บทที่ 6
State Space Graphs

6.1 โครงสร้างและกลวิธี (Structures and strategies)

เราได้พูดถึง predicate calculus ไปแล้วในแง่ของการใช้ predicate calculus เป็น repredicatation language ในงานด้าน AI Well-formed predicate calculus เป็นรูปแบบที่ดีที่จะใช้ในการอธิบายถึง objects และ relations ในปัญหาที่เรากำลังสนใจ และยังมี inference rules ซึ่งช่วยให้เราสามารถสร้างความรู้ใหม่ขึ้นมาได้จากรายละเอียดที่เรามีอยู่ โดยการอนุมานนี้จะมีการกำหนดขอบเขต (space) ซึ่งจะใช้ในการค้นหาข้อสรุป (solution) ของปัญหาดังนั้น เราจะมาพูดถึง ทฤษฎี state space search.

ในการออกแบบและสร้างกลไกของการ search ให้ประสบผลสำเร็จนั้นโปรแกรมเมอร์จะต้องสามารถวิเคราะห์และทำนายพฤติกรรม (behavior) ของการ search นั้นได้ นั่นคือ ต้องมีการพิจารณารายละเอียดต่อไปนี้

problem solver นั้นจะให้ข้อสรุปได้จริงหรือเปล่า

problem solver นั้นจะต้องทำงานเสร็จจริง หรือติดอยู่ใน loop หรือเปล่า

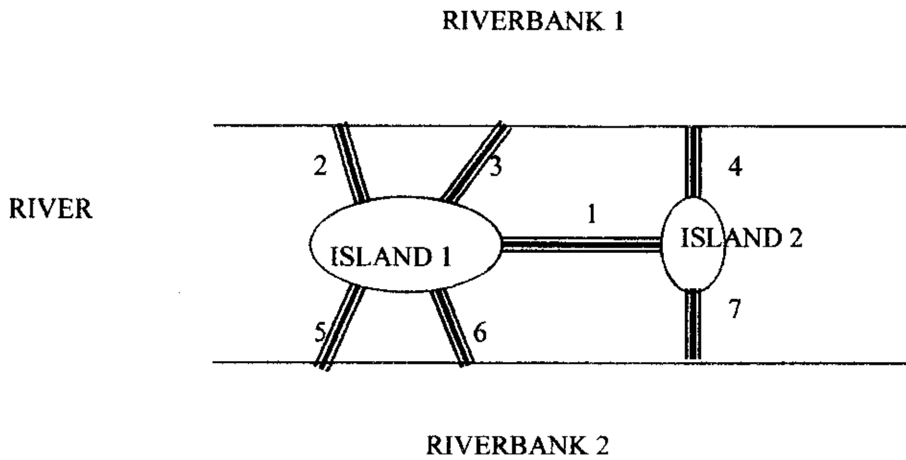
ข้อสรุป (solution) ที่ได้ จะต้องแน่ใจได้ว่าเป็น optimal solution

ความสับซ้อนของการ search โดยพิจารณาเวลาที่ใช้ หรือ หน่วยความจำที่ใช้

ทำอย่างไร interpreter จะทำให้การ search นั้นยุ่งยากน้อยลง

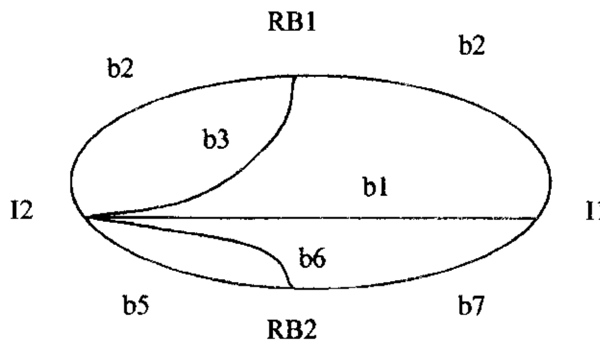
ทำอย่างไร interpreter จึงจะได้รับการออกแบบให้ทำงานกับ representation ได้ดี ที่สุด

ทฤษฎีของ state space search เป็นสิ่งที่จะนำมาใช้ตอบคำถามเหล่านี้ได้อย่างดีโดยเราจะมีการอธิบายปัญหาในรูปของ state space graph จากนั้นเราใช้ graph theory ในการวิเคราะห์โครงสร้างและความซับซ้อนของปัญหา และวิธีที่จะแก้ปัญหาที่เกิดขึ้น graph theory เป็นเครื่องมือที่ดีที่สุดที่จะใช้ในการให้เหตุผลเกี่ยวกับโครงสร้างของ object และ relations โดยการใช้กราฟนั้นเริ่มขึ้นจากปัญหา "bridges of Konigsberg" ใน คริสตวรรษที่ 18



ปัญหาก็คือจะเป็นไปได้อย่างไรที่จะให้คนๆ หนึ่งเริ่มเดินจากจุดหนึ่ง แล้วเดินไปรอบเมือง โดยข้ามสะพานแต่ละแห่งเพียงครั้งเดียว แล้วกลับมายังที่เดิมได้

Euler ได้แสดงปัญหานี้ใหม่ในรูปของ graph



ในการพิจารณาเรื่องการเดิน Euler ได้พิจารณาจาก degree ของ node (คือจำนวน arc ที่มีในแต่ละ node นั่นเอง) ว่าเป็น odd หรือ even degree

odd degree node : คือ node ที่มีจำนวน arc เป็นเลขคี่

even degree node : คือ node ที่มีจำนวน arc เป็นเลขคู่จากการพิจารณา degree ของ node ทำให้ Euler สรุปได้ว่า การเดินรอบเมืองตามกฎเกณฑ์ที่กำหนดไว้ จะเป็นได้เมื่อ มีจำนวน odd degree node อยู่ใน graph เป็นจำนวน 0 หรือ 2 node นอกเหนือจากนี้การเดินทางดังกล่าวจะไม่มีวันสำเร็จ

ปัญหานี้ได้ถูกเรียกว่า **Euler path :**

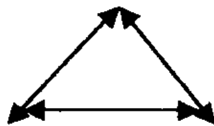
เราสามารถใช้ predicate calculus แสดงปัญหา Konigsberg ได้เช่นกันโดย จะอยู่ในรูปของ

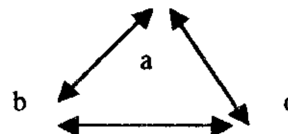
CONNECT(I1,I2,b1)	CONNECT(I1,I2,b1)
CONNECT(RB1,I1,b2)	CONNECT(I1,RB1,b2)
CONNECT(RB1,I1,b3)	CONNECT(I1,RB1,b3)
CONNECT(RB1,I2,b4)	CONNECT(I2,RB1,b4)
CONNECT(RB2,I1,b5)	CONNECT(I1,RB2,b5)
CONNECT(RB2,I1,b6)	CONNECT(I1,RB2,b6)
CONNECT(RB2,I2,b7)	CONNECT(I2,RB2,b7)

ทฤษฎีกราฟ

กราฟ (graph) ประกอบด้วย set ของ nodes และ set arcs ซึ่งทำหน้าที่เชื่อม node 2 ตัวเข้าด้วยกัน ในการสร้าง state space model ในการแก้ปัญหา node ของกราฟ จะหมายถึง state ที่จะเกิดขึ้นในการทำ problem-solving ส่วน arc จะหมายถึง ทางเดินจาก state หนึ่งไปยังอีก state หนึ่งซึ่งจะเกิดขึ้นตาม logical inference ของปัญหานั้นนั่นเอง

