บทที่ 10
กรณีศึกษา
(Case study)

ทฤษฎีโปรแกรมหมากรุกไทย

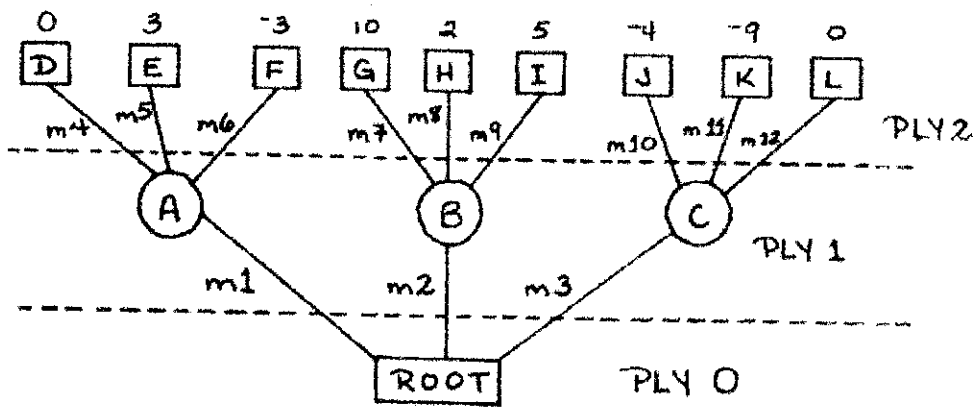
หมากรุกไทย เป็นเกมการเล่นอย่างหนึ่งซึ่งมีมาตั้งแต่ช้านานแล้ว ซึ่งน่าจะมีต้นกำเนิดมาจากประเทศอินเดีย โดยเกมนี้จะมีผู้เล่น 2 ฝ่าย ต่างฝ่ายต่างมีตัวหมากรุก 16 ตัว คือ เบี้ยจำนวน 8 ตัว เรือจำนวน 2 ตัว ม้าจำนวน 2 ตัว โคนจำนวน 2 ตัว ขุน 1 ตัวและเม็ดอีก 1 ตัว ซึ่งวิธีการเล่นนั้นผู้เล่นต่างฝ่ายจะผลัดกันเดินคนละที ฝ่ายใดสามารถรุกขุนอีกฝ่าย โดยที่ขุนฝ่ายที่ถูกรุกไม่สามารถหาทางหนีได้ถือว่าชนะ ซึ่งจากวิธีการเล่นที่มีกฎกติกาที่รัดกุมและถือได้ว่าเป็นเกมที่เรียกว่า Perfect information ในลักษณะของทฤษฎีเกม โดยมีผู้เล่น 2 ฝ่ายค่าของเกมเป็นศูนย์ (Two persons Zero sum game) จึงสามารถนำกฎพื้นฐานเหล่านี้มาเขียนเป็นเกมได้ โดยอาศัยทฤษฎีหรือหลักที่เกี่ยวข้องดังนี้

1. Minimax Search
2. Alpha – Beta Cutoff
3. Quiescence Search
4. Static Evaluation
5. Board Representation
6. Makemove & Unmakemove

Minimax Search

เป็นขั้นตอนวิธีการค้นหาอย่างง่ายที่ใช้กับโครงสร้างข้อมูลแบบต้นไม้ โดยหลักการแล้วจะเริ่มค้นหาจากราก (root) ของต้นไม้และค้นลึกลงไปเพื่อหาค่าคาดหวังมากที่สุดจากน้อยที่สุดที่คาดว่าจะได้

CT 488 209



จากรูปพิจารณาที่ตำแหน่งราก (root) สมมติเป็นเหตุการณ์ของฝ่ายขาวเล่น ซึ่งฝ่ายขาวมีทางเลือกทั้งหมด 3 ทางเลือก คือ m1, m2, m3 ถ้าฝ่ายขาวเดินไปยังบัพ (node) A แล้วฝ่ายดำจะต้องเลือกเดินระหว่าง m4, m5, m6 โดยที่ดำจะต้องเลือกที่จะเสียต้องที่สุด นั้นคือ (-3) แต่ถ้าฝ่ายขาวเดินไปยังบัพ B แล้วฝ่ายดำจะเลือกเดินได้ในเส้นทาง m7, m8, m9 ซึ่งค่าสูญเสียน้อยที่สุดคือ 2 และถ้าฝ่ายขาวเดินไปยังบัพ C แล้วฝ่ายดำจะเลือกเดินได้คือ m10, m11, m12 ซึ่งค่าสูญเสียน้อยที่สุดคือ –9 ดังนั้น ค่ามากที่สุดจากน้อยที่สุดตามกฎของ minimax ก็คือ 2 นั้นเอง ดังนั้นฝ่ายขาวจึงเลือกตาเดิน m2 เป็นเส้นทางที่ดีที่สุด เพราะด้วยคาดหวังว่าฝ่ายดำจะเดิน m8 เพื่อให้ฝ่ายดำสูญเสียน้อยที่สุด

Pseudo code for the minimax search method

```
int Search (int depth, int ply)
{
  int score, bestscore=-INFINITY;

  if (depth less than 1) {
    this is a "leaf." we need to call an
    evaluation function to get a score
    for this position.

    return score for this position
  }

  generate the list of possible moves

  for (every move) {
    make move
    score = -Search(depth-1,ply+1);
    unmake move
    if (score greater than bestscore) {
```

```
            bestscore=score;
            the best move = current move
            update the principal continuation
          }
        }

        return bestscore;
      }
```

