นี้จะเรียกว่า "policy iteration" แต่ละ policy จะมีการ evaluate ตัวมันเองและทำการคำนวณซ้ำ โดยจะเริ่มจาก policy ก่อนหน้า อัลกอริทึมในการทำ policy iteration แสดงดังรูปที่ 2.8

1. Initialization

$V(s) \in \Re$ and $\pi(s) \in A(s)$ arbitrarily for all $s \in S$

2. Policy Evaluation

Repeat

$\Delta \leftarrow 0$

For each $s \in S$

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_{s'} \mathcal{P}_{ss'}^{\pi(s)} \left[\mathcal{R}_{ss'}^{\pi(s)} + \gamma V(s')\right]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Until

$\Delta < \theta$

3. Policy Improvement

$policy\text{-}stable \leftarrow true$

For each $s \in S$

$b \leftarrow \pi(s)$

$\pi(s) \leftarrow \arg\max_a \sum_{s'} \mathcal{P}_{ss'}^{a} [\mathcal{R}_{ss'}^{a} + \gamma V(s')]$

If $b \neq \pi(s)$, then policy-stable $\leftarrow false$

If policy-stable , then stop; else go to 2

รูปที่ 2.8 Policy Iteration algorithm

ข้อเสียของ policy iteration คือ ในการ iteration ในแต่ ละครั้งจะมีการทำ policy evaluation ทุกครั้ง

สิ่งที่สำคัญคือเมื่อ policy evaluation ถูก stop หลังจากมีการ sweep เพียง 1 ครั้ง ( 1 backup ของแต่ละ state) อัลกอริทึมนี้เรียกว่า "value iteration" สามารถเขียน simple backup operation ที่มีการรวม policy improvement และ policy evaluation step เข้าด้วยกันตามสมการ 2.18

$$V_{k+1}(s) = \max_a E\{r_{t+1} + \gamma V_k(s_{t+1}) \mid s_t = s, a_t = a\}$$
$$= \max_a \sum_{s'} P_{ss'}^{a} [R_{ss'}^{a} + \gamma V_k(s')] \quad \ldots(2.18)$$

จะสังเกตได้ว่า value iteration backup จะเหมือนกับ policy evaluation backup แต่ต่างกันที่จะทำการ maximize ในทุกๆ action อัลกอริทึมที่ใช้ในการทำ value iteration แสดงดังรูปที่ 2.9

Initialize $V$ arbitrarily,e.g. $V(s) = 0$ ,for all $s \in S+$

Repeat

$\Delta \leftarrow 0$

For each $s \in S$:

$v \leftarrow V(s)$

$V(s) \leftarrow \max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Until

$\Delta < \theta$

Output a deterministic policy, $\pi$, such that

$\pi(s) = \arg\max_a \sum_{s'} \mathcal{P}^a_{ss'} [\mathcal{R}^a_{ss'} + \gamma V(s')]$

รูปที่ 2.9 Value iteration algorithm

การที่จะทำให้ converge สู่ค่า optimal ได้เร็ว คือการทำ multiple policy evaluation sweep ระหว่างการทำ policy improvement sweep 1 ครั้ง

ข้อเสียของวิธีการ DP คือ จะต้องเกี่ยวข้องกับโอเปอเรชั่นในทุกๆ state set ของ MDP ซึ่งจะต้องมีการ sweep state set เหล่านั้น ซึ่งจะไม่เหมาะกับ state set ที่มีขนาดใหญ่ ซึ่งสามารถแก้ไขได้โดยการใช้ Asynchronous Dynamic Programming

วิธีการของ Asynchronous Dynamic algorithm จะใช้แทนที่วิธีการ iterative DP algorithm โดยจะทำการ backup ค่าของ stateใดๆก็ได้ ซึ่งค่าของบาง state อาจจะถูก backup หลายๆครั้งก่อนที่ value ของ state อื่นจะถูก backup 1 ครั้ง โดย Asynchronous Dynamic algorithm จะมีความยืดหยุ่นในการเลือก state ที่จะทำการ backup ได้ดีกว่า

Asynchronous algorithm เหมาะกับการนำไปใช้ในงานแบบ real-time interaction ซึ่งสามารถรัน iterative DP algorithm ในเวลาเดียวกับที่ agent กำลังเรียนรู้จาก MDP

**2.8.2 Temperal-Difference Learning (TD)**

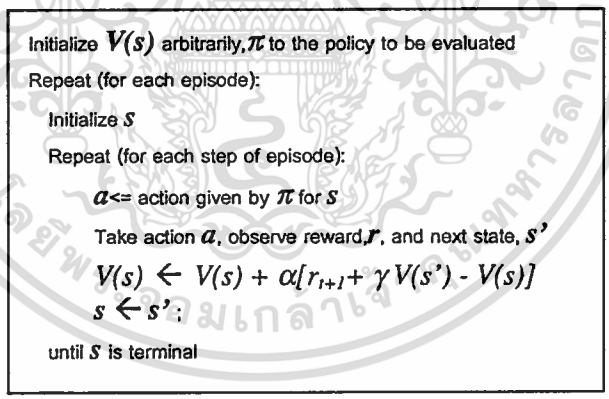
TD Learning จะใช้ประสบการณ์ในการแก้ปัญหา โดยไม่ต้องอาศัยโมเดลของ environment โดยจะทำการอัพเดทค่า estimate $V(S_t)$ จาก reward $R_{t+1}$ และ estimate $V(S_{t+1})$ เรียกวิธีการแบบนี้ว่า "TD(0)" สามารถเขียนได้ตามสมการ 2.19

$$V(s_t) \longleftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad ...(2.19)$$

จากสมการ 2.19 จะเห็นได้ว่า target ในการอัพเดทของ TD คือ $r_{t+1} + \gamma V_t(S_{t+1})$

algorithm ของ TD(0) แสดงดังรูปที่ 2.10 และรูปที่ 2.11 แสดง backup diagram ของ TD(0) ที่เป็น "sample backup" เพราะเกี่ยวข้องกับ sample successor state โดยใช้ค่า value ของ successor และค่า reward ในการคำนวณหา backup value จากนั้นก็เปลี่ยนแปลงค่า value ของ original state

Sample backup แตกต่างจาก full backup ตรงที่ sample backup จะขึ้นกับแค่ single sample successor แทนที่จะเกี่ยวข้องกับทุก state ที่เป็นไปได้




**รูปที่ 2.10** TD(0) algorithm

**รูปที่ 2.11** backup diagram ของ TD(0)

วิธีการของ TD method เป็นการ estimate state ที่ตามมา แต่ละ error ที่เกิดขึ้นในการ prediction จะมีการเปลี่ยนแปลงตลอดเวลา ซึ่งเรียกว่า "temperal difference"

ข้อดีของ TD method

- ไม่ต้องมีโมเดลของ environment
- สามารถทำการ implement แบบ on-line ได้โดยไม่ต้องรอจนกระทั่งสิ้นสุด episode โดย TD method จะรอเพียงแค่ 1 time step เท่านั้น
- ทำให้การเรียนรู้เร็วเนื่องจากเป็นการเรียนรู้จากแต่ละ transition โดยที่ไม่คำนึงถึง subsequence ของ action ที่ถูกเลือก

2.8.3 TD($\lambda$)

เป็นอัลกอรึธึมที่นำเอา eligibility trace มาใช้งานร่วมกับ TD Learning โดยในแต่ละ state จะมี eligibility trace เข้ามาเกี่ยวข้องโดย $e_t(s) \in \Re^+$ ซึ่งในแต่ละช่วงเวลา eligibility trace จะลดลงด้วย $\gamma\lambda$ ส่วน state ที่ถูก visit ค่าของ eligibility trace จะมีค่าเพิ่มขึ้นทีละหนึ่งตามสมการ 2.20

$$e_t(s) = \begin{cases} \gamma\lambda_t e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda_t e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \quad \ldots(2.20)$$

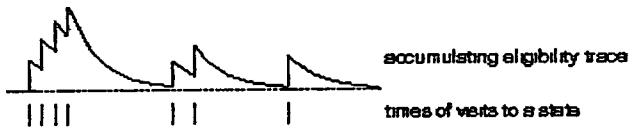
เมื่อ

$s \in S$

$\gamma$ คือ discount rate

$\lambda$ คือ trace-decay parameter

ลักษณะของ eligibility trace แบบนี้เรียกว่า "accumulating trace" เนื่องจากในแต่ละครั้งที่มีการ visit state ค่า eligibility trace จะถูกสะสมและจะค่อยๆลดลงเมื่อ state ไม่ได้ถูก visit ซึ่งแสดงดังรูป




**รูปที่ 2.12** Accumulating eligibility trace

ในแต่ละเวลาจะมีการบันทึก eligibility trace state ที่ถูก visit โดยจะนิยามในเทอมของ $\gamma\lambda$ ซึ่ง trace จะเป็นตัวชี้วัดว่า state ไหนเหมาะสมที่จะมีการเปลี่ยนแปลงการเรียนรู้ที่จะเป็น TD error ณ ขณะนั้น โดย TD error สำหรับ state-value prediction จะเป็นไปตามสมการ 2.21

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \quad \text{...(2.21)}$$

โดย TD error จะเป็นสัดส่วนตามการอัพเดทของทุก state ที่มีการ visit มาก่อนหน้านี้ตามสมการ 2.22

$$\Delta V_t(s) = \alpha\delta_t e_t(s) \quad \text{...(2.22)}$$

ซึ่งการ increment จะเป็นการกระทำในแต่ละ step ที่เป็น on-line algorithm หรือแบบ off-line โดยอัลกอริธึมแบบ on-line TD($\lambda$) แสดงดังรูปที่ 2.13 ซึ่งแต่ละการอัพเดทจะขึ้นกับ current TD error กับ trace ของเหตุการณ์ที่เกิดก่อน ซึ่งแสดงดังรูปที่ 2.14

Initial $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$

Repeat (for each episode)

Initialize $s$

Repeat (for each step of episode):

$a \leftarrow$ action given by $\pi$ for $s$

Take action $a$, observe reward $r$, and next state $s'$

$\delta \leftarrow r + \gamma V(s') - V(s)$

$e(s) \leftarrow e(s) + 1$

for all $s$:

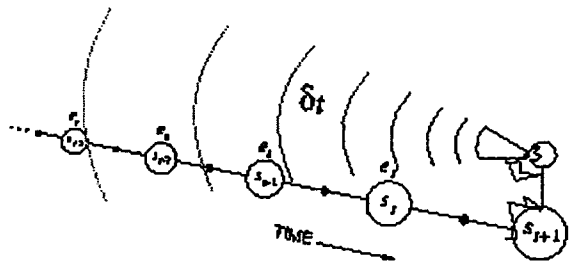
$V(s) \leftarrow V(s) + \alpha\delta e(s)$

$e(s) \leftarrow \gamma\lambda e(s)$

$s \leftarrow s'$

until $s$ is terminal

**รูปที่ 2.13** On-line tabular TD($\lambda$)




รูปที่ 2.14 Backward view of TD($\lambda$)

ถ้าค่า $\lambda = 0$ จะทำให้ eligibility trace มีค่าเป็น 0 หมด ยกเว้นที่ state $S_t$ ทำให้การอัพเดทตามสมการ 2.22 มีค่าเป็น

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \quad ...(2.23)$$

ในกรณีของ TD(0) จะมีเพียง 1 state ก่อนหน้า state ปัจจุบันที่มีการเปลี่ยนแปลงด้วย TD error สำหรับ ค่า $0<\lambda<1$ จะมีจำนวน state ก่อนหน้าที่มีการเปลี่ยนแปลงมากขึ้น แต่ละ state จะถูกเปลี่ยนแปลงน้อยลงเพราะ eligibility trace มีค่าน้อยลง ซึ่งอาจกล่าวได้ว่า state ที่เกิดก่อนจะมีผลต่อ credit ของ TD error น้อยลง