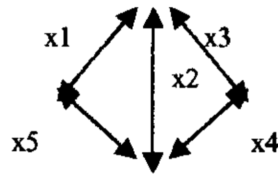
Labeled graph :จะหมายถึง กราฟที่มีการให้รายละเอียดแก่ node ที่มีอยู่ในกราฟ





unlabeled graph **labeled graph**

the arcs of the graph may be labeled



Directed graph : จะหมายถึง กราฟที่มีการให้ทิศทางกับ arc โดยการใช้ลูกศรเป็นสิ่งกำหนดทิศทาง

Path : หมายถึง ทางเดินของกราฟจากจุดกำหนดไปยังจุดปลายทาง เช่น [a,b,c,d]

Rooted graph : คือกราฟที่มี node หนึ่งทำหน้าที่เป็น root คือเป็น node ที่ทำให้เกิด path ไปยัง node อื่นๆ ในกราฟ

Tree : คือ กราฟซึ่งมี arc เพียงเส้นเดียวเชื่อมระหว่าง node 2 ตัว

Connected graph : คือ กราฟที่มี node 2 ตัวมี path เชื่อมกันไว้

6.2 State space representation of problems

ในการแสดงปัญหาที่เกิดขึ้นโดยใช้ state space graph นั้น

node ของกราฟจะหมายถึง state ที่เป็นวิธีการแก้ปัญหาอย่างหนึ่ง ส่วน arc จะหมายถึงขั้นตอนในการแก้ปัญหา

initial state จะหมายถึง information ที่กำหนดให้ในตอนเริ่มต้นซึ่งทำหน้าที่เป็น root และ graph ยังระบุ goal condition ซึ่งก็คือ solution ของปัญหานั่นเอง

State space search ก็คือ กระบวนการในการหา solution path จาก state space ไปยัง goal นั่นเอง

state space จะประกอบด้วยองค์ประกอบ 4 ส่วนคือ [N,A,S,GD]

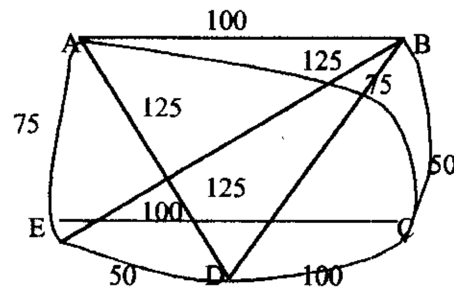
N – คือ set ของ nodeS ที่จะมีในกราฟ

A – คือ set ของ arcs ที่เชื่อม nodes ในกราฟ

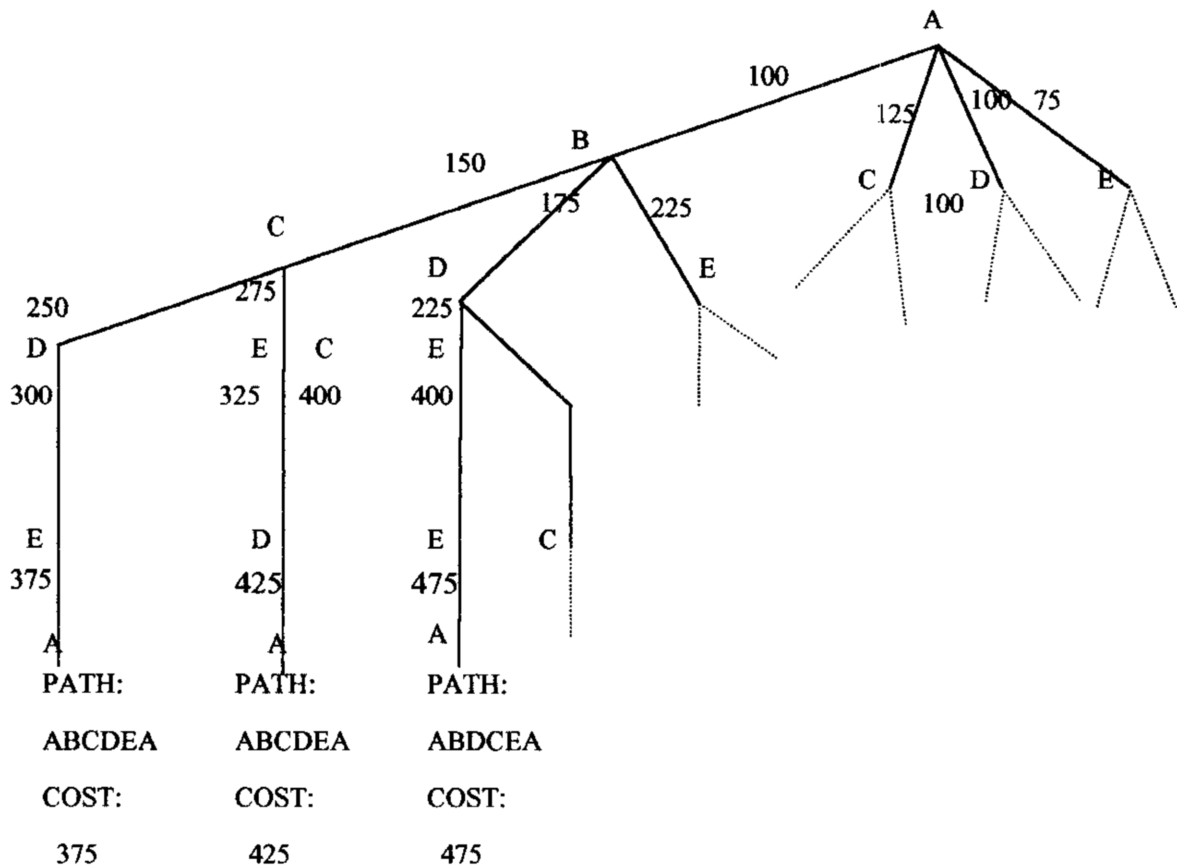
S – คือ subset ของ N ที่มี start state เป็นสมาชิก (ต้องไม่เป็น empty)

Solution path : คือ ทางเดินที่เริ่มจาก node ใน S ไปยัง node ใน GD

<u>ตัวอย่าง</u> : traveling salesman



an instance of the problem:



ตัวอย่างรูปนี้แสดงให้เห็นถึง exhaustive search คือต้อง search ถึง (N-1)! (Nคือจำนวนเมืองที่มีกราฟ)

6.3 Strategies for state space search

การ search ใน state space ทำได้ 2 แนวทาง นั่นคือ การเริ่มจาก data ที่มีอยู่ มุ่งไปหา goal ที่ต้องการ หรือ จาก goal เดินกลับมาหา data