พิจารณาจากตัวอย่างต่อไปนี้

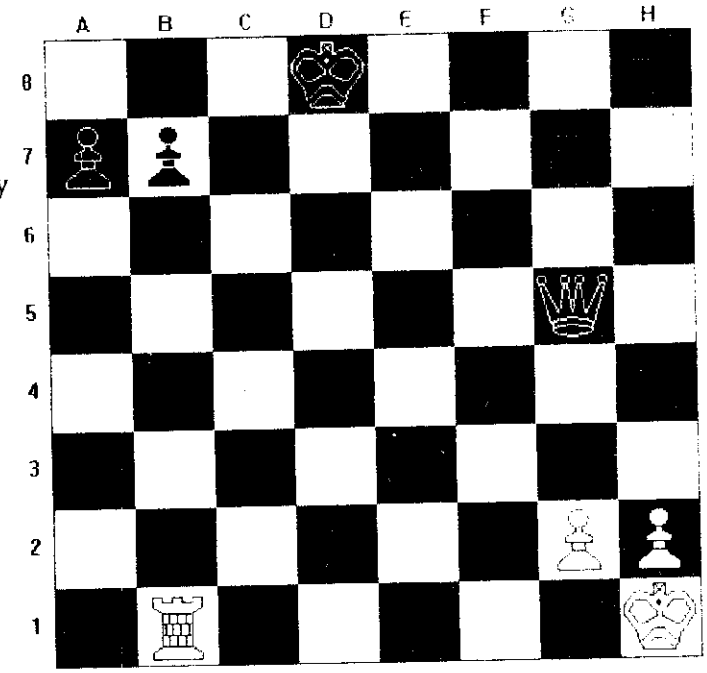
จากภาพฝ่ายขาวเป็นผู้เล่นก่อน

พิจารณาที่ความลึก (depth) 3 ชั้น ณ จุดนี้

เราเรียกว่าเป็นรากของต้นไม้ดังนั้นค่า ply

จึงเท่ากับศูนย์ค่าตัวแปรเริ่มต้นถูก

กำหนดค่าดังนี้



ply = 0

depth = 3

bestscore = -INFINITY

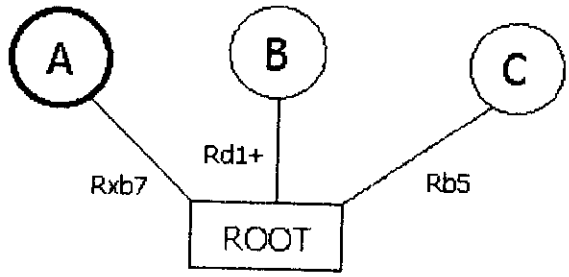
สำหรับส่วนแรกของขั้นตอนวิธีกล่าวว่า

```
if (depth < 1){
  return eval;
}
```

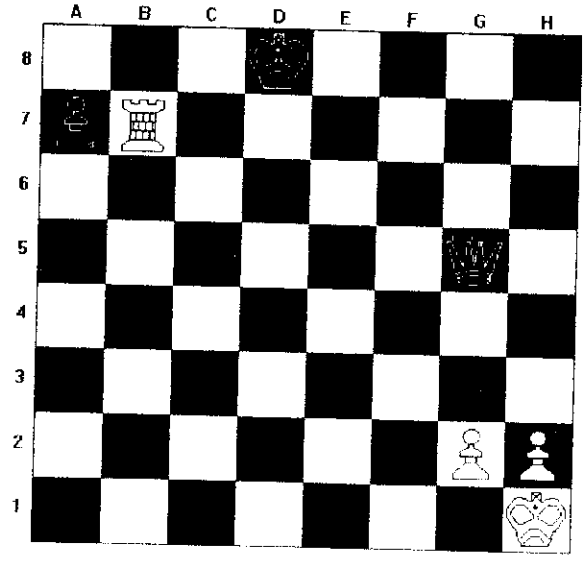
ซึ่งในที่นี้ความลึกของเราเป็น 3 ดังนั้นในชั้นนี้จึงไม่มีการส่งค่าใด ๆ กลับไป และต้องมาทำในขั้นตอนต่อไปคือ

CALL A FUNCTION TO GENERATE ALL POSSIBLE MOVES.