**2.8.4 Gradient-Descent Method**

ในกรณีปัญหาที่มีจำนวน state ไม่มากก็สามารถใช้การ lookup table ได้ แต่ในกรณีที่เป็นปัญหาที่มีความซับซ้อนต้องใช้ function approximation ที่ใช้ใน supervised learning เช่น neural network เป็นต้น

ถ้ากำหนดให้value function ($V_t$) เป็นฟังก์ชันของ $\vec{\theta}_t$ โดยทั่วไปจำนวนพารามิเตอร์จะน้อยกว่าจำนวน state และการเปลี่ยนแปลงพารามิเตอร์ 1 ตัวจะมีผลทำให้ estimate value เกิดการเปลี่ยนแปลงหลาย state

Gradient-Descent เป็นวิธีที่นิยมใช้กันมากในการทำ function approximation โดย parameter vector จะเป็น vector ที่เป็นจำนวนจำกัดของ real valued component, $\vec{\theta}_t = (\theta_t(1), \theta_t(2), ..., \theta_t(n))^T$ และ $V_t(s)$ จะเป็น differentiable function ของ$\vec{\theta}_t$ โดย gradient-descent method สำหรับ state-value prediction จะเป็นไปตามสมการ 2.24

$$\vec{\theta}_t = \vec{\theta}_t + \alpha(v_t - V_t(s_t))\nabla_{\vec{\theta}_t} V_t(s_t) \quad \ldots(2.24)$$

ในกรณีของ TD($\lambda$) ค่า $v_t = R_t^{\lambda}$ เพราะฉะนั้นสมการที่ 2.24 สามารถเขียนได้เป็น

$$\begin{aligned} \vec{\theta}_{t+1} &= \vec{\theta}_t + \alpha(R_t^{\lambda} - V_t(s_t))\nabla_{\vec{\theta}_t} V_t(s_t) \\ &= \vec{\theta}_t + \alpha\delta_t \vec{e}_t \end{aligned} \quad \ldots(2.25)$$

เมื่อ

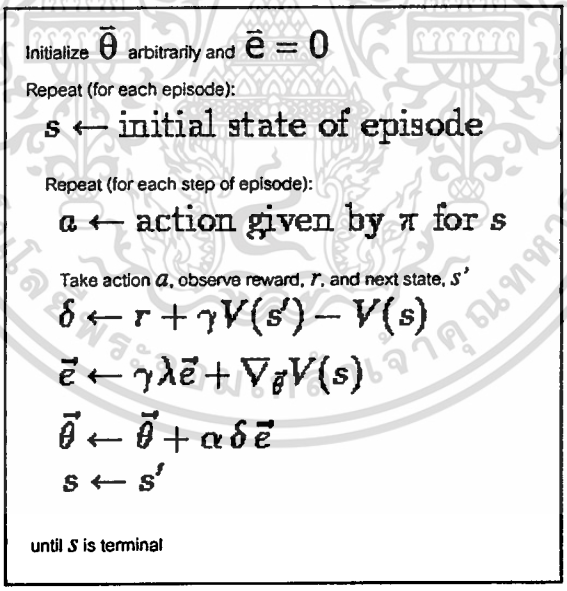
$\delta_t$ คือ TD error จะได้

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t) \quad \ldots(2.26)$$

$\vec{e}_t$ คือ column vector ของ eligibility traceซึ่งแต่ละ component ของ $\vec{\theta}_t$ จะอัพเดทโดย

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) \quad \ldots(2.27)$$

อัลกอริธึมของ on-line gradient-descent TD($\lambda$) แสดงดังรูปที่ 2.15




**รูปที่ 2.15** On-line gradient-descent TD($\lambda$)

### 2.9 การ Prediction โดยใช้ Temporal Difference

Prediction เป็นการวิเคราะห์ข้อมูลที่ผ่านมาเพื่อจะคาดการณ์สถานการณ์ล่วงหน้า เช่น การทำนายตำแหน่งของหมากรุก,การพยากรณ์อากาศ, การคาดคะเนสภาวะตลาดหุ้น เป็นต้น การเรียนรู้

ในการทำ prediction มักจะพบมากใน heuristic search ซึ่งสิ่งสำคัญของการทำ prediction คือ การเรียนรู้แบบไม่มีผู้สอน

ขณะที่การเรียนรู้แบบเดิมจะเป็นการวัด error โดยคำนวณจากผลต่างระหว่างค่าเอาต์พุทที่แท้จริงกับค่าเอาต์พุทที่ต้องการ แต่วิธีการของ Temporal Difference จะเป็นลักษณะของผลต่างระหว่าง successive prediction ที่เป็นการเรียนรู้ที่มีการเปลี่ยนแปลงตลอดเวลา

ข้อดีของการทำ Prediction โดยใช้ Temporal Difference

1. เป็นวิธีการแบบ incremental ทำให้ง่ายต่อการคำนวณ ตัวอย่างเช่น การพยากรณ์อากาศในวันเสาร์ ถ้าเป็น Prediction แบบ Temporal Difference จะสามารถอัพเดทการ prediction ในแต่ละวันกับวันถัดไปได้ ในขณะที่การ prediction แบบเดิมจะต้องรอจนถึงวันเสาร์จึงจะมีการเปลี่ยนแปลงการ prediction ของทั้งอาทิตย์ ซึ่งจะต้องใช้การคำนวณมากกว่าแบบ Temporal Difference และต้องเปลืองหน่วยความจำในการเก็บข้อมูลของทั้งสัปดาห์
2. มีประสิทธิภาพที่ดีกว่า และให้ผลลัพธ์ในการ prediction ที่แม่นยำ

Prediction-learning แบ่งออกเป็น 2 ประเภท คือ

1. Single-step prediction เป็นการ prediction ที่ทุกๆข้อมูลจะถูกเปิดเผยในครั้งเดียว
2. Multi-step prediction ความถูกต้องในการ prediction จะไม่ถูกเปิดเผยจนกระทั่งหลังจากการ prediction ผ่านไป 1 step และข้อมูลบางส่วนที่เกี่ยวข้องกันจะถูกเปิดเผยในแต่ละ step

ในบทความนี้จะพิจารณาเฉพาะ multi-step prediction ซึ่ง experience จะมาจากผลลัพธ์ที่สังเกตได้ โดยมีลำดับดังนี้ คือ $x_1, x_2, x_3, \ldots, x_m, z$ โดยที่แต่ละ $x_t$ เป็น vector ของสิ่งที่สังเกตได้ที่เวลา t และ $z$ คือ ผลลัพธ์ของ sequence

ในแต่ละ observation-outcome sequence Agent จะสร้างลำดับของ prediction ดังนี้ คือ $P_1, P_2, P_3, \ldots, P_m$ ซึ่งแต่ละอันจะเป็นการ estimate ค่า $z$

แต่ละ prediction $P_t$ จะเป็นฟังก์ชันของเหตุการณ์ที่เกิดขึ้นก่อนหน้าที่เวลา t ซึ่งเป็นฟังก์ชันของ $x_t$ โดยการ prediction จะพิจารณาจากค่า weight ($w$) ดังนั้น $P_t$ จะเป็นฟังก์ชันของ $x_t$ และ $w$ สามารถเขียนได้เป็น $P(x_t, w)$

ค่าของ $w$ ที่เปลี่ยนแปลงไปจะแทนด้วย $\Delta w_t$ หลังจากสิ้นสุด sequence ค่า $w$ จะถูกอัพเดทด้วยผลรวมของการเปลี่ยนแปลงของ $w$ ในทุก sequence

$$w \leftarrow w + \sum_{t=1}^{m} \Delta w_t \qquad \ldots(2.28)$$

ในวิธีการของ supervised-learning แต่ละลำดับของสิ่งที่สังเกตได้และผลลัพธ์ของมันจะมีการจับเป็นคู่ของ sequence ของ observation-outcome pair เช่น $(x_1,z)$ , $(x_2,z)$ , ...,$(x_m,z)$ ค่าที่เพิ่มขึ้นจะขึ้นกับ error ระหว่าง $P_t$ กับ $z$ โดยการอัพเดทของ supervised-learning จะเป็น

$$\Delta w_t = \alpha(z - P_t)\nabla_w P_t \qquad \ldots(2.29)$$

เมื่อ

$\alpha$ คือ learning rate

$\nabla w_t$ คือ vector of partial derivatives ของ $P_t$

ในกรณีที่ $P_t$ เป็น linear function ของ $x_t$และ $w$ โดยที่

$$P_t = w^T x_t = \sum_i w(i)x_i(i) \qquad \ldots(2.30)$$

เมื่อ

$w(i)$ และ $x_t(i)$ เป็น component ตัวที่ i ของ $w$ และ $x_t$ ตามลำดับ

ในที่นี้ $\nabla w\, P_t = x_t$ ดังนั้นสมการที่ 2.29 สามารถเขียนให้อยู่ในรูปของ Widrow-Hoff rule ได้ดังสมการ

$$\Delta w_t = \alpha(z - w^T x_t)x_t \qquad \ldots(2.31)$$

แนวความคิดของวิธีนี้ คือ ผลต่างของ $z$-$w^T x_t$ จะเป็น scalar error ระหว่าง prediction ของ $w^T x_t$ และ $z$ ซึ่งจะคูณด้วย observation vector $x_t$ โดย $x_t$ จะเป็นตัวชี้วัดว่าการเปลี่ยนแปลงของแต่ละ weight จะมีผลต่อ error อย่างไร ถ้า error เป็นบวกจะทำให้ $x_t(i)$ เป็นบวกด้วย ดังนั้นจะทำให้ $w_i(t)$ มีค่าเพิ่มขึ้น เป็นผลทำให้ $w^T x_t$ เพิ่มขึ้น ซึ่งจะทำให้ error ลดลง

ในกรณีที่ $P_t$ เป็น non-linear function ของ $x_t$ และ $w$ ซึ่งสามารถคำนวณโดยใช้ multi-layer network

ไม่ว่าจะเป็นกรณีใดก็ตาม ทุก $\Delta w_t$ ในสมการที่ 2.29 จะขึ้นกับค่า $z$ และไม่สามารถพิจารณาได้จนกว่าจะจบ sequence ดังนั้นทุก observation และ prediction ระหว่าง sequence จะต้องถูกเก็บเอาไว้จนกว่าจะจบ sequence ซึ่งจะเห็นว่าสมการที่ 2.29ไม่สามารถคำนวณแบบ incremental ได้

ในวิธีการของ Temporal Difference สามารถทำการคำนวณแบบ incremental ได้โดย error ระหว่าง $z$-$P_t$จะเป็นผลรวมของการเปลี่ยนแปลง prediction ตามสมการ 2.32

$$z - P_t = \sum_{k=t}^{m}(P_{k+1} - P_k) \qquad \ldots(2.32)$$

สมการ 2.28 และ 2.29 สามารถเขียนได้เป็น

$$w \leftarrow w + \sum_{t=1}^{m}\alpha(z - P_t)\nabla_w P_t = w + \sum_{t=1}^{m}\alpha\sum_{k=1}^{m}(P_{k+1} - P_k)\nabla_w P_t$$

$$= w + \sum_{k=1}^{m}\alpha\sum_{t=1}^{m}(P_{k+1} - P_k)\nabla_w P_t \qquad \ldots(2.33)$$

$$= w + \sum_{t=1}^{m}\alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\nabla_w P_k$$