<u>**data-driven search**</u> **:** เรียกอีกอย่างว่า **backward chaining**

เราเริ่มจากการพิจารณา goal นี้ และ มีเงื่อนไข (condition) ใดบ้างที่จะทำให้กฎ เหล่านั้นเป็นจริง conditions ดังกล่าวจะกลายมาเป็น goal ใหม่ที่เรียกว่า subgoal ซึ่งจะนำไปใช้ในการหา subgoal ตัวต่อไป จนกระทั่งเราย้อนกลับไปถึง data ของปัญหาที่เรากำลังแก้อยู่ ทั้ง data-driven search และ goal-driven search นี้ถึงแม้จะมีวิธีการที่ ต่างกันแต่ก็ใช้ใน state space เดียวกัน การเลือกใช้ search นั้นจะพิจารณาจากลักษณะของปัญหา, ความยุ่งยากของกฎที่ใช้, รูปร่างของ state space และความพร้อมของ fact ที่จะใช้ในการแก้ปัญหา (ซึ่งจะแตกต่างกันไปตามชนิดของปัญหา)

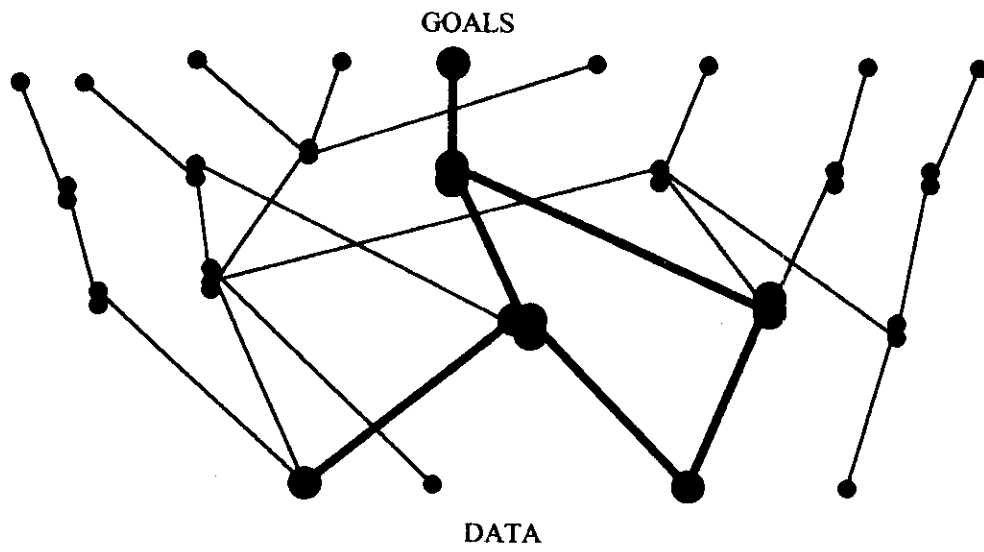
<u>**goal-driven search**</u> จะเหมาะแก่การใช้เมื่อ

1. มีการระบุ goal ไว้ใน problem statement แล้ว หรือ goal นั้นสามารถ นำมากำหนดเป็น formula ได้ง่าย (เพราะ goal ก็คือ ทฤษฎีที่ต้องการ การพิสูจน์นั่นเอง)
2. มีกฎ (rule) หลายกฎที่ตรงกับ facts ที่มีใน problem และสามารถนำมาใช้ สร้าง goal ได้ง่ายขึ้น
3. ไม่มีการกำหนด problem data มาให้

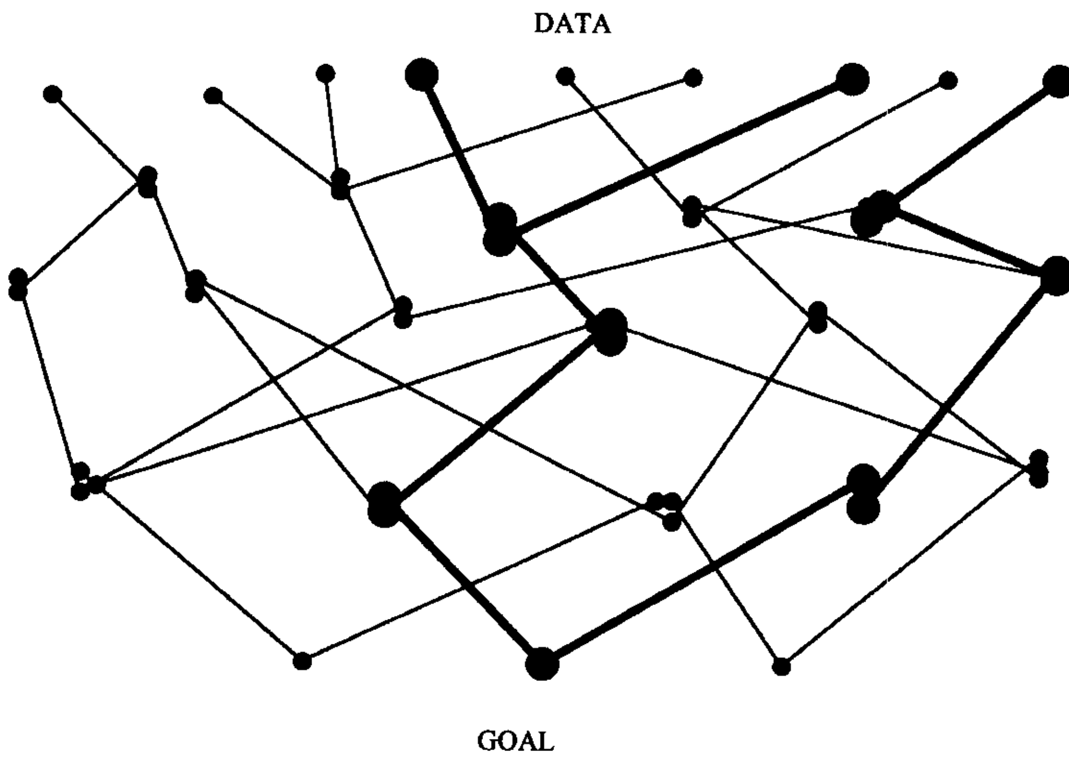
ส่วน **data-driven apporach** นั้นจะเหมาะแก่การใช้เมื่อ

1. มีการกำหนด data ทั้งหมด หรือเกือบทั้งหมดให้กับ problem statement
2. มี goal ที่อาจเป็นไปได้จำนวนมาก แต่มีแนวทางที่จะใช้ fact เพื่อจะนำมาซึ่ง ข้อมูลที่ต้องการนั้นน้อย
3. การกำหนด goal นั้นทำได้ยาก

GOAL-DRIVEN



DATA-DRIVEN



ในการแก้ปัญหานั้นไม่ว่าจะเป็นการใช้ goal-driven search หรือ data-driven search ก็ตาม ผู้แก้ปัญหาต้องหา path ที่ถูกต้องใน state space graph ซึ่งก็คือ path ที่เดินจาก start state ไปยัง goal state นั่นเอง แต่การที่จะหา path นี้พบในการพยายามครั้งแรกเลยนั้นเป็นไปได้ยาก ดังนั้นจึงเกิดกรณีที่เราค้นลงไปตาม path หนึ่งแล้วพบว่าเป็น path ที่ไม่ถูกต้อง เราเลยต้องย้อนกลับมาเริ่มใหม่ใน path ใหม่ที่เราคาดว่าคงจะเป็น path ที่ถูกต้อง ซึ่งก็อาจเป็นกรณีที่เราต้องย้อนกลับไปเริ่มใหม่อีกก็ได้เช่นกัน เราจะทำเช่นนี้ไปเรื่อยๆ จนพบ path ที่ถูกต้องจริงๆ

backtracking คือ เทคนิคที่ใช้พิจารณาเรื่องการย้อนกลับไปเริ่มต้นใหม่ในการค้นหา path แต่เป็นการทำงานอย่างมีรูปแบบที่ถูกต้อง โดยจะเริ่มจาก start state จนพบ goal state หรือไม่ก็ dead end เราเรียกลักษณะการค้นหา solution path นี้ว่า **backtracking search**