เป็นการเรียกฟังก์ชันในการสังเคราะห์ตาเดินที่สามารถเดินได้ทั้งหมดของผู้เล่นซึ่งในที่นี้สมมติว่าได้ค่าตาเดินคือ : Rxb7 (เอาเรือกินเบี้ยตาเดิน b7), Rd1+ (เอาเรือรุกขุนที่ตาเดิน d1), Rb5 (เอาเรือไปตาเดิน b5) ซึ่งเราสามารถสร้างเป็นต้นไม้ได้ดังนี้คือ



พิจารณาตาเดินแรกก่อนคือ Rxb7 เราจะได้ผลลักษณะนี้คือ



ซึ่งในลำดับต่อไปจะต้องมีการเรียกขั้นตอนวิธีแบบเวียนบังเกิดอีกครั้ง (Recursive algorithms)

score = -search(depth-1,ply+1);

สาเหตุที่ต้องให้ค่า score = -Seach เพราะว่าในการเดินครั้งต่อไปนั้นฝ่ายดำจะเป็นผู้เดินซึ่งฝ่ายดำ ดังนั้นต้องยึดถือหลักความจริงที่ว่า

-opponent score = our score

opponent score = -our score

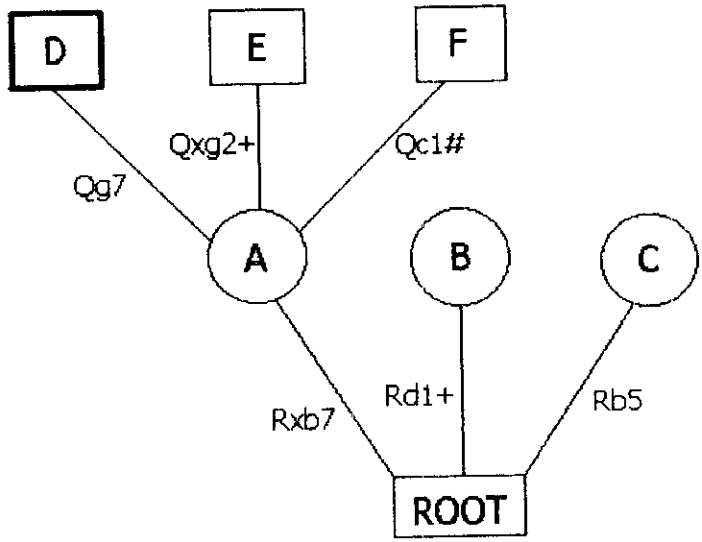
ซึ่งการเรียกตัวเองอีกครั้งนี้จะได้ค่าเริ่มต้นคือ

ply = 1

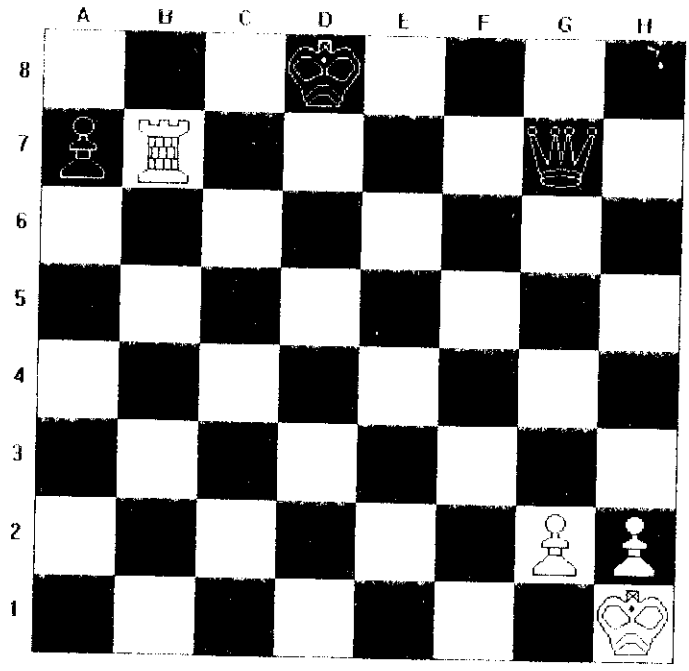
depth = 2

bestscore = -INFINITY

ซึ่งค่าความลึกก็ยังไม่ได้น้อยกว่า 1 ดังนั้นจึงยังต้องเรียกฟังก์ชันในการสังเคราะห์เส้นทางเดินโดยคราวนี้จะสังเคราะห์เส้นทางเดินของฝ่ายดำ ซึ่งสมมติว่าได้ดังนี้คือ Qg7, Qxg2+, Qc1#.



ซึ่งลำดับต่อไปที่จะถูกพิจารณาคือ Qg7 ด้วยเหตุที่ว่าเราค้นแบบ Depth first search ผลลัพธ์ของการเดินของฝ่ายดำไปที่ Qg7 กรณีที่ขาวเดิน Rxb7 ก็คือ



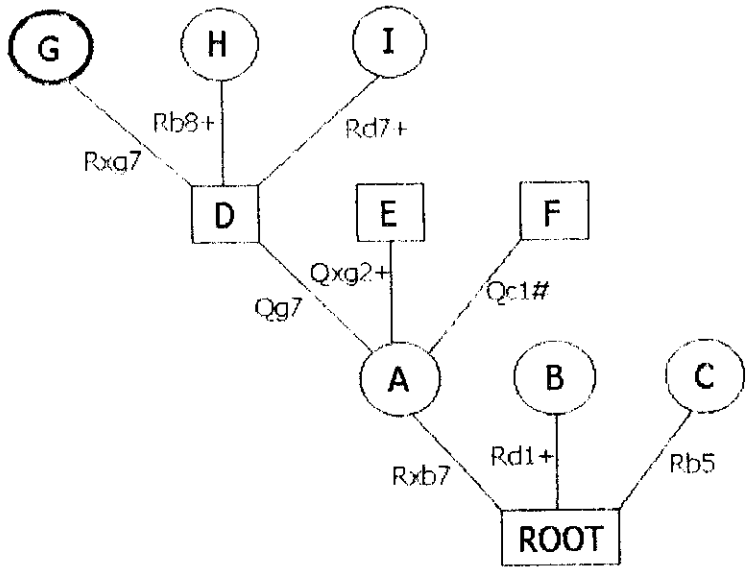
จากนั้นจะถูกเรียกใช้งาน Search อีกครั้งโดยมีค่าเริ่มต้นคือ

ply = 2

epth = 1

bestscore = -INFINITY

และค่าความลึกยังไม่น้อยกว่า 1 ดังนั้นจึงต้องสังเคราะห์ตาเดินสมมติได้ Rxg7, Rb8+, Rd7+.