สามารถเขียนให้อยู่ในรูปสมการที่ 2.28 ได้เป็น

$$\Delta w_t = \alpha(P_{t+1} - P_t)\sum_{k=1}^{t}\nabla_w P_k \qquad \ldots(2.34)$$

สมการที่ 2.34 ต่างจากสมการที่ 2.29 ตรงที่สามารถคำนวณแบบ incremental ได้ เพราะแต่ละ $\Delta w_t$ จะเกี่ยวข้องกับ prediction ที่ติดกันและผลรวมของ $\nabla w P_t$ ซึ่งช่วยประหยัดหน่วยความจำในการเก็บทุกค่าของ $\nabla w P_t$ และจะมีการคำนวณค่า $w$ ที่เปลี่ยนแปลงในแต่ละ time step แต่สมการที่ 2.29 จะต้องรอจนกระทั่งจบ sequence แล้วจึงค่อยทำการคำนวณ

## 2.10 ตัวอย่างการนำ Reinforcement Learning ไปใช้ในการแก้ปัญหา

### 2.10.1 การจัดแบ่งช่องสัญญาณในระบบโทรศัพท์เคลื่อนที่แบบไดนามิก

ปัญหาสำคัญของการสื่อสารในระบบโทรศัพท์เคลื่อนที่ คือ การจัดแบ่งช่องสัญญาณที่เหมาะสมในการรองรับการใช้งานที่มีการเปลี่ยนแปลงตลอดเวลาได้ โดยมีการแบ่งพื้นที่ในการให้บริการออกเป็นเซลล์ และช่องสัญญาณทั้งหมดของระบบจะถูกแบ่งย่อยออกเป็น channel โดยที่แต่ละ channel สามารถใช้งานได้พร้อมๆกันที่เซลล์ต่างกันเพื่อป้องกันการรบกวนของสัญญาณ ซึ่งระยะห่างน้อยที่สุดที่ channel เดียวกันสามารถใช้งานได้พร้อมๆกัน เรียกว่า "channel reuse constraint"

วัตถุประสงค์ในการจัดแบ่งช่องสัญญาณ คือ

1. การจัดแบ่ง channel ว่างในการรองรับการโทรเข้าเพื่อลดจำนวนการ block call
2. ลดปริมาณการ drop call เมื่อมีการ hand off (การที่เครื่องโทรศัพท์เคลื่อนที่ซึ่งกำลังสนทนากับโทรศัพท์เครื่องอื่น เคลื่อนที่จากเซลล์หนึ่งไปยังอีกเซลล์หนึ่งโดยไม่ขัดจังหวะการสนทนาเลย)

การจัดแบ่งช่องสัญญาณในระบบโทรศัพท์เคลื่อนที่ส่วนใหญ่เป็นแบบ "Fixed assignment (FA)" โดยมีการกำหนดช่องสัญญาณให้กับเซลล์แบบตายตัว ทำให้ไม่สามารถรองรับการใช้งานที่มีการเปลี่ยนแปลงตลอดเวลาได้

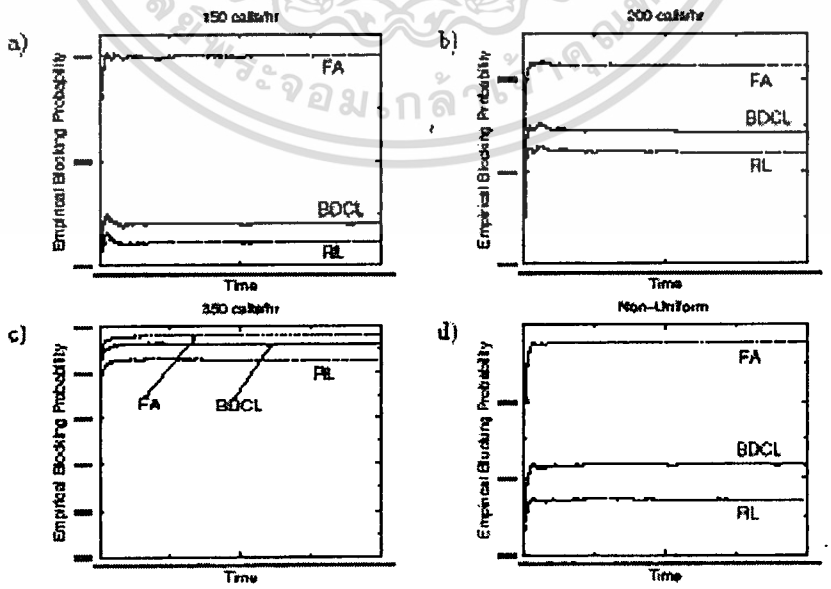
วิธีการที่ช่วยให้การจัดแบ่งช่องสัญญาณมีประสิทธิภาพที่ดีขึ้น คือ การจัดแบ่งช่องสัญญาณแบบไดนามิก วิธีที่นิยมใช้ คือ "Borrowing with Directional Channel Locking (BDCL)" โดยมีการเรียงลำดับหมายเลข channel จาก 1-N ซึ่งมีการแบ่งและกำหนด channel ให้กับเซลล์เหมือนกับวิธีของ FA ถ้ามี channel ว่างเมื่อมี call หมายเลข channel ที่ต่ำสุดจะถูกกำหนดให้กับ call นั้น แต่ถ้าในกรณีที่ไม่มี channel ว่างก็จะทำการยืม channel ที่มีหมายเลขสูงสุดจาก channel ใกล้เคียงทำให้เซลล์นั้นไม่สามารถถูกนำไปใช้งานได้เนื่องจากจะเกิดการรบกวนสัญญาณกัน

วิธีการของ BDCL ค่อนข้างซับซ้อน ในบทความนี้จะเป็นการเปรียบเทียบประสิทธิภาพของ Dynamic Channel allocation policy โดยใช้วิธีของ RL เปรียบเทียบกับวิธี FA และ BDCL

ในบทความนี้จะใช้วิธีการของ Dynamic Programming ในการแก้ปัญหา Dynamic channel allocation

การเปลี่ยนแปลง state (state transition) เกิดขึ้นเมื่อ

- มี channel ว่างเนื่องจาก call departure
- มีการโทรเข้าเกิดขึ้นในเซลล์และต้องการที่จะกำหนด channel ให้
- มีการ hand off




**รูปที่ 2.16** เปรียบเทียบ blocking probability ของ RL,FA และ BDCL ที่ call arrival rate ต่างๆ

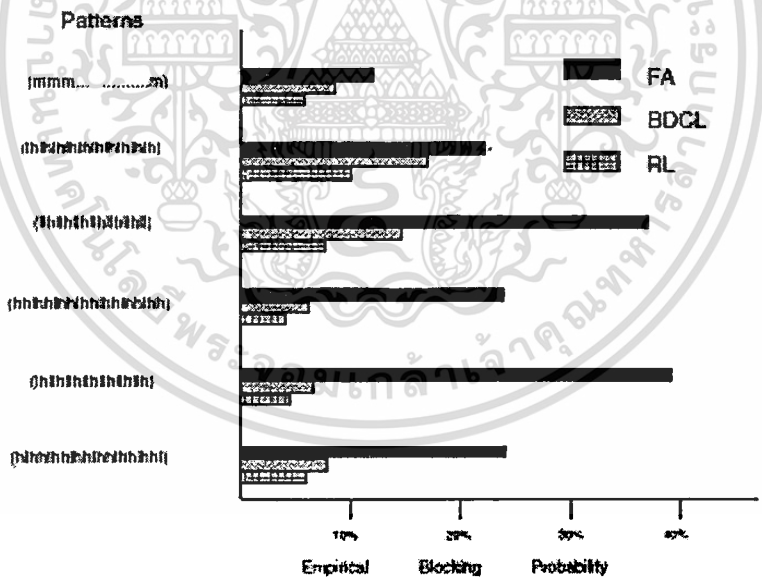
วิธีการของ Reinforcement Learning จะเน้นการจัดการเซลล์กับ call departure โดยจะมีการยอมรับการโทรเข้าก็ต่อเมื่อมี channel ว่าง

จุดประสงค์ในการเรียนรู้ policy คือ การพยายามทำให้จำนวนของ ongoing call มีค่ามากที่สุด ซึ่งจะเป็นการลดจำนวนการ block call โดยการเรียนรู้ optimal value function โดยใช้สมการ Bellman optimality equation ในการแก้ปัญหา

รูปแบบของ call arrival จะอยู่ในรูปแแบบของสมการ Poisson และช่วงเวลาของการ call จะอยู่ในรูปของ exponential distribution จากการทดลองใช้ 7*7 cellular array (70 channel) ซึ่งจะมีจำนวน configuration ทั้งหมด $70^{49}$ state

จากรูปที่ 2.16 แสดงโอกาสที่ call จะถูก block ขณะที่มีการ learning พบว่าวิธีการของ RL จะดีกว่าวิธของ BDCL และ FA ที่ call arrival rate ต่างๆ

ข้อแตกต่างระหว่าง RL และ BDCL คือ BDCL policy จะไม่คำนึงถึง traffic ของการ call แต่ RL policy จะมีการปรับปรุง policy ตาม traffic ของการ call




**รูปที่ 2.17** เปรียบเทียบ blocking probability ของ RL,FA และ BDCL ที่ call arrival rate = 120 calls/hr. โดยมีรูปแบบของ call arrival เป็นแบบ non-uniform

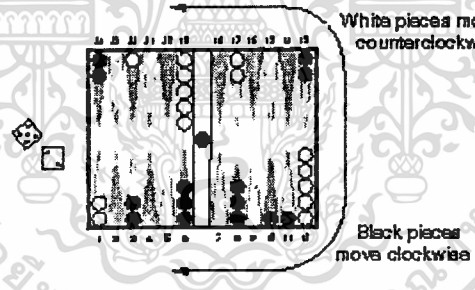
จากรูปที่ 2.17 แสดง bar-graph ของ asymptotic blocking probability โดยที่มีค่า arrival rate เท่ากับ 120 calls/hr. และรูปแบบของ call arrival เป็นแบบ non-uniform พบว่าวิธีการของ RL สามารถหา Dynamic channel allocation policy ได้ดีกว่าแบบ BDCL และ FA

สรุปได้ว่าวิธีการแก้ปัญหา Dynamic channel allocation ที่ใช้วิธีการของ Dynamic Programming สามารถแก้ปัญหา optimal control ได้ดีกว่าวิธีการของ BDCL และ FA

**2.10.2 TD-Gammon**

เป็นตัวอย่างของการนำ Reinforcement Learning ไปใช้ในการเล่นเกมส์ Backgammon โดย Gerry Tesauro ซึ่งใช้ความรู้ในการเล่นเกมส์เพียงเล็กน้อยเท่านั้น แต่สามารถเล่นได้ในระดับแชมเปี้ยน อัลกอริทึมที่ใช้ใน TD-Gammon คือ TD($\lambda$) และ nonlinear function approximation ซึ่งใช้ multi layer neural network ในการ train backpropagating TD error

ในการเล่น Backgammon จะมีตัวเดินสีขาวและดำอย่างละ 15 ตัว วางบนกระดานที่มี 24 ตำแหน่ง (เรียกว่า "point") แสดงดังรูปที่ 2.18