<u>อัลกอริทึม</u> ของการทำ backtracking มีดังนี้

<u>กำหนดให้</u>

SL – เป็น State List คือ list ที่เก็บ state ที่กำลังถูกค้นหา path ถ้าพบ goal สิ่งที่อยู่ใน SL ก็คือบรรดา state ที่เป็น solution path นั่นเอง

NSL – New State List – เก็บ state ที่กำลังรอการตรวจสอบ

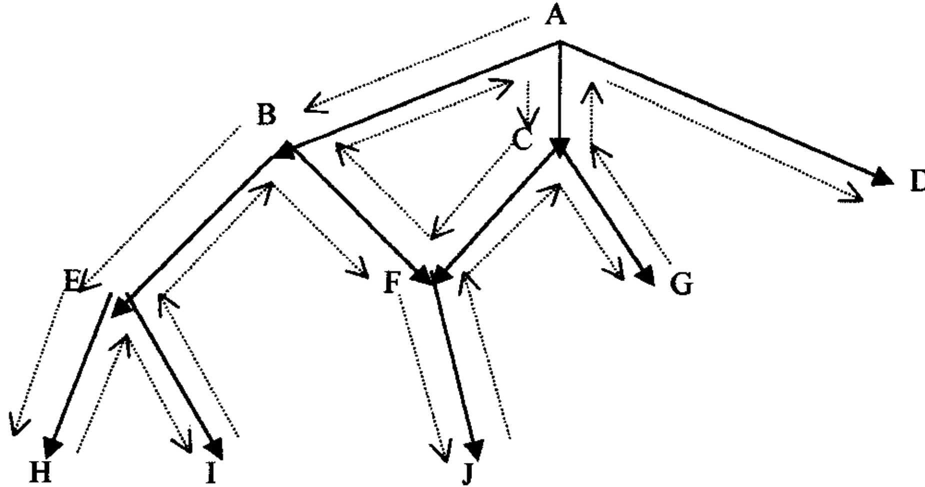
DE – Dead End – คือ List ของ state ที่ไม่สามารถไปถึง goal ได้

CS – Current State – คือ state ที่กำลังถูกพิจารณา

FUNCTION BACKTRACK

```
BEGIN
  SL := [START]; NSL := [START]; DE := [ ]; CS := START; %INITIALZE:
  WHILE NSL <> [ ] DO
        BEGIN
          IF CS = GOAL THEN RETURN (s1);
           IF CS HAS N CHILDREN (EXCLUDING nodeS ALLREADY IN DE,SL,AND NSL
           THEN
                BEGIN
                  WHILE SL IS NOT EMPTY AND CS = FIRST ELEMENT OF SL DO
                        BEGIN
                          ADD CS TO DE;
                          REMOVE FIRST ELEMENT FROM SL;
                          REMOVE FIRST ELEMENT FROM NSL;
                          CS := FIRST ELEMENT OF NSL;
                        END;
                        ADD CS TO SL;
                END
           ELSE
           BEGIN
                PLACE CHILDREN OF CS (EXCEPT NODES ALLREADY ON DE,SL,NSL) ON NSL;
                CS := FIRST ELEMENT OF NSL;
                ADD CS TO SL
           END
        END
  RETURN FAIL
END.
```

ตัวอย่าง



INITIAL : SL = [A]; NSL = [A]; DE = [ ]; CS = A

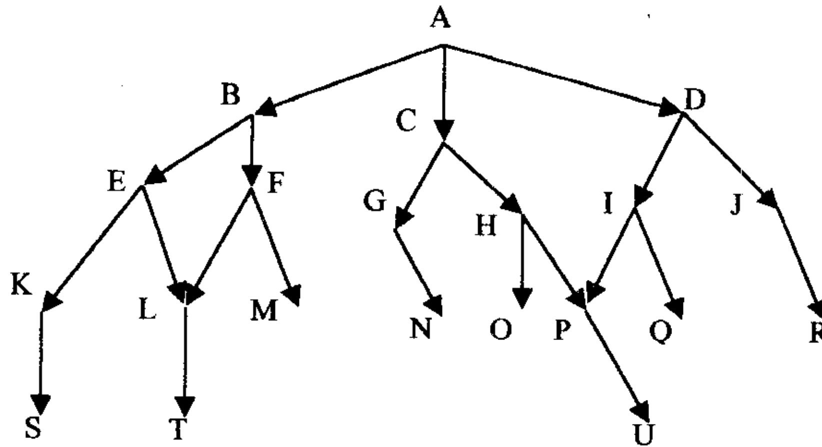
AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[B,A]	[B,C,D,A]	[ ]
2	E	[E,B,A]	[E,F,B,C,D,A]	[ ]
3	H	[H,E,B,A]	[H,I,E,F,B,C,D,A	[ ]
4	I	[I,E,B,A]	[I,E,F,B,C,D,A]	[H]
5	F	[F,B,A]	[F,B,C,D,A]	[E,I,H]
6	J	[J,F,B,A]	[J,F,B,C,D,A]	[E,I,H]
7	C	[C,A]	[C,D,A]	[B,F,J,E,I,H]
8	G	[G,C,A]	[G,C,D,A]	[B,F,J,E,I,H]

backtrack คือว่าเป็น basic algorithm สำหรับการค้นหา state space graph

Depth-and breath-first search

ในการ search นั้นนอกจากจะมีการกำหนดทิศทางในการค้นหาแบบ data-driven หรือ goal-driven search แล้ว เรายังเพิ่มกลไกในการ search ให้แน่นอนขึ้นได้อีกโดยการพิจารณาจากลำดับของการพิจารณา state ที่อยู่ใน state space graph ของเราทำให้เกิดวิธีการค้นหาแบบ depth-first และ breath-first search

ตัวอย่าง



Depth-first search :

A,B,E,K,S,L,T,F,M,C,G,N,H,O,P,U,D,I,Q,J,R

Breath-first search :

A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U

จะเห็นได้ว่า algorithm ของการทำ backtrack สามารถนำมาใช้กับการค้นหาแบบ depth-first search ได้ algorithm ของการทำ breath-first-search นั้นสามารถทำได้โดยการใช้ list 2 ตัว (open และ close) เก็บรายละเอียดเกี่ยวกับ state ในขณะที่มีการ search เข้าไปใน state space

OPEN – เป็น list ที่เก็บ state ที่ถูกตรวจสอบ

CLOSE – เป็น list ที่เก็บ state ที่ผ่านการตรวจสอบแล้ว

PROCEDURE BREADETH_FIRST_SEARCH

INITIAL : OPEN = [START]; CLOSE = [ ]

WHILE OPEN ≠ [ ] DO

BEGIN

REMOVE THE LEFT-MOST START FROM OPEN, CALL IT X;

IF X IS GOAL THEN RETURN(SUCCESS);