พิจารณาที่ตาเดิน Rxg7 เราจะต้องเรียก Search อีกครั้งพร้อมทั้งกำหนดค่าเริ่มต้นคือ

```
ply = 3
depth = 0
bestscore = -INFINITY

if (depth < 1) {
   return eval();
}
```

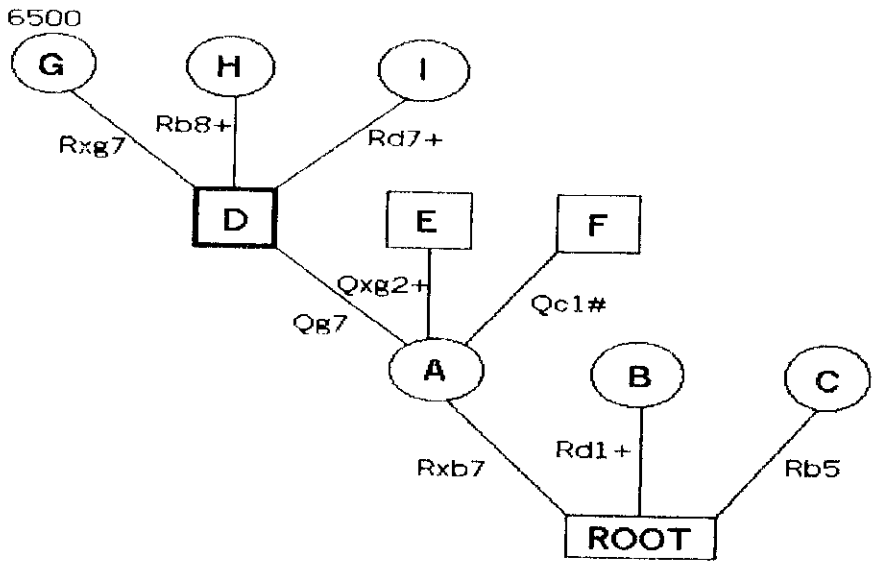
ซึ่งในตอนนี้ขั้นตอนทดสอบความลึก depth < 1 ผ่าน ฟังก์ชันประเมินค่าจะต้องถูกเรียกเพื่อพิจารณาความได้เปรียบเสียเปรียบ สำหรับในที่นี้ฝ่ายดำจะส่งค่ากลับไปคือ 6500 หลังจากนั้นค่านี้จะถูกส่งกลับไปที่บัพ D และคำสั่งต่อไปจะถูกประมวลผลต่อคือ

```
if (score > bestscore) {
   bestscore = score;
```

update the pc[][] array with this move.

}

ซึ่งจากตัวอย่างเราจะได้ค่า score = 6500 ดังนั้นค่า bestscore จึงถูกปรับเป็น 6500



และจะไปเรียกฟังก์ชัน Search อีกครั้งเพื่อค้นหาค่าของเส้นทาง Rb8+ ซึ่งจะทำอย่างนี้ไปเรื่อย ๆ ดังนั้นปัญหาที่สำคัญก็คือ Minimax จะเสียเวลาอย่างมากในการค้นหาเพราะในความเป็นจริงแล้วแต่ละฝ่ายมี 16 ตัวแต่ละตัวนั้นมีเส้นทางเดินได้หลายเส้นทางโดยเฉลี่ยแต่ละฝ่ายมีประมาณ 30 เส้นทางเดิน ดังนั้นหากเราทำการค้นหาในความลึกมาก ๆ จะพบว่ามีบัพที่ต้องคนหาจำนวนมากมายมหาศาล ดังนั้นวิธีการ minimax จึงเป็นวิธีการที่ไม่ดีในการนำมาใช้กับการโปรแกรมหมากรุก พิจารณาตารางต่อไปนี้ กำหนดให้แต่ละฝ่ายมีตาเดินโดยเฉลี่ย 30 ตาเดิน ในการค้นหาโปรแกรมจะสามารถค้นหาได้ 50,000 ตาเดินต่อ 1 วินาที ซึ่งก็จะได้ผลลัพธ์ที่ละดับความลึกต่าง ๆ ดังนี้

Depth (ply)	Number of positions	Time to Search
2	900	0.018 s
3	27,000	0.54 s
4	810,000	16.2 s
5	24,300,000	8 minutes
6	729,000,000	4 hours
7	21,870,000,000	5 days