รูปที่ 2.18 Backgammon

จากรูปที่ 2.18 จะเป็นทางด้านตัวเดินสีขาวทอดลูกเต๋าได้5แต้มกับ 2 แต้ม หมายความว่าสีขาวสามารถเดินตัวใดตัวหนึ่งได้ 5 ช่องและตัวอื่นได้อีก 2 ช่อง(หรืออาจจะเดินตัวเดิมก็ได้) ตัวอย่าง เช่น สามารถเดิน 2 ตัวจากตำแหน่ง12 ,1 ตัวจากตำแหน่ง17 และ 1 ตัวที่ตำแหน่ง14 โดยมีจุดประสงค์คือ การพยายามทำให้สีขาวไปอยู่ที่ quarant สุดท้ายคือตำแหน่ง 19-24 และเอาตัวเดินออกจากบอร์ด โดยมีกติกาว่าใครที่สามารถเอาตัวเดินของตนเองออกจากกระดานได้หมดก่อนจะเป็นผู้ชนะ ซึ่งความซับซ้อนของเกมส์อยู่ที่การ interact กันระหว่างตัวเดินที่มีทิศทางในการเดินที่แตกต่างกัน ยกตัวอย่างจากรูป ถ้าเป็นคราวของสีดำ ถ้าได้แต้ม2 โดยการเดินจากตำแหน่ง24 ไปยังตำแหน่ง 22 ซึ่งจะชนกับตัวเดินสีขาวที่ตำแหน่ง22 ในกรณีนี้เราจะย้ายตัวเดินสีดำไปไว้ตำแหน่งตรงกลางของกระดาน (เรียกว่า "bar") และต้องเริ่มต้นใหม่ ถ้ามีตัวเดิน 2 ตัวบนตำแหน่งเดียวกัน จะทำให้คู่

ต่อสู้ไม่สามารถเดินไปยังตำแหน่งนั้นได้ ฉะนั้นสีขาวไม่สามารถใช้แต้ม 5-2 ในการเดินในตำแหน่งที่1 เพราะว่าสีดำอาจจะจองตำแหน่งนั้นได้ ซึ่งการ block ตำแหน่งคู่ต่อสู้เป็นกลยุทธ์หนึ่งที่ใช้ในการเล่นเกมส์

Backgammon จะมีตัวเดินทั้งสิ้น 30 ตัวและมีตำแหน่งของกระดาน 26 ตำแหน่ง ดังนั้นตำแหน่งการเดินที่เป็นไปได้มีจำนวนมหาศาล และยังขึ้นกับการทอดลูกเต๋าอีกด้วย ซึ่งไม่สามารถใช้วิธีของ heuristic search ได้

TD learning สามารถนำไปประยุกต์ใช้กับเกมส์ได้ ซึ่งจะเป็นลำดับของการเดินและตำแหน่ง จนกระทั่งเกมส์สิ้นสุด ผลลัพธ์ของเกมส์จะเป็น final reward แต่เนื่องจากจำนวน state มีจำนวนมากจึงไม่สามารถใช้การ lookup table ได้โดย TD-Gammon ใช้ multilayer neural network ในการ estimate ค่า $V_t(s)$ ของตำแหน่งของกระดาน โดย reward มีค่าเป็น 1 เมื่อชนะ

ในการใช้ neural network คำนวณจะใช้ signal จากแต่ละ input unit ซึ่งจะคูณด้วยค่า weight และทำการ sum ที่ hidden unit โดยที่ output : $h(j)$ ของ hidden unit j จะเป็น nonlinear sigmoid function ของ weighted sum ตามสมการ

$$h(j) = \sigma(\sum_t w_{ij}\phi(i)) = \frac{1}{1+e^{-\sum_i w_{ij}\phi(i)}} \qquad \ldots(2.35)$$

เมื่อ

$\phi(i)$ คือ ค่าของ input ตัวที่ i

$W_{ij}$ คือ weight ที่ connect กับ hidden unit ตัวที่ j

โดยผลลัพธ์ของ sigmoid function จะอยู่ระหว่าง 0-1 แต่ละ connection ของ hidden unit กับ output unit จะมี weight ที่ต่างกัน

TD-Gammon ใช้รูปแบบของ Gradient-descend ของ TD($\lambda$) ซึ่งคำนวณ error จาก backpropagation algorithm ซึ่งมีกฎการอัพเดทตามสมการ 2.36

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha[r_{t+1} - \gamma V_t(s_{t+1}) - V_t(s_t)]\vec{e}_t \qquad \ldots(2.36)$$

เมื่อ

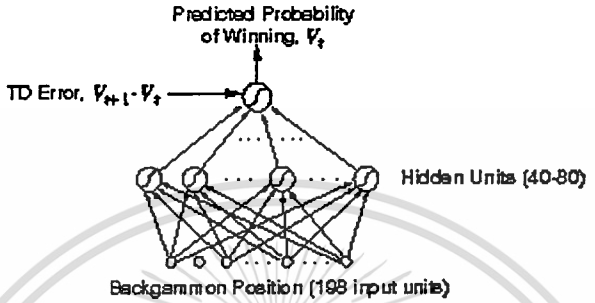
$\vec{\theta}_t$ คือ vector ของ weight ของ network

$\vec{e}_t$ คือ vector ของ eligibility trace

แต่ละ component ของ $\vec{\theta}_t$ จะอัพเดทโดยสมการ

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t) \qquad \ldots(2.37)$$

ซึ่ง gradient คำนวณโดยใช้วิธีของ backpropagation โดย Backgammon จะกำหนดให้ reward มีค่าเป็น 0 ยกเว้นถ้าชนะจะมีค่าเป็น 1 และมี TD error คือ $V_t(S_{t+1})-V_t(S_t)$ ซึ่งแสดงดังรูปที่ 2.19




รูปที่ 2.19 neural network ที่ใช้ใน TD-Gammon

ในการเรียนรู้ Backgammon ใช้การเรียนรู้แบบ self-play ในการเลือกวิธีการเดินจะพิจารณาจากการทอดลูกเต๋าและตำแหน่งที่สอดคล้องกัน โดยใช้ neural network ในการ estimate ค่าของแต่ละ state ซึ่งจะเลือกตำแหน่งที่มี estimate value สูงสุด ในแต่ละเกมส์จะจัดเป็น episode ที่มีลำดับของตำแหน่งเป็น $S_0, S_1, S_2, \ldots$ Tesauro ประยุกต์กฎของ nonlinear TD ตามสมการที่ 2.36 ในการ increment หลังจากที่มีการเดินในแต่ละครั้ง

หลังจากเล่นไปได้ 300,000 เกมส์ TD-Gammon0.0 สามารถเล่นได้ในระดับของ backgammon computer program ที่ดีที่สุดในขณะนั้น ซึ่งมีการใช้ backgammon knowledge เช่น โปรแกรม "Neurogammon" ที่ใช้ neural network ที่มีการ train ด้วยผู้เชี่ยวชาญ ในขณะที่ TD-Gammon0.0 เริ่มต้นจากการที่ไม่มีความรู้ทางด้าน backgammon เลย

ต่อมา TD-Gammon0.0 ได้มีการพัฒนาเป็น TD-Gammon 1.0 ,2.0 และ 3.0 โดยเพิ่มประสิทธิภาพในการวางแผนมากขึ้นจนในTD-Gammon3.0 สามารถเอาชนะแชมเปี้ยนระดับโลกได้ วิวัฒนาการของ TD-Gammon แสดงดังตารางที่ 2.2

| Program | Hidden Units | Training Games | Opponents | Results |
|---|---|---|---|---|
| TD-Gam0.0 | 40 | 300,000 | Other programs | Tied for best |
| TD-Gam1.0 | 80 | 300,000 | Robertie,Magriel | -13 points/ 51 games |
| TD-Gam2.0 | 40 | 800,000 | Various Grandmasters | -7 points/ 38 games |
| TD-Gam2.1 | 80 | 1,500,000 | Robertie | -1 points/ 40 games |
| TD-Gam3.0 | 80 | 1,500,000 | Kazaros | +6 points/ 20 games |

ตารางที่ 2.2 วิวัฒนาการของ TD-Gammon

# บทที่ 3

## การใช้ Reinforcement Learning ในการค้นหาเส้นทางในระบบเครือข่าย

เนื่องจากระบบเครือข่ายส่วนใหญ่จะมีการรับส่งข้อมูลในรูปของ packet ซึ่งแต่ละโหนดในระบบเครือข่ายจะต้องตัดสินใจได้ว่าจะส่งต่อ packet ไปให้โหนดใดเพื่อให้ packet ถูกส่งไปถึงปลายทางได้ ซึ่งกระบวนการดังกล่าวเรียกว่า "Packet Routing"

ปัจจัยที่มีผลต่อเวลาที่ใช้ในการส่ง packet (delivery time) คือ เวลาที่ packet จะต้องรออยู่ในคิว และ policy ของ Packet Routing ซึ่งถ้าไม่มี policy ที่มีประสิทธิภาพที่ดีพอก็อาจจะส่งผลทำให้ระบบเครือข่ายเกิดความล่าช้าได้

ในโครงงานนี้ได้นำเอา Reinforcement Learning มาใช้ทำ Packet Routing policy เพื่อให้แต่ละโหนดในระบบเครือข่ายสามารถแลกเปลี่ยนข้อมูลซึ่งกันและกัน เพื่อช่วยปรับปรุงประสิทธิภาพในการค้นหาเส้นทางในระบบเครือข่ายให้ดีขึ้น ซึ่งอัลกอริธึมมีสามารถในการเปลี่ยนแปลงตามระบบเครือข่ายที่มีการเปลี่ยนแปลงอยู่ตลอดเวลาได้

จุดประสงค์หลักของอัลกอริธึม คือ พยายามให้เวลาในการส่ง packet (delivery time) มีค่าน้อยที่สุดด้วย local information โดยเปรียบเทียบกับอัลกอริธึม static shortest-path (Bellman Ford algorithm) ภายใต้สภาวะโหลดต่างๆ โดยที่อัลกอริธึมที่ใช้จะไม่รู้เกี่ยวกับสภาวะแวดล้อม, network topology และ input rate ซึ่งจะมีลักษณะการทำงานแบบ online-policy

### 3.1 การนิยามปัญหาของ Network routing ด้วย Reinforcement Learning

วัตถุประสงค์ คือ การพิจารณาว่าจะส่ง packet ไปให้โหนดไหนเพื่อให้ไปถึงปลายทางได้เร็วที่สุดเท่าที่จะเป็นไปได้ ซึ่งการตัดสินใจจะใช้ข้อมูลที่มีอยู่ในโหนดนั้นๆ

ยกตัวอย่างเช่น ถ้าต้องการส่ง packet จากโหนด $x$ ไปยังโหนดปลายทาง $d$ สามารถนิยามปัญหาของ Network routing ด้วย Reinforcement learning ได้ดังนี้

#### 3.1.1 Reward

ค่า Reward สามารถนิยามให้อยู่ในรูปของค่า estimate ของเวลาในการส่ง packetจากโหนด $x$ ไปยังโหนดปลายทาง $d$ โดยส่งผ่านโหนด neighbor $y$ ซึ่งนิยามด้วยฟังก์ชัน $Q_x(y,d)$

โดยแต่ละโหนดในระบบเครือข่ายจะต้องพยายามให้ $Q_x(y,d)$ มีค่าน้อยที่สุด ถ้าทำให้ค่า estimate $Q_x(y,d)$ มีค่าน้อยลง ค่า reward ก็จะมีค่ามากขึ้น

**3.1.2 State**

State ประกอบด้วย

- โหนดที่กำลังรับ packet ซึ่งในแต่ละโหนดจะมี agent ในการแก้ปัญหาในการค้นหาเส้นทาง
- โหนดปลายทาง

โหนด $x$ จะเก็บค่า estimate ของเวลาในการส่ง packet ของทุกๆโหนด neighbor และโหนดปลายทาง $(y,d)$ ว่า packet จะถูกส่งไปถึงปลายทาง $d$ โดยใช้เวลาเท่าไหร่ ถ้าส่งผ่านโหนด neighbor $y$