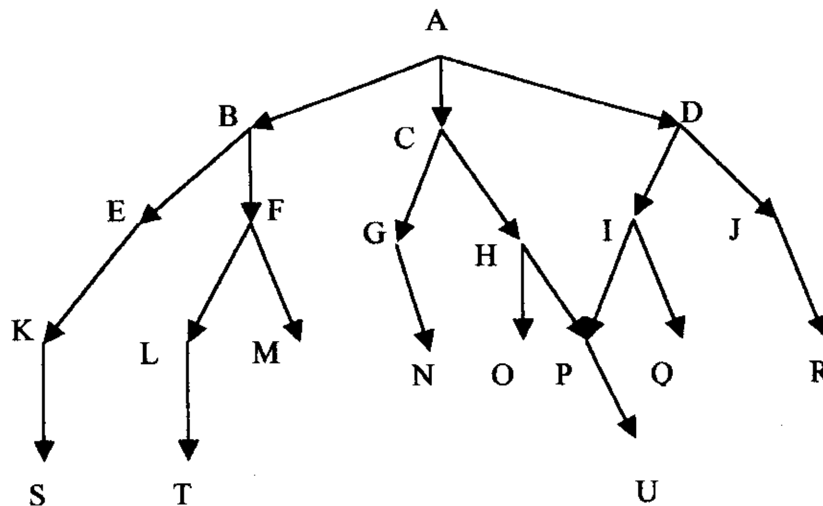
GENERATE ALL CHILDREN OF X;

PUT X ON CLOSE;

ELIMINATE ANY CHIDREN OF X ALLREADY ON EITHER OPEN OR CLOSE, AS THESE WILL CAUSE LOOPS IN THE search;

PUT THE RAMANING DESCENDENTS, IN ORDER OF DISCOVERY, ON THE RIGHT END OF OPEN;

END.



<u>TRACE:</u> BREADTH_FIRST_SEARCH

1. OPEN = [A]; CLOSE = [ ]
2. OPEN = [B,C,D]; CLOSE = [A]
3. OPEN = [C,D,E,F]; CLOSE = [B,A]
4. OPEN = [D,E,F,G,H]; CLOSE = [C,B,A]

5. OPEN = [E,F,G,H,I,J]; CLOSE = [D,C,B,A]
6. OPEN = [F,G,H,I,K,L]; CLOSE = [E,D,C,B,A]
7. OPEN = [G,H,I,J,K,L,M]; CLOSE = [F,E,D,C,B,A]
8. OPEN = [H,I,J,K,L,M,N[; CLOSE = [G,F,E,D,C,B,A]
9. :

ทำเช่นนี้ไปเรื่อยๆ จนกระทั่งพบ goal หรือ OPEN = []

PROCEDURE DEPTH_FIRST_SEARCH

INITIAL : OPEN = [START]; CLOSE = [ ]

WHILE OPEN $\neq$ [ ] DO

BEGIN

REMOVE THE NEXT state FROM THE LEFTF OF OPEN, CALL IT X;

IF X IS GOAL THEN RETURN(SUCCESS);

GENERATE ALL POSSIBLE CHILDREN OF X;

PUT X ON CLOSE;

ELIMINATE ANY CHIDREN OF X ALLREADY ON EITHER OPEN OR CLOSE, AS THESE WILL CAUSE LOOPS IN THE search;

PUT THE REMAINING CHILDREN OF X, IN ORDER OF DISCOVERY, ON THE LEFT END OF OPEN

END.

TRACE :

1. OPEN = [A]; CLOSE = [ ]
2. OPEN = [B,C,D]; CLOSE = [A]
3. OPEN = [E,F,C,D]; CLOSE = [B,A]
4. OPEN = [K,L,F,C,D]; CLOSE = [E,B,A]
5. OPEN = [S,L,F,C,D]; CLOSE = [K,E,B,A]
6. OPEN = [L,F,C,D]; CLOSE = [S,K,E,B,A]

7. OPEN = [T,F,C,D]; CLOSE = [L,S,K,E,B,A]

8. OPEN = [F,C,D[; CLOSE = [T,L,S,K,E,B,A]

9. OPEN = [M,C,D[; CLOSE = [F,T,L,S,K,E,B,A]

10. OPEN = [C,D[; CLOSE = [M,F,T,L,S,K,E,B,A]

11. OPEN = [G,H,D[; CLOSE = [C,M,F,T,L,S,K,E,B,A]

:

UNTIL OPEN = [ ] OR GOAL IS FOUND

6.4 And/Or graph

ใน state space graph ที่เราสร้างขึ้นมานั้น node แต่ละตัวจะหมายถึง state ที่อาจทำให้เราไปหาข้อสรุปของปัญหาได้ จากนั้นเราสามารถใช้ predicate calculus แสดงให้เห็นถึง state ต่างๆ ที่จะเป็นองค์ประกอบของกระบวนการแก้ปัญหาได้ ส่วน inference rule ก็คือ arc ที่เชื่อม node ในกราฟนั่นเอง ดังนั้น ปัญหาที่อยู่ในรูปของ predicate calculus อาจจะหา solution ได้โดยการ search เช่นกัน

<u>ตัวอย่าง</u> ของการแสดง proposition calculus ในรูปของ graph

$q \Rightarrow p$

$r \Rightarrow p$

$v \Rightarrow q$

$s \Rightarrow r$

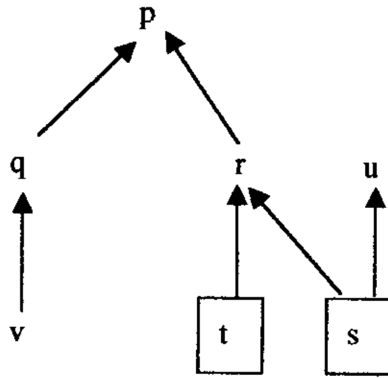
$t \Rightarrow r$

$s \Rightarrow u$

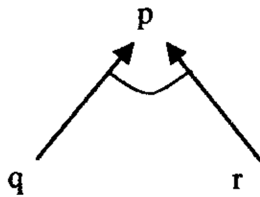
s

t

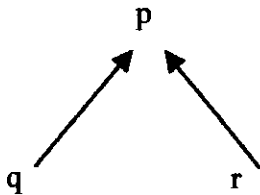
จะสามารถแสดงในรูปกราฟได้เป็น



ซึ่งกราฟนี้จะแสดงถึงความสัมพันธ์ในรูปของ implication $\Rightarrow$ ซึ่งอยู่ในรูปแบบของ directed graph และการ search จะเป็นแบบ data-driven search ในความเป็นจริง เรายังมี and และ or operation ที่ควรจะแสดงได้เช่นกันในรูปของ graph จึงได้มีการกำหนด **<u>And/Or graph</u>** ขึ้น ดังรูปแบบต่อไปนี้