Alpha-Beta Cutoff Or Alpha-beta pruning

วิธีการของ Alpha Beta นั้นเป็นแนวคิดที่จะใช้พัฒนาการค้นหาให้ดีขึ้นอีกระดับ โดยความเป็นจริงแล้วในการเล่นหมากรุกเราจะไม่พิจารณาตาเดินที่ไม่ดีหรือว่าเสียเปรียบ เช่น การเอาเรือไปแลกกับเบี้ย หรือเดินตัวหมากรุกไปให้อีกฝ่ายกินเปล่า ๆ ดังนั้นหลักการนี้จึงนำมาใช้กับการค้นหาแบบ Minimax โดยแนวคิดคือ หากเราทราบว่าในการค้นลึกไปในต้นไม้แบบ Depth First Search แล้วเราพบว่าในการค้นหาบัพลูกลำดับแรกได้ค่าคาดหวังเท่ากับ 10 และในระดับเดียวกันกับบัพลูกอีกบัพหนึ่งได้ค่าคาดหวังคือ 11 แสดงว่าในเส้นทางนี้เราได้คาดคาดหวังสูงสุดคือ 10 (อีกฝ่ายย่อมต้องเลือกหนทางที่สูญเสียน้อยที่สุด) และมือเราย้อนกลับไปเพื่อจะค้นหาอีกเส้นทางหนึ่งปรากฏว่าเราพบว่าค่าคาดหวังที่พบคือ 9 แสดงว่าเราไม่จำเป็นต้องค้นหาในลำดับต่อไปแล้วในระดับนี้ เพราะหมายถึงค่าที่ค้นเจอ 9 นี้ต่ำกว่าค่าที่เราพบครั้งก่อนคือ 10 นั้นหมายถึงเราไม่ต้องไปหาต่อเพราะหาต่อไปเราก็คาดว่าคู่ต่อสู้ก็จะต้องเลือกทางที่เสียน้อยคือ 9 ดังนั้นจึงไม่ค้น และนี่ก็คือแนวความคิดในเรื่องการใช้ค่า Alpha Beta มาทำการ Cutoff เส้นทางที่ไม่จำเป็นทิ้งไป เพื่อความเข้าใจขอให้พิจารณาจากรูปและตัวอย่างโปรแกรมประกอบ

ค่า Alpha ก็คือค่าคาดหวังมากที่สุดจากน้อยที่สุดใช้สำหรับถ่วงน้ำหนักสำหรับฝ่ายหาค่า Min

ค่า Beta จะเป็นค่าคาดหวังมากที่สุดจากน้อยที่สุดใช้สำหรับถ่วงน้ำหนักสำหรับฝ่ายหาค่า Max

```
/* alpha-beta.c - Discussion at
     http://www.cis.temple.edu/~ingargio/cis587/readings/alpha-beta.html
 */

#include <stdio.h>
#include <stdlib.h>

enum kind {Min, Max};

/* A node can have any number of successors.
   We represent the tree using a binary tree. The link child goes
   to one of the successors, and from there the link sibling will
   take to all other successors.
 */

struct node {
   char * name;    /* used only for convenience in displaying information */
   enum kind knd;  /* if the node is a Max node or a Min node */
   int value;      /* relevant only for leaves */
   int alpha;
   int beta;
   struct node *child;
   struct node *sibling;
};
```

```
/* preorder traversal of the tree */
void printNode(const struct node * const p)
{
 const struct node * r;

 for (r = p; r != NULL; r = r->sibling) {
  printf ("name = %s\t kind = %d\t value = %d\t alpha = %d\t beta =
                %d\n", r->name, (int)(r->knd),
          r->value, r->alpha, r->beta);
  printNode(r->child);
 }
}

#define min(x,y) ((x)<(y))?x:y
#define max(x,y) ((x)>(y))?x:y

/* The Minimax algorithm with Alpha-Beta cutoff */
int minimaxAB (struct node * n, int a, int b)
/* Here A is always less than B */
{
 if (n->child == NULL)
  return n->value;
 n->alpha = INT_MIN;
 n->beta = INT_MAX;
 if (n->knd == Min) {
  struct node *r;
  for (r = n->child; r != NULL; r = r->sibling) {
   int newb = min(b, n->beta);
```

```
      int val = minimaxAB(r, a, newb);
            printf("minimax at %s, with a = %d, b = %d is %d\n",
                  r->name, a, newb, val);
      n->beta = min(n->beta, val);
      if (a >= n->beta)       break;
    }
    return n->beta;
  } else {
    struct node *r;
    for (r = n->child; r != NULL; r = r->sibling) {
      int newa = max(a, n->alpha);
      int val = minimaxAB(r, newa, b);
            printf("minimax at %s, with a = %d, b = %d is %d\n",
                  r->name, newa, b, val);
      n->alpha = max(n->alpha, val);
      if (n->alpha >= b) break;
    }
    return n->alpha;
  }
}

/* Just an example of a game */
struct node game[31] = {
/* 0 */   {"A",  Max, 0, INT_MIN, INT_MAX, (struct node *)(game+1),
            (struct node *)NULL},
/* 1 */   {"B",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+3),
                  (struct node *)(game+2)},
```

```
/* 2 */  {"Q",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+5),
                (struct node *)NULL},
/* 3 */  {"C",  Max, 0, INT_MIN, INT_MAX, (struct node *)(game+7),
                (struct node *)(game+4)},
/* 4 */  {"J",  Max, 0, INT_MIN, INT_MAX, (struct node *)(game+9),
                (struct node *)NULL},
/* 5 */  {"R",  Max, 0, INT_MIN, INT_MAX, (struct node *)(game+11),
                (struct node *)(game+6)},
/* 6 */  {"Y",  Max, 0, INT_MIN, INT_MAX, (struct node *)(game+13),
                (struct node *)NULL},
/* 7 */  {"D",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+15),
                (struct node *)(game+8)},
/* 8 */  {"G",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+17),
                (struct node *)NULL},
/* 9 */  {"K",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+19),
                (struct node *)(game+10)},
/*10 */  {"N",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+21),
                (struct node *)NULL},
/*11 */  {"S",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+23),
                (struct node *)(game+12)},
/*12 */  {"V",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+25),
                (struct node *)NULL},
/*13 */  {"Z",  Min, 0, INT_MIN, INT_MAX, (struct node *)(game+27),
                (struct node *)(game+14)},
/*14 */  {"Z3", Min, 0, INT_MIN, INT_MAX, (struct node *)(game+29),
                (struct node *)NULL},
/*15 */  {"E",  Max, 10, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)(game+16)},
```