ค่า estimate จะแทนด้วย $Q(y,d)$ ในที่นี้ โหนดปลายทาง $d$ จะหมายถึง state และการเลือกโหนด neighbor จะเป็น action

**3.1.3 Actions**

การเลือก action คือ การเลือกโหนด neighbor และส่ง packet ไปให้โหนดนั้น โดยที่โหนด $x$ จะทำการส่ง packet ไปให้โหนด neighbor ที่มีค่า estimate ถึงปลายทาง $d$ ที่น้อยที่สุดตามสมการ 3.1

$$\bar{y} = \arg\min_{y \in \text{ neighbor of } x}[Q_x(y,d)] \qquad \ldots(3.1)$$

แต่มีในบางกรณีที่มีการกระทำ action แบบ random ซึ่งถูกกำหนดด้วยพารามิเตอร์ Epsilon-greedy action value ($\varepsilon$) มีค่าระหว่าง 0-1 ซึ่งเป็นการกำหนดว่าโอกาสในการที่ต้องการให้เลือกกระทำ action แบบ random เป็นเท่าไหร่

**3.1.4 การอัพเดทค่า Estimates**

$Q_x(\bar{y},d)$ คือ ค่า estimate ของโหนด $x$ ที่ใช้เวลาในการส่ง packet ไปยังปลายทาง $d$ ที่น้อยที่สุด โดยผ่านโหนด $\bar{y}$ จากนั้นโหนด $\bar{y}$ ก็จะทำการเลือกโหนด neighbor $\bar{z}$ ที่มี time ไปยังปลายทาง $d$ ที่น้อยที่สุดเช่นกัน ซึ่งแสดงตามสมการ 3.2 ...(3.2)

$$\bar{z} = \arg\min_{z \in \text{ neighbor of } y}[Q_y(z,d)]$$

ค่า estimate ที่ได้จะถูกส่งกลับไปยังโหนด $x$ ด้วยช่องสัญญาณที่ใช้สำหรับการควบคุม (control channel) เพื่อไม่ให้ส่งผลกระทบต่อประสิทธิภาพในการรับส่ง packet ในระบบเครือข่าย

โหนด $x$ สามารถอัพเดทค่า estimate ของ delivery time ไปยังปลายทาง $d$ โดยผ่านโหนด $\bar{y}$ โดยการรวมผลลัพธ์ที่ได้มาใหม่กับ knowledge เดิมที่มีอยู่ จะได้ตามสมการ 3.3

$$Q_x(\bar{y},d) \leftarrow Q_x(\bar{y},d) + \alpha[\overbrace{Q_{\bar{y}}(\bar{z},d) + t + q}^{\text{new estimate}} - \overbrace{Q_x(\bar{y},d)}^{\text{old estimate}}] \quad \ldots(3.3)$$

โดย

$\alpha$ คือ learning rate เป็นค่าคงที่มีค่าอยู่ระหว่าง 0-1

$t$ คือ transmission time ระหว่างโหนด $x$ และโหนด $\bar{y}$

$q$ คือ เวลาที่ packet รออยู่ในคิว

**3.2 อัลกอริธึม Q-routing**

อัลกอริธึม Q-routing เป็นอัลกอริธึมที่พัฒนาจากอัลกอริธึม Temporal-Difference Learning ซึ่งพัฒนาขึ้นมาเพื่อให้เหมาะสมกับการนำไปแก้ปัญหาการค้นหาเส้นทางในระบบเครือข่าย ดังนั้นในโครงงานนี้จึงได้นำอัลกอริธึม Q-routing มาประยุกต์ใช้ในการสร้างแบบจำลองในการค้นหาเส้นทางในระบบเครือข่าย ซึ่งสามารถทำได้โดยกำหนดให้แต่ละโหนดในระบบเครือข่ายมีโมดูลในการค้นหาเส้นทาง โดยที่โมดูลมีหน้าที่ต่างๆ ดังนี้

- ทำหน้าที่ส่ง packet ไปยังโหนด neighbor โดยใช้เวลาในการส่ง packet (delivery time) ให้น้อยที่สุด
- เมื่อ packet มาถึงโหนด โหนดก็จะส่งค่า delivery time ที่เหลือ ($Q(z,d)$)กลับไปยังโหนดที่ส่ง packet
- เมื่อโหนดได้รับการตอบกลับ จากโหนด neighbor ก็จะทำการอัพเดทค่า estimate ของโหนด neighbor นั้น ($Q(y,d)$)ด้วยค่า estimate ใหม่ที่ได้รับ($Q(z,d)$) ซึ่งเป็นไปตามสมการ 3.3

ซึ่งค่า estimate จะถูกจัดเก็บอยู่ในรูปแบบของการ lookup table $Q(y,d)$ เพื่อเก็บค่า estimate total delivery time ไปจนถึงปลายทาง $d$ โดยผ่านโหนด neighbor $y$ ซึ่งจะทำเช่นนี้กับทุกโหนดที่มีการส่ง packet จึงทำให้แต่ละโหนดในระบบเครือข่ายสามารถเรียนรู้ได้ว่าควรจะส่งต่อ packet ไปยังโหนด neighbor ใดจึงจะใช้เวลาในการส่ง packet น้อยที่สุด

การทำงานในแต่ละโมดูลประกอบด้วย 2 ขั้นตอน คือ

**3.2.1 ขั้นตอนการส่ง packet**

1. เลือก packet จากคิว (โดยใช้ policy FIFO) โดยให้ $d$ คือโหนดปลายทางที่จะส่ง packet ไป
2. เลือกโหนด $\bar{y}$ ที่มีค่า $Q(y,d)$ ต่ำที่สุดจากทุกๆ โหนด neighbor $y$ ของโหนดปัจจุบัน
3. รอการตอบกลับ $Q(z,d)$ จากโหนด $y$
4. ทำการอัพเดทค่า $Q(\bar{y},d)$ ของโหนดปัจจุบันโดยใช้ค่า estimate ใหม่จากโหนด $\bar{y}$

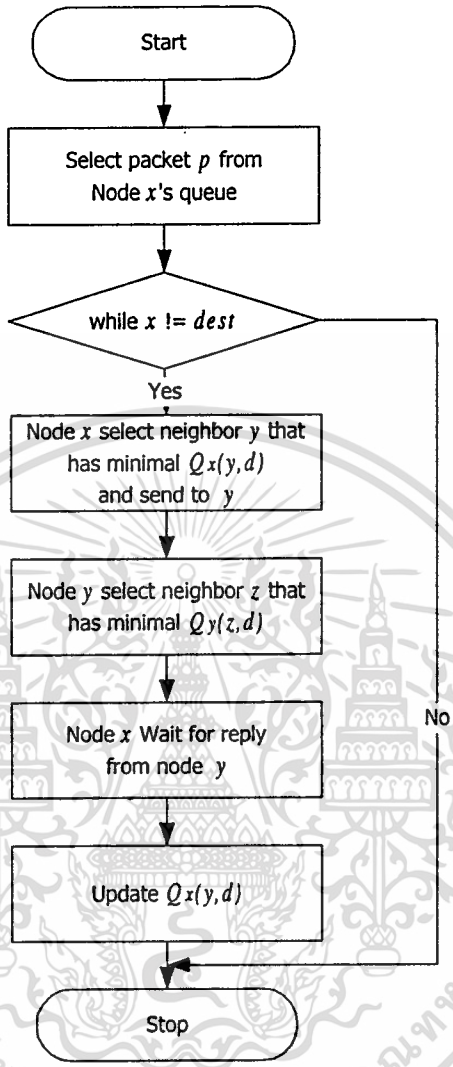
$$Q(\bar{y},d) = Q(\bar{y},d) + \alpha[Q_{\bar{y}}(\bar{z},d) + t + q - Q(\bar{y},d)]$$

**3.2.2 ขั้นตอนการรับ packet**

1. รับ packet $p$ จากโหนด $s$
2. เลือกโหนด $\bar{z}$ ที่มีค่า $Q(z,d)$ ต่ำที่สุดจากทุกๆ neighbor $z$ ของโหนด $x$
3. ส่งค่า $Q(\bar{z},d)$ กลับไปยังโหนด $s$
4. ใส่ packet $p$ ลงในคิว

ในขั้นตอนของการรับ packet จะไปอินเทอร์รัพการทำงานของขั้นตอนในการส่ง เพื่อลดเวลาที่แต่ละโหนดต้องรอ reply โดยที่ reply จะถูกส่งก่อนที่ packet จะถูกใส่ลงในคิว

จากขั้นตอนดังกล่าวสามารถแสดงไดอะแกรมการทำงานของอัลกอริธึม Q-routing ได้ดังนี้




รูปที่ 3.1 แสดงไดอะแกรมการทำงานของอัลกอริธึม Q-routing

อัลกอริธึมของ Reinforcement Learning แบบ Q-routing ที่ใช้ในการพัฒนาโปรแกรมสำหรับโครงงานนี้มีขั้นตอน ดังนี้

<u>ขั้นตอนที่ 1</u> ทำการวนลูป link ของโหนดต้นทาง(โหนด $x$) เพื่อคำนวณหา link ที่มีค่า estimate ของ delivery time ($Q_x(y,d)$) ในการส่ง packet ไปยังโหนดปลายทาง(โหนด $d$) ที่ต่ำที่สุด ซึ่งเป็นไปตามสมการ

$$link = \min[Q_x(y,d)] \quad \ldots(3.4)$$

โดย

$Q_x(y,d)$ คือ ค่า estimate ของโหนด $x$ ที่มี delivery time ไปยังโหนดปลายทาง $d$ ที่น้อยที่สุด โดยส่ง packet ผ่านโหนด $y$

ขั้นตอนที่ 2 จากขั้นตอนที่ 1 ก็จะรู้โหนด neighbor $y$ ที่จะส่ง packet จากนั้นทำการส่ง packet ไปยังโหนด neighbor $y$

ขั้นตอนที่ 3 รอการตอบกลับ (reply) จากโหนด neighbor $y$ ซึ่งมีสมการ คือ

$$\text{reply} = Q_y(z,d) \qquad \ldots(3.5)$$

โดย

$Q_y(z,d)$ คือ ค่า estimate ของโหนด $y$ ที่มี delivery time ไปยังโหนดปลายทาง $d$ ที่น้อยที่สุด โดยส่ง packet ผ่านโหนด $z$

ขั้นตอนที่ 4 ทำการอัพเดทค่า estimate $Q_x(y,d)$ ของ link ที่ส่ง packet ไป ซึ่งเป็นไปตามสมการ

$$Q_x(y,d) = Q_x(y,d) + \alpha[\text{reply} + t + q - Q_x(y,d)] \qquad \ldots(3.6)$$

โดย

$\alpha$ คือ learning rate เป็นค่าคงที่มีค่าอยู่ระหว่าง 0-1

$reply$ คือ ค่า estimate ของโหนด $y$ ที่มี delivery time ไปยังโหนดปลายทาง $d$ ที่น้อยที่สุด โดยส่ง packet ผ่านโหนด $z$ (ตามสมการที่ 3.5)

$t$ คือ ค่า transmission time ระหว่างโหนด $x$ และโหนด $y$

$q$ คือ เวลาที่ packet รออยู่ในคิวของโหนด $x$ (queueing delay)