หมายถึง $q \wedge r \Rightarrow p$



หมายถึง $q \vee r \Rightarrow p$

<u>ตัวอย่าง</u> ของการใช้ And/Or graph โดยพิจารณา proposition calculus ต่อไปนี้

a

b

c

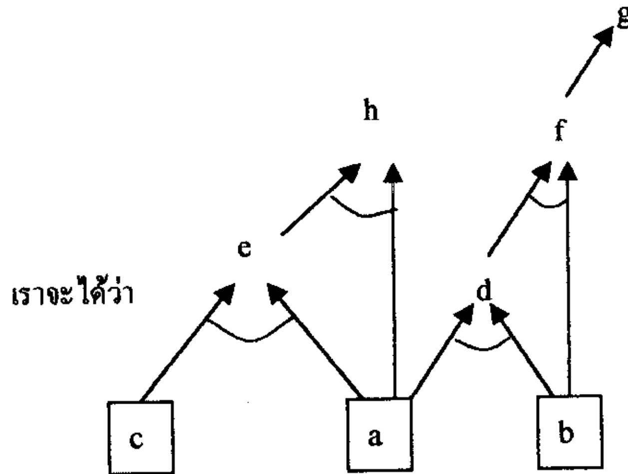
$a \wedge b \Rightarrow d$

$a \wedge c \Rightarrow d$

$b \wedge c \Rightarrow d$

$f \Rightarrow g$

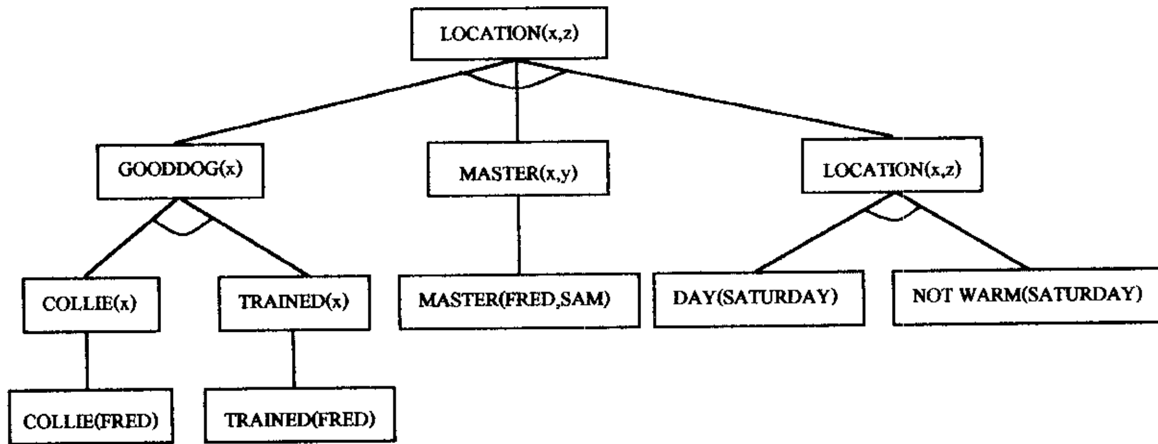
$a \wedge c \Rightarrow h$



<u>ตัวอย่าง</u> การสร้าง And/Or graph จาก predicate calculus ต่อไปนี้

1. FRED IS A COLLIE. : COLLIE(FRED).
2. SAM IS FRED'S MASTER. : MASTER(FRED,SAM).
3. IT IS SATURDAY. : DAY(SATURDAY).
4. IT IS COLD ON SATURDAY. : $\neg$WARM(SATURDAY).
5. FRED IS A TRAINED DOG. : TRAINED(FRED).
6. SPANIELS OR COLIES THAT ARE TRAINED ARE GOOD DOGS. :
   $\forall$ x [SPANIEL(x) $\vee$ (COLLIE(x) $\wedge$ TRAINED(x)) $\Rightarrow$ GOODDOG(x))].
7. IF A DOG IS GOOD DOG AND HAS A MASTER THEN HE WILL BE WITH HIS MASTER. : $\forall$ x $\forall$ y $\forall$ z[GOODDOG(x) $\wedge$ MASTER(x,y) $\wedge$ LOCATION(y,z)
   $\Rightarrow$ LOCATION(x,z)].
8. IF IT IS SATURDAY AND WARM, THEN SAM IS AT THE PARE. :
   DAY(SATURDAY) $\wedge$ WARM(SATURDAY) $\Rightarrow$ LOCATION(SAM,PARK).
9. IF IT IS SATURDAY AND NOT WARM, THEN SAM IS AT THE MUSEUM.
   DAY(SATURDAY) $\wedge$ $\neg$WARM(SATURDAY) $\Rightarrow$ LOCATION(SAM,MUSEUM).

goal ของปัญหานี้อยู่ในรูป expression ของ ∃xLOCATION(FRED,x) หมายถึง "FRED อยู่ที่ไหน" นั่นเอง การ search ของปัญหานี้จะใช้ And/Or graph และจะใช้ goal-driven search



SUBSTITUTIONS = {FRED/x,SAM,y,MUDEUM/z}

6.5 Production Systems:PS

PS เป็น computation model ที่Newell ได้เสนอแนวความคิดเกี่ยวกับเรื่อง PS ขึ้นราวปี ค.ศ. 1967 เพื่อใช้เป็นเครื่องมือในการศึกษาถึงกระบวนการคิดของมนุษย์ ปัจจุบัน PS ได้กลายมาเป็น knowledge representation scheme ที่ได้รับความนิยมอย่างมากในงานด้าน expert system

โครงสร้างของ PS

PS ประกอบด้วยองค์ประกอบสำคัญ 3 ส่วน คือ

1. Production memory:PM

PM เก็บรายละเอียดเกี่ยวกับความรู้ โดยความรู้ดังกล่าวจะถูกอธิบายไว้ในรูปของ **productions** หรือ **rules** ซึ่งมีรูปแบบเป็น

"CONDITION ⟶ CONCLUSION"

ส่วนของ **condition** ของ production นี้บางทีก็เรียกว่า assumption,Left-Hand Side, LHS part หรือ if part โดย condition นี้จะอธิบายถึง pre-condition ที่จะต้องเกิดขึ้น ก่อนที่จะมีการนำ production ไปใช้

ส่วนของ conclusion ของ production นี้บางครั้งก็เรียกว่า action part, Right-Hand Side part, RHS part หรือ then part โดยcondition นี้จะอธิบายถึงการทำงานขององค์ประกอบของ production เช่น การแสดง output หรือ การเก็บข้อมูลเข้าไปใน working memory