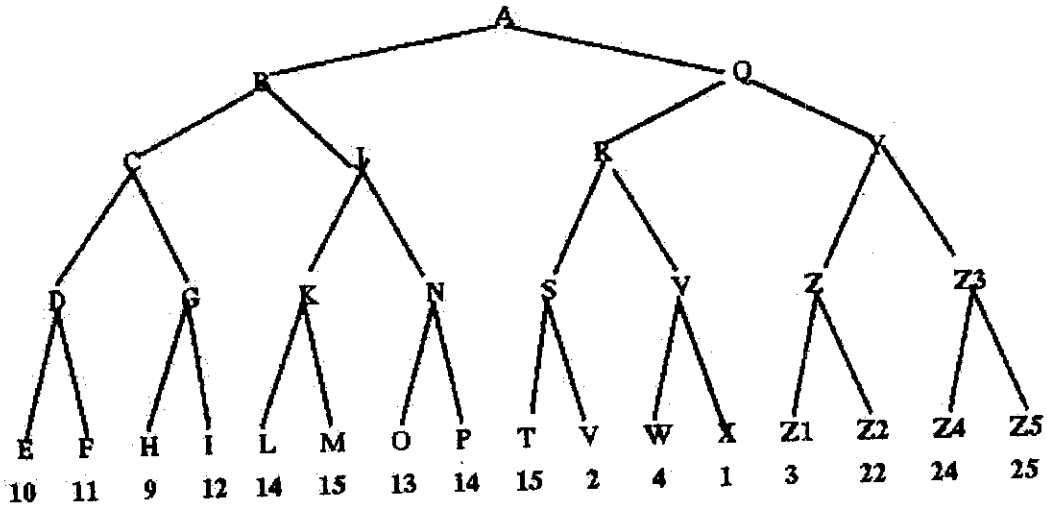
```
/*16 */  {"F", Max, 11, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)NULL},
/*17 */  {"H", Max,  9, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)(game+18)},
/*18 */  {"I", Max, 12, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)NULL},
/*19 */  {"L", Max, 14, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)(game+20)},
/*20 */  {"M", Max, 15, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)NULL},
/*21 */  {"O", Max, 13, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)(game+22)},
/*22 */  {"P", Max, 14, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)NULL},
/*23 */  {"T", Max, 15, INT_MIN, INT_MAX, (struct node *)NULL,
                (struct node *)(game+24)},
/*24 */  {"U", Max,  2, INT_MIN, INT_MAX, NULL, NULL},
/*25 */  {"W", Max,  4, INT_MIN, INT_MAX, NULL, game+26},
/*26 */  {"X", Max,  1, INT_MIN, INT_MAX, NULL, NULL},
/*27 */  {"Z1", Max,  3, INT_MIN, INT_MAX, NULL, game+28},
/*28 */  {"Z2", Max, 22, INT_MIN, INT_MAX, NULL, NULL},
/*29 */  {"Z4", Max, 24, INT_MIN, INT_MAX, NULL, game+30},
/*30 */  {"Z5", Max, 25, INT_MIN, INT_MAX, NULL, NULL} };
int main()
{
 int value;

/* Example from Nillson: Principles of AI, 1980, page 124 */
```

```
printNode(game);
value = minimaxAB(game, INT_MIN, INT_MAX);
printf("\nValue is: %d\n", val);

return 0;
}
```

จะเห็นว่าผู้เล่นฝ่าย Max (คือเริ่มที่โหนด A) ต้องการหาค่า Minimax ที่คาดหวังว่าจะได้รับ ดังนั้นจึงถ่วงน้ำหนักค่า Alpha เป็นค่าติดลบ และถ่วงน้ำหนักค่า Beta เป็นค่าบวก ซึ่งค่าที่ถ่วงนี้ต้องอยู่ในที่ค้นหา เพราะค่าที่ได้นั้น (value) จะเป็นค่าที่อยู่ในช่วง $INT\_MIN \leq value \leq INT\_MAX$



Here is a trace of the execution of the Minimax strategy with Alpha Beta Cutoff

NODE	TYPE	A	B	ALPHABETA		SCORE
A	Max	-I	+I	-I	+I	
B	Min	-I	+I	-I	+I	
C	Max	-I	+I	-I	+I	
D	Min	-I	+I	-I	+I	
E	Max	-I	+I			10