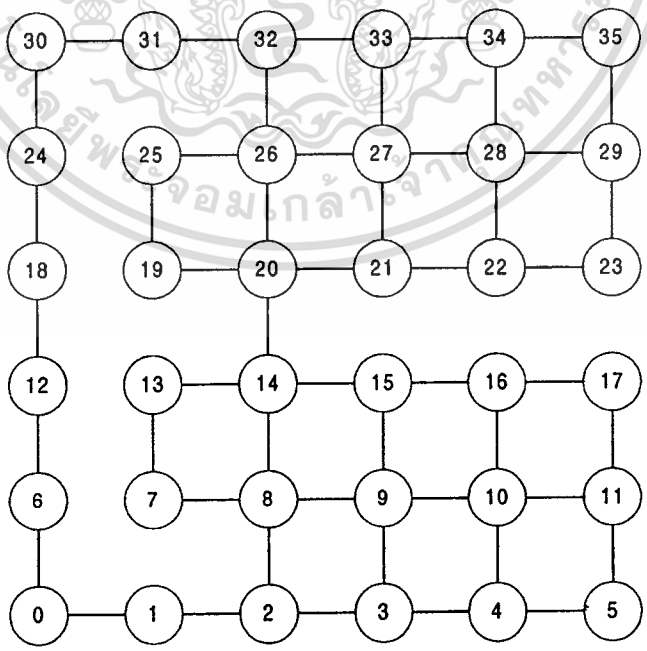
# บทที่ 4
## การพัฒนาโปรแกรม

แบบจำลองการค้นหาเส้นทางในระบบเครือข่าย ผู้พัฒนาได้ใช้โปรแกรม Jbuilder4.0 เนื่องจากเป็นโปรแกรมที่มีคุณสมบัติตามที่ผู้พัฒนาต้องการ กล่าวคือ

1. เป็นเครื่องมือพัฒนาแอปพลิเคชั่นแบบวิชวลโปรแกรมมิ่ง (Visual Programming) ซึ่งทำให้ผู้พัฒนาสามารถเห็นผลลัพธ์การทำงานไปพร้อมๆกับการสร้างแอปพลิเคชั่น ทำให้ช่วยลดเวลาในการสร้างแอปพลิเคชั่น
2. มีความสามารถในการสร้าง Graphic User Interface
3. มีความสามารถในการพัฒนาเป็นแอ็พเพลต (Applet) ทำให้สามารถแสดงผลบนบราวเซอร์(Browser) ได้

### 4.1 ขั้นตอนการทำงานของระบบ

แบบจำลองการค้นหาเส้นทางในระบบเครือข่ายที่ใช้จะทดลองกับ Network topology แบบ Gird ขนาด 6*6 ซึ่งแสดงดังรูปที่ 4.1




**รูปที่ 4.1** Network topology ขนาด 6*6

เงื่อนไขของโปรแกรมจำลองการค้นหาเส้นทาง

- ในการเลือก packet ที่จะทำการส่งจะใช้ policy FIFO (First In Fist Out) คือ packet ที่รออยู่ในคิวก่อน จะมีสิทธิ์ในการส่งออกไปก่อน
- จำนวนข้อมูลที่ส่งไปบนระบบเครือข่าย = 6,000 packet
- เวลาที่ใช้ในการส่ง packet ไปยังปลายทาง (Delivery time) จะคำนวณจากเวลาที่ใช้ในการส่ง packet ไปยังโหนดข้างเคียง (Transmission time) ซึ่งกำหนดให้มีค่าเท่ากับ 1 เวลาที่ใช้ในการให้บริการในการเลือกเส้นทางของแต่ละ packet (Service time) ซึ่งมีค่าเท่ากับ 0.1 และเวลาที่ packet รออยู่ในคิว (Queueing delay) ซึ่งสามารถคำนวณได้โดย

$$\text{Queueing delay} = 0.1 * \text{จำนวน packet ที่รออยู่ในคิว} \quad \ldots(4.1)$$

ดังนั้นเวลาที่ใช้ในการส่ง packet จากโหนดใดๆไปยังโหนดข้างเคียงมีค่าเท่ากับผลรวมของ Transmission time , Service time และ Queueing delay

โดยโปรแกรมจะจำลองการค้นหาเส้นทางของระบบเครือข่ายในสภาวะ Low load และ High load ซึ่งในสภาวะ Low load จะทำการทดลองโดยการส่ง packet จากโหนดขวาล่างไปยังโหนดขวาบน คือ ส่งจากโหนด 5 ไปยังโหนด 35

ในสภาวะ High load จะทำการทดลองโดยการส่ง packet เป็นแบบ random โดยข้อมูล 1,000 packet แรกจะส่งจากโหนดต้นทางคือโหนด 5 ไปยังโหนด 35 และข้อมูลอีก 5,000 packet จะเป็นการส่งโดยกำหนดให้โหนดต้นทางอยู่ด้านขวาล่าง (โหนด 4,8,9,11,14) ส่วนโหนดปลายทางจะเป็นด้านบน(โหนด30,31,32,33,34,35) เพื่อให้เกิดความลับดั่งของเส้นทางที่จะส่ง packet ซึ่งมีรายละเอียดดังนี้

- packet ที่ 0-999 จะทำการส่งจากโหนด 5 ไปยังโหนดปลายทาง 35
- packet ที่ 1,000-1,999 จะทำการส่งจากโหนด 4 ไปยังโหนดปลายทาง {33,34,35}
- packet ที่ 2,000-2,999 จะทำการส่งจากโหนด 14 ไปยังโหนดปลายทาง {33,34,35}
- packet ที่ 3,000-3,999 จะทำการส่งจากโหนด 9 ไปยังโหนดปลายทาง {30,31,32}
- packet ที่ 4,000-4,999 จะทำการส่งจากโหนด 8 ไปยังโหนดปลายทาง {30,31,32}
- packet ที่ 5,000-5,999 จะทำการส่งจากโหนด 11 ไปยังโหนดปลายทาง {30,31,32}

แบบจำลองการค้นหาเส้นทางในระบบเครือข่าย มีขั้นตอนการทำงานดังนี้

- ผู้ใช้เลือกรูปแบบสภาวะโหลดที่ต้องการทดสอบ ซึ่งมี 2 รูปแบบ คือ Low Load และ High Load
- ผู้ใช้สามารถเลือกตั้งค่าพารามิเตอร์เริ่มต้นต่างๆของระบบ ได้แก่ อัตราการเรียนรู้ (Learning rate) ซึ่งมีค่าอยู่ระหว่าง 0 ถึง 1 โดยที่ค่าดีฟอลต์ของอัตราการเรียนรู้จะมีค่าเท่ากับ 0.7 และ Epsilon –greedy action value ซึ่งมีค่าอยู่ระหว่าง 0 ถึง 1 โดยที่ค่าดีฟอลต์ของอัตราการเรียนรู้จะมีค่าเท่ากับ 0 (เลือก action แบบ greedy เพื่อให้ได้ค่า reward มากที่สุด)
- เมื่อผู้ใช้กำหนดค่าพารามิเตอร์เรียบร้อยแล้ว ก็สั่งให้เริ่มทำการประมวลผล เพื่อแสดงรูปการค้นหาเส้นทางของแต่ละ packet
- เมื่อได้ทำการส่ง packet จนครบแล้ว โปรแกรมจะนำเวลาที่ใช้ในการส่ง packet (Delivery time) มาทำการวาดกราฟเพื่อแสดงผลการเปรียบเทียบระหว่างเวลาที่ใช้ในการส่ง packet ระหว่าง Q-Routing และ Bellman Ford

จากขั้นตอนดังกล่าวข้างต้น ทำให้ผู้พัฒนาสามารถนำไปพัฒนาแบบจำลองการค้นหาเส้นทางในระบบเครือข่ายได้ต่อไป

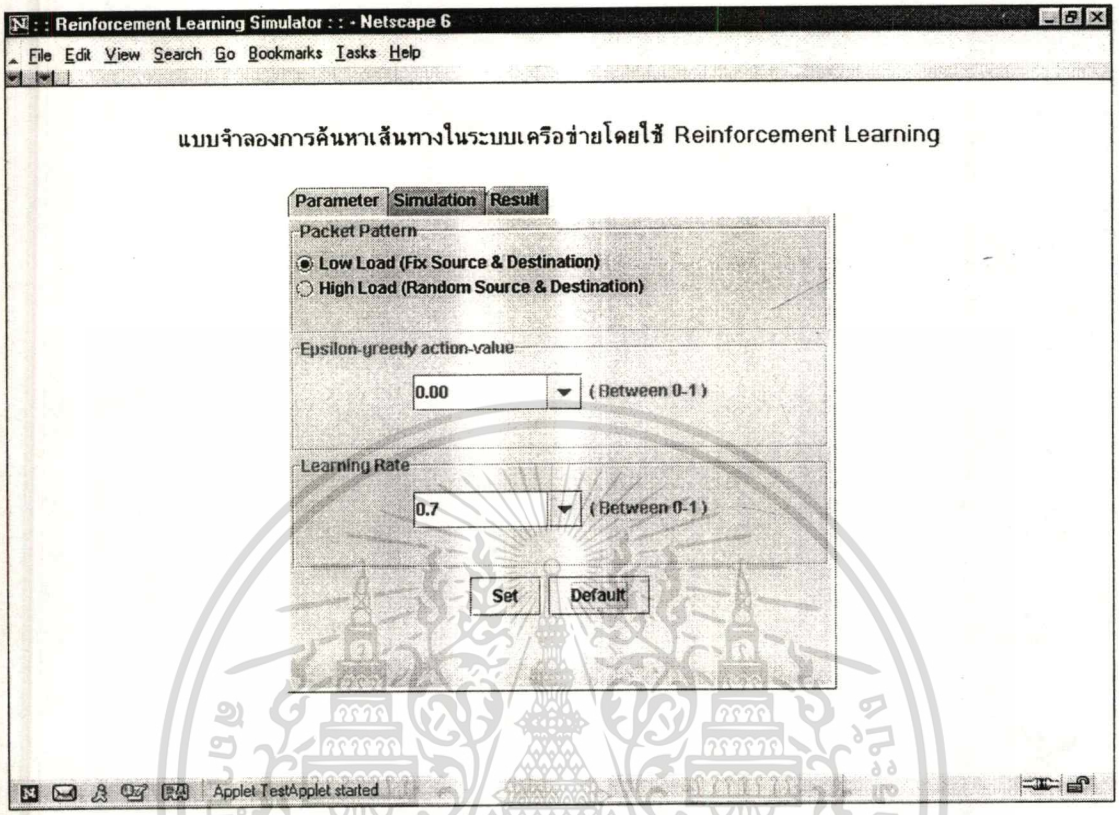
**4.2 การออกแบบหน้าจอของแบบจำลองการค้นหาเส้นทางในระบบเครือข่าย**

ในการออกแบบหน้าจอที่ใช้ในการแสดงผล ผู้พัฒนาได้ทำการออกแบบส่วนของการติดต่อกับผู้ใช้และส่วนที่ใช้ในการแสดงผล โดยมีรายละเอียดดังนี้