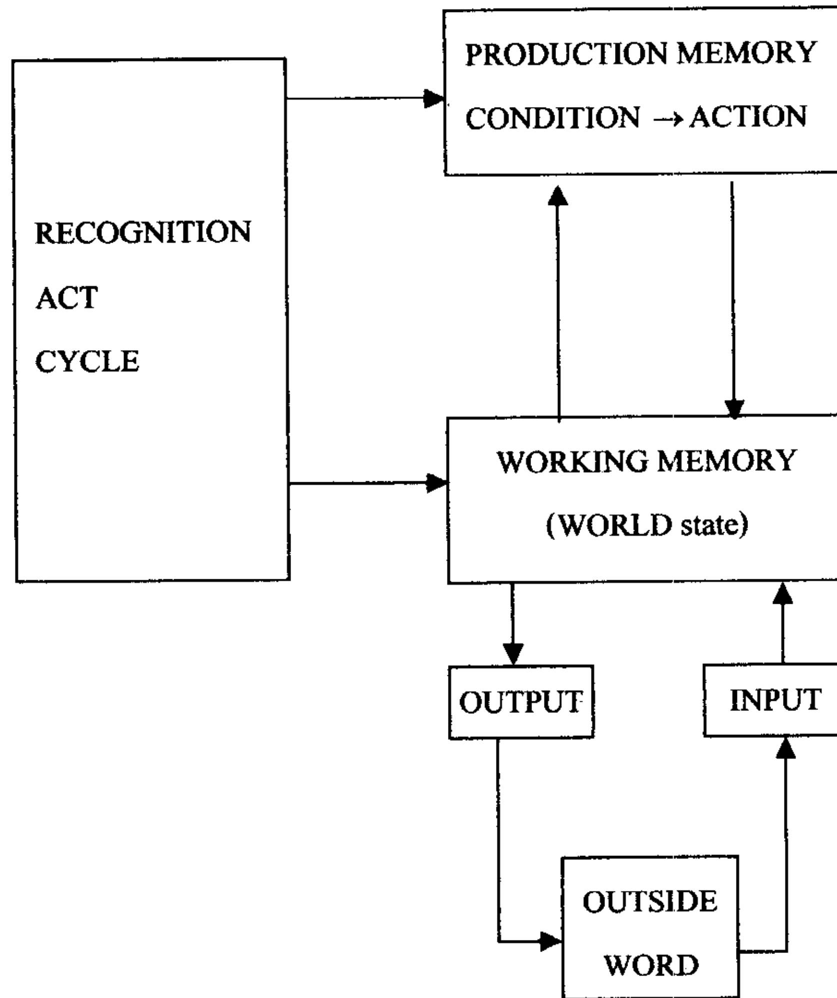
2. Working Memory:WM

WM เก็บข้อมูลซึ่งถูก PM เรียกไปใช้หรือข้อมูลที่ถูก production เปลี่ยนแปลงไป WM เรียกอีกอย่างได้ว่า "database" หรือ "context"

3.Recognize-act cycle :

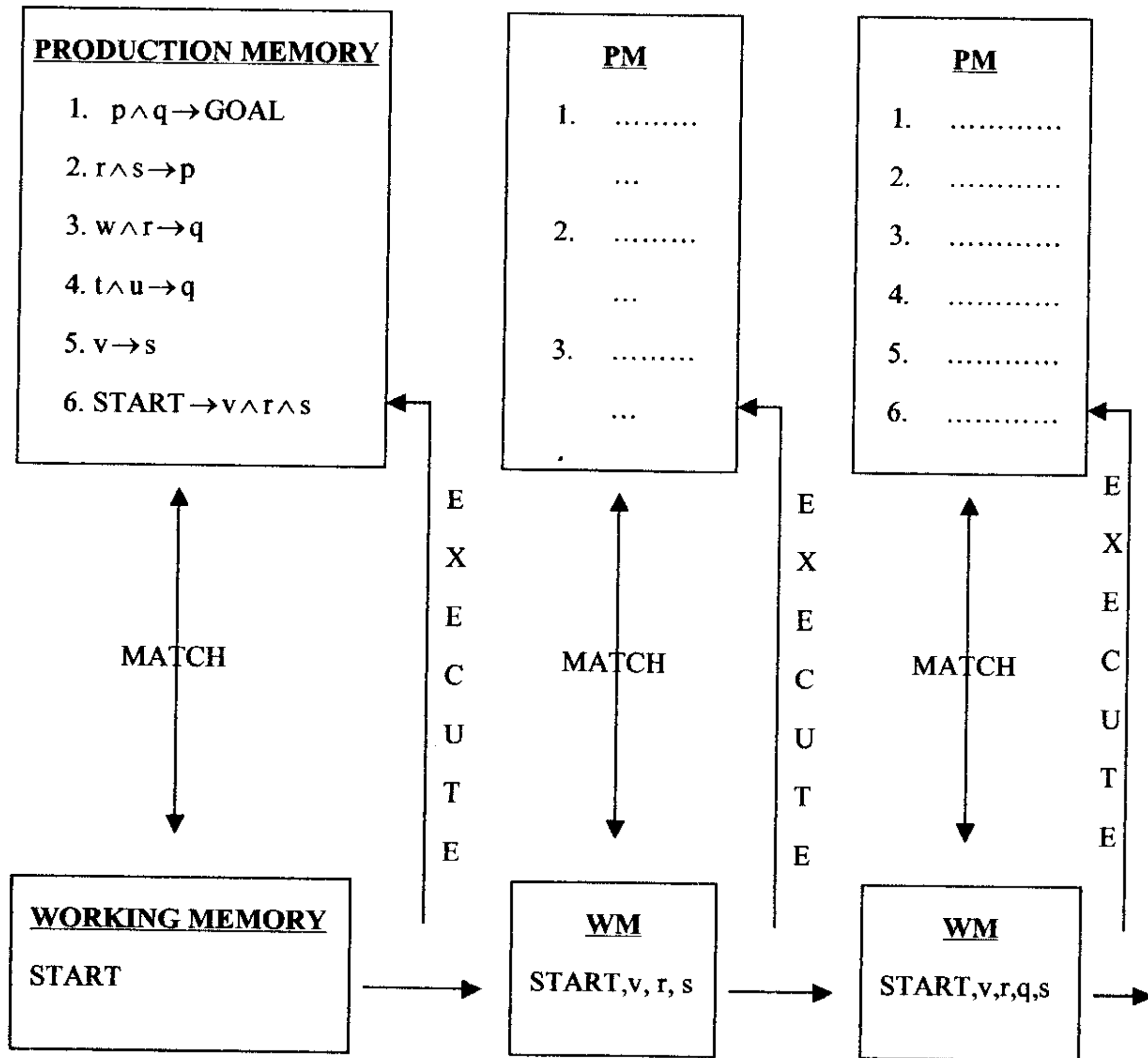
เป็นส่วนควบคุมการทำงานของ PS โดยการควบคุมจะมีรูปแบบดังนี้

- current state ของกระบวนการแก้ปัญหาจะถูกเก็บไว้ใน WM
- WM จะเป็นตัวเก็บขั้นตอนเริ่มต้นที่เกิดขึ้นในการแก้ปัญหา
- รูปแบบที่มีใน WM จะถูกนำไปหาความสัมพันธ์กับ productions ที่อยู่ใน PM ซึ่งทำให้เกิด **conflict set** คือ subset ของ productions ที่สัมพันธ์กับสิ่งที่อยู่ใน WM
- productionS ที่อยู่ใน conflict set จะถูกกำหนดให้เป็น enable
- productionS ใน conflict set ตัวใดตัวหนึ่งจะถูกเลือกมาใช้ในการทำงาน ซึ่ง เรียกว่าเป็นการ fired นั่นคือ ส่วน conclusion ของ production ที่ถูก **fired** จะเกิดการทำงาน ซึ่งทำให้เกิดการเปลี่ยนแปลงรายละเอียดใน WM
- การทำงานในลักษณะนี้จะเกิดขึ้นเป็นรอบๆ ไปจนกระทั่งไม่มี production ใดที่จะสัมพันธ์กับรายละเอียดที่มีใน WM จึงหยุดการทำงาน