D	Min	-I	+I	-I		10
F	Max	-I	10			11
D	Min	-I	+I	-I	10	10
C	Max	-I	+I	10	+I	
G	Min	10	+I	-I	+I	
H	Max	10	+I			9
G	Min	10	+I	-I	9	9
C	Max	-I	+I	10	+I	10
B	Min	-I	+I	-I	10	
J	Max	-I	10	-I	+I	
K	Min	-I	10	-I	+I	
L	Max	-I	10			14
K	Min	-I	10	-I	14	
M	Max	-I	10		15	
K	Min	-I	10	-I	14	14
J	Max	-I	10	14	+I	14
B	Min	-I	+I	-I	10	10
A	Max	-I	+I	10	+I	
Q	Min	10	+I	-I	+I	
R	Max	10	+I	-I	+I	
S	Min	10	+I	-I	+I	
T	Max	10	+I			15
S	Min	10	+I	-I	15	
V	Max	10	+I			2
S	Min	10	+I	-I	2	2
R	Max	10	+I	2	+I	
Y	Min	10	+I	-I	+I	
W	Max	10	+I		4	

Y	Min	10	+I	-I	4	4
R	Max	10	+I	4	+I	4
Q	Min	10	+I	-I	4	4
A	Max	-I	+I	10	4	10

Quiescence Search

ในการเล่นหมากรุกนั้นค่อนข้างมีเทคนิคที่สิ่งสำคัญค่อนข้างมาก เช่นในกรณีมีการกินกันเกิดขึ้น ฝ่ายหนึ่งอาจเอาเรือไปกินม้าอีกฝ่ายเพราะเห็นว่าฟรี ซึ่งหากมีการกินกันเกิดขึ้นแล้วเรือลำดังกล่าวอาจไปถูกขังไว้ก็ได้ หรืออาจโดนรุกฆาต ซึ่งก็จะกลายเป็นเสียเปรียบ ซึ่งเหตุการณ์เหล่านี้ในการค้นหาด้วย Alpha-Beta นั้นไม่สามารถรองรับมือได้ ดังนั้นจึงมีการใช้ฟังก์ชันตัวหนึ่งซึ่งเราเรียกกันว่า **quiescent search** สิ่งที่สำคัญสำหรับฟังก์ชันนี้ก็คือการประเมินค่า (Evaluation) ซึ่งจะเป็นการประเมินว่าหากมีการกินกันจริงฝ่ายใดจะได้เปรียบเสียเปรียบอย่างไรหรือมีผลอย่างไร ซึ่งมีอัลกอริทึมดังนี้

```
int Quies(int alpha, int beta)
{
   val = Evaluate();
   if (val >= beta)
      return beta;
   if (val > alpha)
      alpha = val;
   GenerateGoodCaptures();
   while (CapturesLeft()) {
      MakeNextCapture();
      val = -Quies(-beta, -alpha);
      UnmakeMove();
      if (val >= beta)
         return beta;
```

```
        if (val > alpha)
            alpha = val;
    }
    return alpha;
}
```

หากดูรหัสโปรแกรมแล้วอาจเห็นว่าจะใกล้เคียงกับฟังก์ชัน Alpha-Beta แต่ก็จะมีจุดที่แตกต่างกันหลายจุด เช่น ในตัวฟังก์ชันนี้จะไปทำการเรียกฟังก์ชันประเมินค่า (Evaluation Function) เพื่อดูคะแนนว่าเหมาะสมเพียงพอที่จะมีการกินกันหรือไม่ ซึ่งโดยทั่วไปแล้วจะใช้เทคนิคอื่นควบคู่กันไปด้วยเช่นในกรณี โปรแกรมหมากรุกไทยเวอร์ชันมือใหม่นั้น ได้อาศัยเทคนิคที่เรียกว่า MVV/LVA ซึ่งย่อมาจาก Most Valuable Victim/Least Valuable Attacker แนวความคิดก็คือจะมีการเรียงลำดับการค้นหาและการกินจากคะแนนเพื่อดูว่ากินตัวใดจะได้เปรียบที่สุด

ประโยชน์จากเทคนิค MVV/LVA คือง่ายต่อการประยุกต์ใช้งานและผลลัพธ์ให้ได้ดี (and it results in a high nodes/second) ส่วนข้อด้อยคือถ้าการค้นหาของเราไม่ดีเราจะเสียเวลาในการค้นหาไปอย่างมากกับการประเมิณค่า

Static Evaluation

ในการค้นหานั้นทุกครั้งที่เราค้นไปยังจุด ๆ หนึ่งแล้ว จะต้องมีการประเมินค่าผลลัพธ์ที่ได้ ซึ่งในโปรแกรมหมากรุก ฯ ก็เช่นกัน เมื่อเราค้นหาจนถึงปลายทางก็ต้องมีการประเมินค่า ซึ่งค่าเหล่านี้เกิดขึ้นจากความคิดเห็นของแต่ละบุคคลหรือแต่ละตำรา อาจกล่าวได้ว่าค่านี้ก็คือ knowledge BASE ของโปรแกรมหมากรุก ฯ ก็เป็นได้ จริงๆ แล้วในการประเมินค่านั้นมีด้วยกัน 2 ลักษณะคือ