ในรูป 4.2 แสดงหน้าจอหลักที่ใช้ในการติดต่อกับผู้ใช้ ซึ่งประกอบด้วย 3 ส่วน คือ

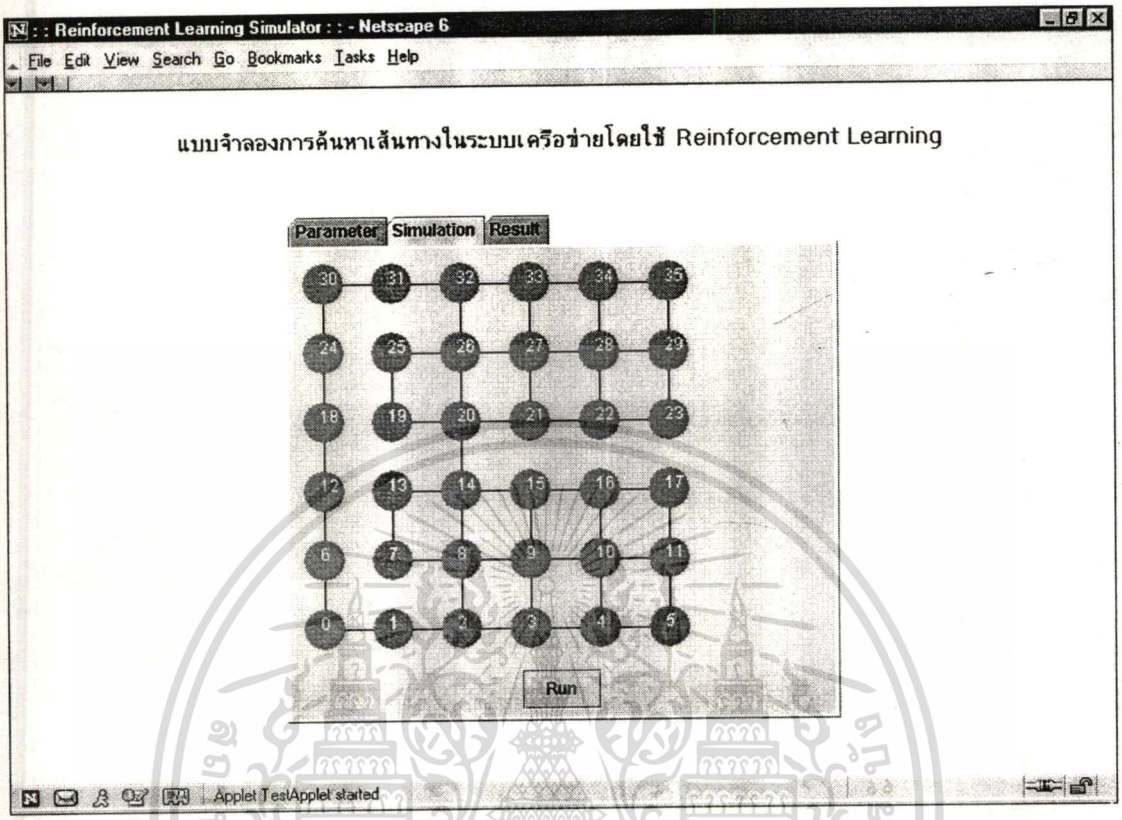
1. Packet Pattern ผู้ใช้สามารถเลือกรูปแบบสภาวะโหลดที่ต้องการรัน ซึ่งสามารถเลือกได้ 2 รูปแบบ คือ แบบ Low load และแบบ High load
2. Epsilon-greedy action value ผู้ใช้สามารถกำหนดค่า Epsilon เพื่อกำหนดค่าความน่าจะเป็นในการเลือกกระทำ action แบบ random ซึ่งถ้ามีค่าเป็น 0 หมายถึงจะทำการเลือก action แบบ greedy โดยมีค่าอยู่ระหว่าง 0 ถึง 1
3. Learning Rate ผู้ใช้สามารถกำหนดอัตราการเรียนรู้ (Learning Rate) ได้ โดยมีค่าอยู่ระหว่าง 0 ถึง 1

จากรูป 4.2 ถ้าผู้ใช้กดปุ่ม Default ก็จะเป็นการกำหนดค่าพารามิเตอร์ของ Epsilon-greedy action value เท่ากับ 0 และค่า Learning rate เท่ากับ 0.7



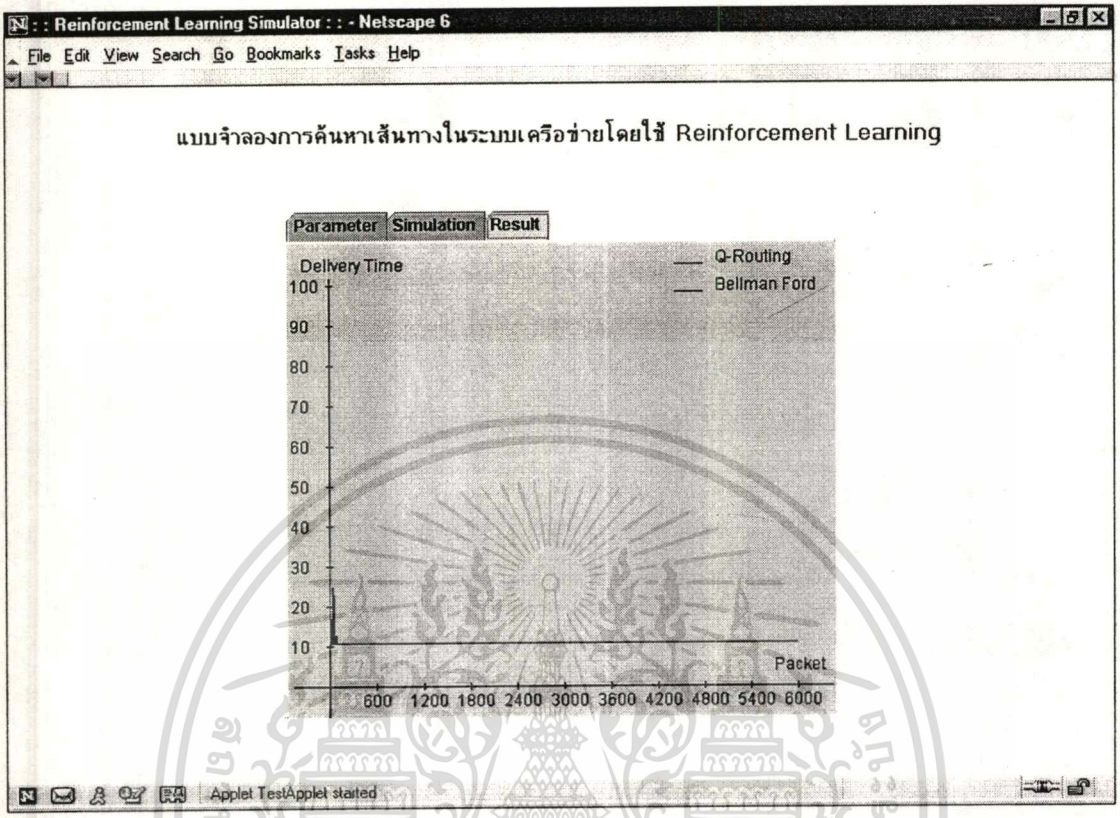

รูปที่ 4.2 แสดงหน้าจอหลักของระบบ

หลังจากที่ผู้ใช้ได้เลือกรูปแบบสภาวะโหลด และกำหนดค่า Epsilon-greedy action value , ค่าอัตราการเรียนรู้เรียบร้อยแล้วให้กดปุ่ม Set เพื่อส่งค่าพารามิเตอร์ให้กับโปรแกรม จากนั้นผู้ใช้สามารถที่จะเริ่มการจำลองการค้นหาเส้นทางในระบบเครือข่ายได้โดยการเลือกไปที่แท็บ Simulation แล้วกดปุ่ม Run ดังรูปที่ 4.3




รูปที่ 4.3 แสดงหน้าจอสำหรับเริ่มการจำลองการค้นหาเส้นทางในระบบเครือข่าย

หลังจากระบบทำงานเสร็จ ผู้ใช้สามารถเลือกไปที่แท็บของ Result เพื่อดูผลลัพธ์ของเวลาที่ใช้ในการส่ง packet (Delivery time) ดังรูปที่ 4.4 ซึ่งแสดงกราฟของเวลาที่ใช้ในการส่ง packet โดยเปรียบเทียบระหว่างอัลกอริธึม Q-Routing และ Bellman Ford




รูปที่ 4.4 แสดงตัวอย่างกราฟของเวลาที่ใช้ในการส่ง packet โดยเปรียบเทียบระหว่าง Q-Routing และ Bellman Ford

**4.3 ผลการทดลอง**

เมื่อได้แบบจำลองการค้นหาเส้นทางตามที่ต้องการแล้ว ผู้พัฒนาได้ทำการศึกษาถึงผลการค้นหาเส้นทางของ Q-Routing และ Bellman Ford โดยทำการเปรียบเทียบเวลาที่ใช้ในการส่ง packet (Delivery time) ในขณะที่ระบบเครือข่ายอยู่ในสภาวะ Low load และ High load

จากการทดลองพบว่า ในสภาวะ Low load เวลาที่ packet รออยู่ในคิว (Queueing delay) จะมีค่าน้อยเนื่องจากเป็นการรับ-ส่ง packet จากโหนดต้นทางเดียว คือ โหนด 5 ไปยังโหนดปลายทางเดียว คือ โหนด 35 จึงทำให้ Queueing delay มีค่าน้อยเมื่อเทียบกับ Transmission time ดังนั้นเวลาที่มีผลต่อเวลาในการส่ง packet (Delivery time) ในสภาวะ Low load คือ Transmission time

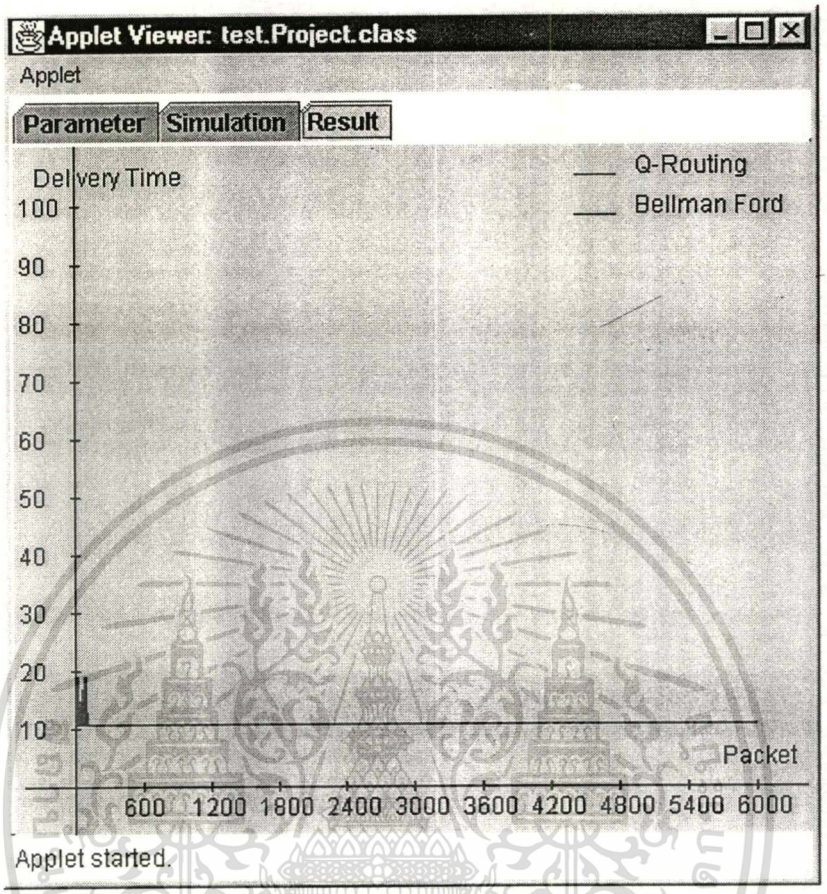
ซึ่งจากผลการทดลองจะพบว่า Delivery time ของ Q-Routing จะมีค่ามากในช่วงแรก เนื่องจากอยู่ในช่วงของการเรียนรู้ เนื่องจากไม่ได้มีการให้ information ใดๆเลยกับ Agent แต่หลังจากนั้นเวลาที่ใช้ในการส่ง packet จะมีค่าเท่ากับเวลาที่ใช้ในการส่ง packet ด้วยอัลกอริธึมของ Bellman