PRODUCTION MEMORY
CONDITION → ACTION
RECOGNITION
ACT
CYCLE
WORKING MEMORY
(WORLD state)
OUTPUT
INPUT
OUTSIDE
WORD

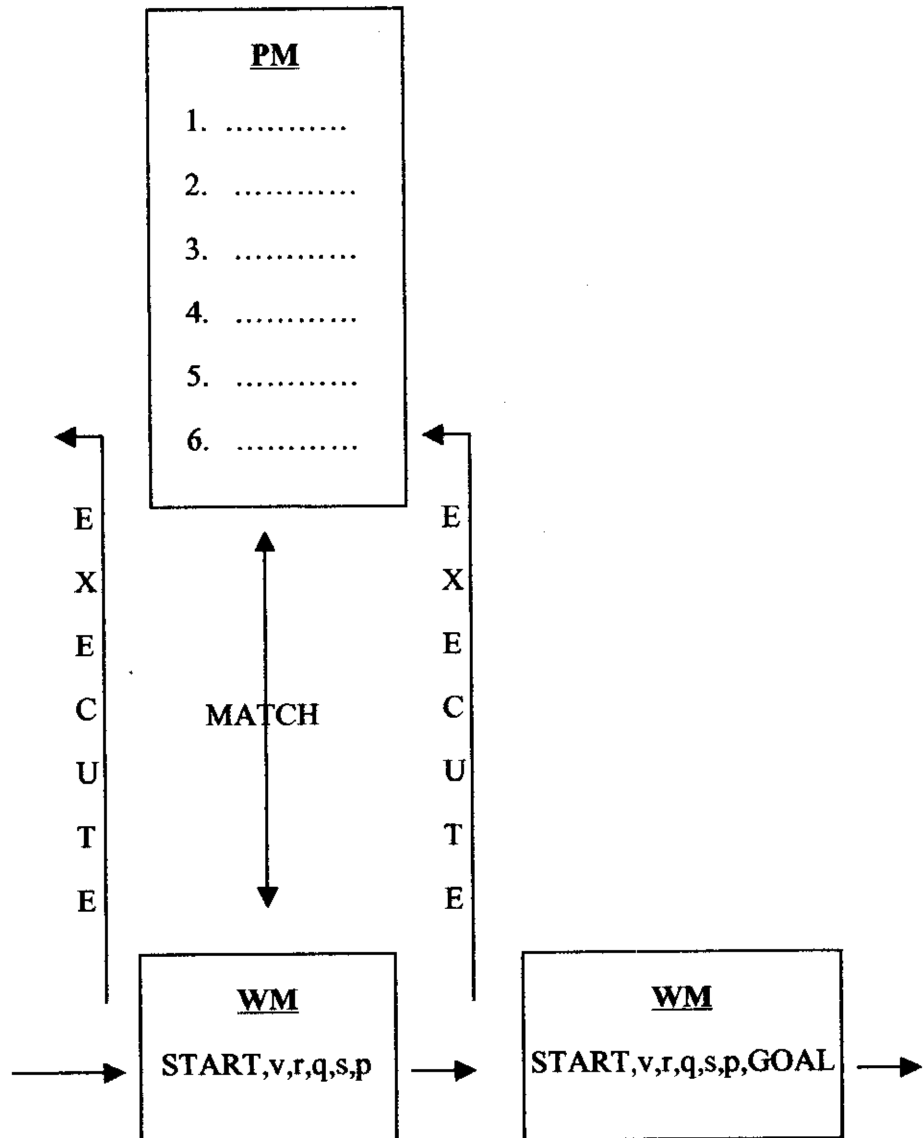
- **<u>รูปแบบโครงสร้างของ PS</u>**

<u>ตัวอย่าง</u> การทำ data-driven search ใน PS



TRACE EXECUTION :

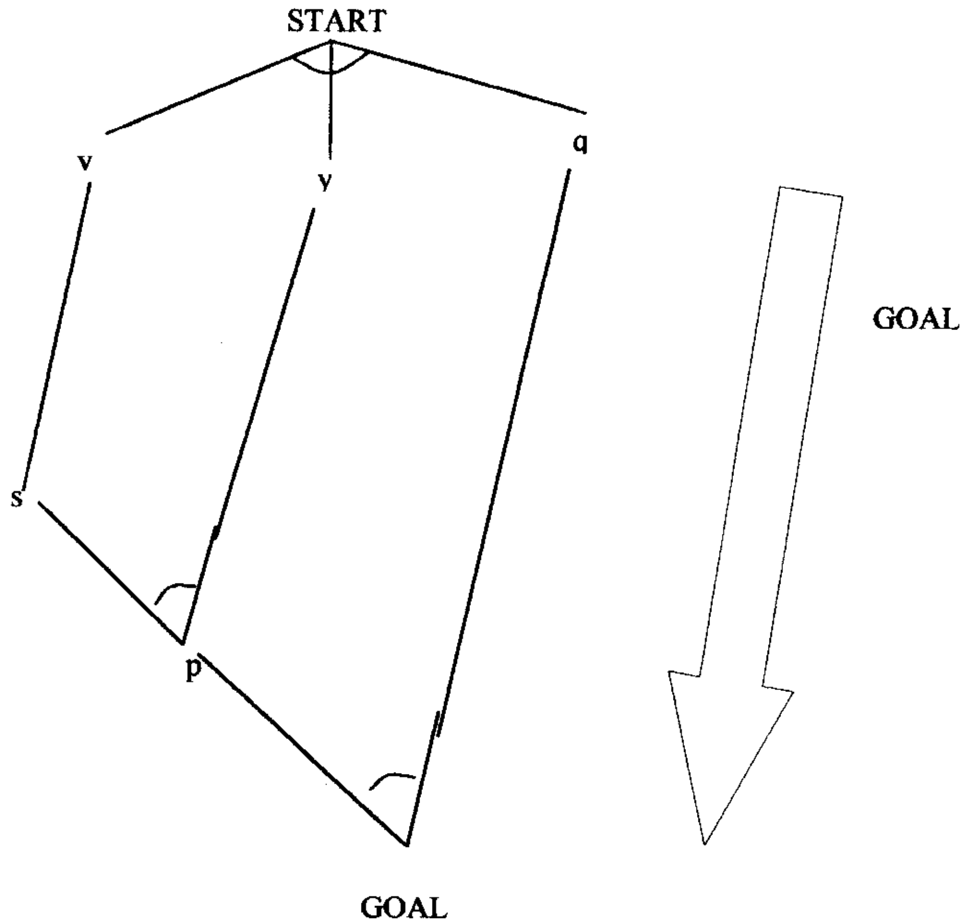
ITERATION#	WM	CONFLICT SET	RULE FIRED
0	START	6	6
1	START,v,r,q	6,5	5
2	START,v,r,q,s	6,5,2	2
3	START,v,r,q,s,p	6,5,2,1	1
4	START,v,r,q,p,GOAL	6,5,2,1	HALT



CT 488

87

Space searched by execution :



<u>ตัวอย่าง</u> การทำ goal-driven search ใน PS

PRODUCTION SET

1. $p \wedge q \rightarrow GOAL$
2. $r \wedge s \rightarrow p$
3. $w \wedge r \rightarrow q$
4. $t \wedge u \rightarrow q$
5. $v \rightarrow s$
6. $START \rightarrow v \wedge r \wedge s$

TRACE OF EXECUTION :

ITERATION#	WORKING MEMORY	CONFLICT SET	RULE FIRED
0	GOAL	1	1
1	GOAL,p,q	1,2,3,4	2
2	GOAL,p,q,r,s	1,2,3,4,5	3
3	GOAL,p,q,r,s,w	1,2,3,4,5	4
4	GOAL,p,q,r,s,w,t,u	1,2,3,4,5	5
5	GOAL,p,q,r,s,w,t,u,v	1,2,3,4,5,6	6
6	GOAL,p,q,r,s,w,t,u,v, START	1,2,3,4,5,6	HALT