Ford ดังแสดงในตารางที่ 4.1 และสามารถแสดงกราฟเปรียบเทียบเวลาที่ใช้ของทั้ง 2 อัลกอริธึมได้ดังรูปที่ 4.5

จากตารางที่ 4.1 และรูปที่ 4.5 จะเห็นว่าเวลาที่ใช้ในการส่ง packet ของอัลกอริธึมจะมีค่าเท่าเดิมตลอด เนื่องจากอัลกอริธึม Bellman Ford ใช้วิธีการนับจำนวน hops ที่น้อยที่สุด โดยเส้นทางที่อัลกอริธึม Bellman Ford ใช้ในการส่ง packet จากโหนด 5 ไปยังโหนด 35 คือ 5->4->10->16->15->14->20->21->22 ->23->29->35

| Packet No. | Delivery time | |
|---|---|---|
| | Q-Routing | Bellman Ford |
| 1 | 659 | 11 |
| 2 | 17 | 11 |
| 3 | 23 | 11 |
| 4 | 19 | 11 |
| 5 | 17 | 11 |
| 6 | 13 | 11 |
| 7 | 21 | 11 |
| 8 | 17 | 11 |
| 9 | 13 | 11 |
| 10 | 17 | 11 |
| ⋮ | ⋮ | ⋮ |
| 5998 | 11 | 11 |
| 5999 | 11 | 11 |
| 6000 | 11 | 11 |

**ตารางที่ 4.1** แสดงการเปรียบเทียบผลลัพธ์ของอัลกอริธึม Q-Routing และ Bellman Ford ในสภาวะ Low load (Learning rate = 0.7,Epsilon=0)

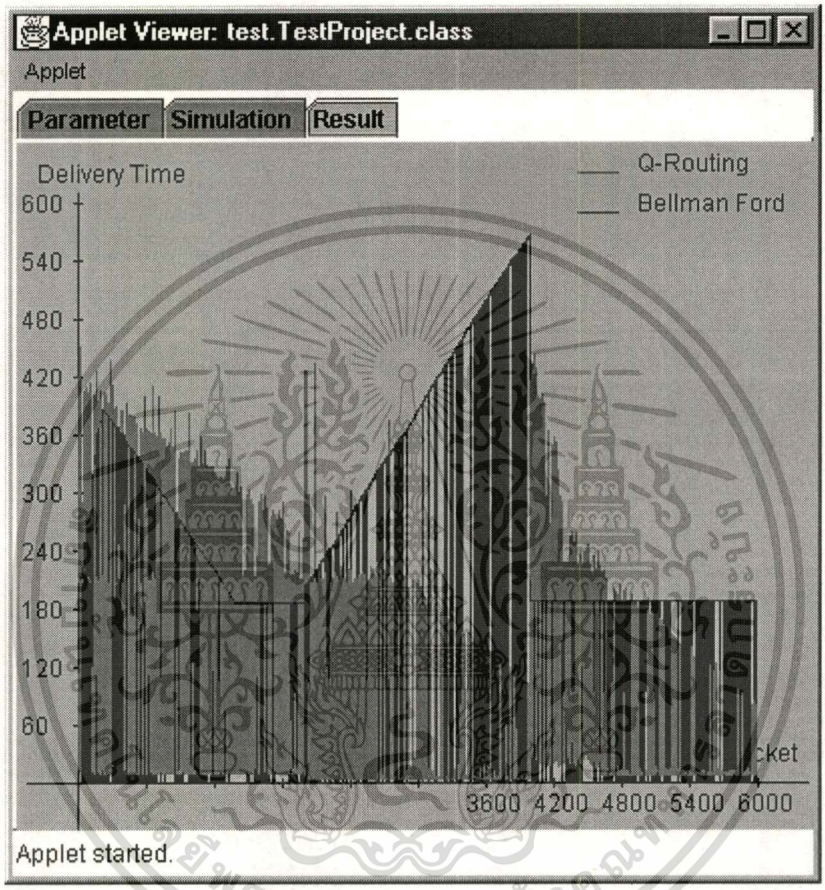



รูปที่ 4.5 กราฟแสดงเวลาที่ใช้ในการส่ง packet (Delivery time) โดยเปรียบเทียบระหว่าง Q-Routing และ Bellman Ford ในสภาวะ Low load (Learning rate = 0.7,Epsilon=0)

ในสภาวะ High load เวลาที่ packet รออยู่ในคิว (Queueing delay) จะมีค่ามากเนื่องจากเส้นทางในการส่งข้อมูลมี packet รออยู่ในคิว และเนื่องจากเป็นการทำงานในลักษณะ FIFO ทำให้ packet ต้องเสียเวลาในการรอคิว ทำให้เกิด Queueing delay ขึ้น ดังนั้นเวลาที่มีผลต่อเวลาในการส่ง packet (Delivery time) ในสภาวะ High load จึงต้องพิจารณาทั้ง Transmission time และ Queueing delay

เวลาที่ใช้ในการส่ง packet ของ Q-Routing จะมีค่าน้อยกว่าอัลกอริธึม Bellman Ford เนื่องจากอัลกอริธึม Bellman Ford ไม่มีความสามารถในการสวิทช์เส้นทางการส่ง packet ไปยังเส้นทางที่ยาวกว่าได้ เนื่องจากอัลกอริธึม Bellman Ford พยายามหาเส้นทางที่มีจำนวน hops น้อยที่สุด เป็นผลทำให้ packet ที่ส่งโดยใช้อัลกอริธึม Bellman Ford ต้องรอคิว เนื่องจากเส้นทางในการส่ง packet เกิดความคับคั่ง (Congestion)

แต่อัลกอริธึม Q-Routing สามารถสวิทซ์ไปยังเส้นทางที่ยาวกว่าได้ ซึ่งมีความคับคั่งน้อยกว่า ทำให้เวลาในการส่ง packet (delivery time) ของ Q-Routing น้อยกว่าของ Bellman Ford รายละเอียดของผลลัพธ์แสดงดังรูปที่ 4.6




**รูปที่ 4.6** กราฟแสดงเวลาที่ใช้ในการส่ง packet (Delivery time) โดยเปรียบเทียบระหว่าง Q-Routing และ Bellman Ford ในสภาวะ High load (Learning rate = 0.7,Epsilon=0)

# บทที่ 5
# บทสรุป

การพัฒนาแบบจำลองการค้นหาเส้นทางโดย Reinforcement Learning ในโครงงานพัฒนาระบบงานนี้ เป็นการนำอัลกอริธึม Q-Routing มาทำการพัฒนาเป็นแบบจำลอง จากการทดลองพบว่าในสภาวะ Low load อัลกอริธึม Q-Routing ให้ผลเหมือนกับอัลกอริธึม Bellman Ford แต่ในสภาวะ High load อัลกอริธึม Q-Routing จะใช้เวลาในการส่ง packet (Delivery time) น้อยกว่าอัลกอริธึม Bellman Ford เนื่องจาก Q-Routing มีความสามารถในการสวิทซ์ไปยังเส้นทางที่ยาวกว่าที่มีความคับคั่งน้อย จึงทำให้ Delivery time น้อยกว่า

ข้อดีของอัลกอริธึม Q-Routing คือ มีความสามารถในการปรับปรุง policy ให้สอดคล้องกับสภาวะแวดล้อมของระบบเครือข่ายที่แปรเปลี่ยนไปได้

ในการนำ Reinforcement Learning มาทำการพัฒนาเป็นระบบจำลองการค้นหาเส้นทางในระบบเครือข่าย เพื่อให้ผู้ที่สนใจศึกษาอัลกอริธึมดังกล่าวได้เห็นประโยชน์ของ Reinforcement Learning และให้ผลดังต่อไปนี้

1. ช่วยทำให้ผู้ศึกษาอัลกอริธึมสามารถเห็นผลลัพธ์ของอัลกอริธึมที่ศึกษา ทำให้สามารถเข้าใจหลักการทำงานของ Reinforcement Learning ได้ดียิ่งขึ้น และสามารถนำหลักการของ Reinforcement Learning ไปประยุกต์ใช้งานในด้านต่างๆได้
2. เป็นต้นแบบสำหรับผู้ที่ต้องการทำการพัฒนาอัลกอริธึมที่ใช้ในการค้นหาเส้นทาง

**ปัญหาและอุปสรรค**

จากการทำการพัฒนาโปรแกรม ผู้พัฒนาพบปัญหาดังนี้

1. เนื่องจากผู้พัฒนาใช้ภาษา Java ซึ่งมีลักษณะการทำงานแบบเชิงวัตถุ (Object Oriented) ในการพัฒนาโปรแกรม ซึ่งผู้พัฒนายังไม่มีความชำนาญดีพอ จึงทำให้ต้องเสียเวลาในการศึกษานานพอสมควร ดังนั้นแบบจำลองที่ได้อาจจะยังไม่สมบูรณ์ หรือมีข้อผิดพลาดบ้าง
2. จากการที่ยังไม่มีการนำ Reinforcement Learning มาใช้กันอย่างแพร่หลาย ทำให้ต้องใช้เวลานานในการค้นหาข้อมูล เพื่อที่จะนำมาประยุกต์ใช้ในการพัฒนาแบบจำลอง

**ข้อเสนอแนะ**

เนื่องจากแบบจำลองการค้นหาเส้นทางที่ได้พัฒนาขึ้นนี้ เป็นแบบจำลองที่มีตัวอย่างเปรียบเทียบระหว่าง 2 อัลกอริธึมเท่านั้น คือ Q-Routing และ Bellman Ford ดังนั้น สำหรับผู้ที่ต้องการจะพัฒนาหรือดัดแปลงแบบจำลองการค้นหาเส้นทาง ควรจะต้องให้แบบจำลองมีคุณสมบัติต่างๆ ดังนี้

1. มีอัลกอริธึมในการค้นหาเส้นทางชนิดต่างๆ ทุกประเภท เพื่อจะได้สามารถเปรียบเทียบผลลัพธ์ของแต่ละอัลกอริธึมได้
2. ควรจะมีรูปแบบของระบบเครือข่าย (Network topology) ให้เลือกหลายแบบ เพื่อจะได้เห็นความสามารถของแต่ละอัลกอริธึมว่ามีประสิทธิภาพอย่างไรกับ Network topology ในแบบต่างๆ

จากข้อเสนอแนะข้างต้น ผู้พัฒนาคาดว่าจะสามารถเป็นแนวทางสำหรับผู้ที่ต้องการปรับปรุงแบบจำลองในการค้นหาเส้นทางในระบบเครือข่ายต่อไป

# บรรณานุกรม

J. Boyan and M. Littman. 1993. "A distributed reinforcement learning scheme for network Routing."**Technical Report CMU-CS-93-165,** School of Computer Science, Carnegie Mellon University.

J. Boyan and M. Littman. 1995. **Packet routing in dynamically changing networks : A reinforcement learning approach.**
Available: http://www.cs.duke.edu/~mlittman/topics/routing-page.html

John W. Bates. 1995. "Packet Routing and Reinforcement Learning: Estimating Shortest Paths in Dynamic Graphs"

Richard S. Sutton and Andrew G. Barto. 1998. **Reinforcement Learning,** MIT Press

Samuel P.M. Choi, Dit-Yan Yeung, "Predictive Q-Routing: A memory-based Reinforcement Learning approach to Adaptive Traffic Control." Department of Computer Science, Hong Kong university of Science and Technology.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. 1994. **Introduction to Algorithms,** MIT Press

Yishay Mansour. 1990. **Packet Routing Using Reinforcement Learning: Estimating Shortest paths in Dynamically changing networks.**
Available : http://www.math.tau.ac.il/~mansour/rl-course/student_proj/shlomy/learnnet.html