1. Static Evaluation
2. Dynamic Evaluation

แต่ที่นิยมใช้กันมากจะเป็นแบบ Static Evaluation เพราะด้วยเหตุที่ว่าในการค้นหาจะเกิดบัพปลายทางจำนวนมาก และยิ่งหากค้นลึกลงไปมาก ๆ ก็จะยิ่งทวีคูณ ซึ่งตรงนี้ทำให้เราเสียเวลาในการค้นลงเยอะมาก และหากยังเสียเวลากับการคำนวณ Evaluation (ประเมินค่า) แบบพลวัตแล้วก็

จะทำให้ยิ่งเสียเวลามาก ๆ ดังนั้น จึงมีการให้ค่าตำแหน่งของตัวหมากรุกลงไปเลยว่าถ้าตัวนี้อยู่บนช่องนี้แล้วควรจะให้ค่าประมาณเท่าใด ซึ่งค่า ๆ นี้นั้นจะเป็นค่าที่ค่อนข้างเป็นกว้างหรือมีความยืดหยุ่นสูงหน่อย ซึ่งโปรแกรมหมากรุกไทยนั้นยังไม่อาจรับประกันได้ว่าจริง ๆ แล้วค่าที่ควรใส่ลงไปควรเป็นเท่าใด ส่วนนี้ยังคงต้องให้เกิดการวิจัยหรือทดลองกันต่อไป

สำหรับการทดลองของทางต่างประเทศนั้น จากการทดลองแล้วหากเรามีเครื่องที่มีกำลังมากเพียงพอแล้วการประเมินค่าแบบ Dynamic จะให้ผลค่อนข้างดีกว่ามาก ซึ่งในการประเมินค่าแบบ Dynamic นั้นจะต้องอาศัยตัวแบบเชิงคณิตศาสตร์เข้ามาสร้าง

Board Representation

โครงสร้างที่เราใช้ในการดำเนินการที่สำคัญสำหรับโปรแกรมหมากรุกนั้นก็คือตัวกระดาษซึ่งเป็นข้อมูลเริ่มต้นที่สำคัญ ซึ่งจะบอกถึงว่าเรามีตัวอะไรอยู่บ้าง และอยู่ ณ ตำแหน่งใด สำหรับโครงสร้างที่ใช้ในการทำงานนั้นก็มีหลายแบบ ซึ่งในสมัยก่อน เนื่องจากคอมพิวเตอร์มีข้อจำกัดมากเรื่องหน่วยความจำก็จึงใช้เป็นลักษณะโครงสร้างข้อมูลแบบ Array สองมิติซึ่งก็เป็นรูปแบบหนึ่งที่ใช้กันแต่เนื่องจากการใช้งานหรือการทำงานจริงค่อนข้างเสียเวลาและดูแลยากจึงไม่เป็นที่นิยม ต่อมาก็มีการพัฒนาเป็นลักษณะ Array แบบมิติเดียว เช่นในโปรแกรมหมากรุกไทยตัวนี้นั้นใช้เป็น Array of integer ขนาด 64 ช่อง ซึ่งเท่ากับจำนวนตารางบนกระดานหมากรุกพอดี สำหรับบางโปรแกรมจะมีการใช้ Array ลักษณะ Mailbox ซึ่งหมายถึงมีการสร้าง Array ไว้สองชุด ซึ่งชุดแรกจะมีขนาด 64 ช่อง และชุดที่สองมีขนาด 120 ช่อง ซึ่งในชุดที่สองนี้เองที่ใช้สำหรับการตรวจสอบว่าตัวหมากรุกที่เดินไกล ๆ ได้นั้นเดินออกนอกกระดานหรือป่าว

-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1

-1	x	x	x	x	x	x	x	x	-1
-1	x	x	x	x	x	x	x	x	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

และวิธีการสุดท้ายที่จะนำเสนอก็คือ Bitboard ซึ่งกำลังเป็นที่นิยมมากในโปรแกรมหมากรุกขนาดใหญ่ชั้นนำ แนวความคิดนี้ก็คือ เราจะเก็บข้อมูลของตัวหมากรุกแต่ละตัวไว้ในเลขจำนวนเต็มขนาด 64 บิต ซึ่งจะมีประโยชน์มากในการค้นหาตำแหน่งและในการสังเคราะห์ตาเดิน ซึ่งในเครื่องคอมพิวเตอร์ระดับ 64 บิตขึ้นไปการปฏิบัติการในระดับบิตจะเป็นทางเลือกที่ดีเพราะว่าสามารถเข้าถึงในด้านฮาร์แวร์ได้รวดเร็ว แต่ปัญหาอย่างหนึ่งของการใช้เทคนิคนี้คือโปรแกรมจะต้องเสียเวลาในการอัพเดตข้อมูลใหม่ทุกครั้งที่มีการเปลี่ยนแปลงตัวหมากรุกบนกระดาน ซึ่งก็คือในช่วงการเรียกใช้ Makemove และ Unmakemove นั้นเอง

Makemove & Unmakemove

เป็นฟังก์ชันที่จะทำการทดลองเคลื่อนย้ายตัวหมากรุกดู โดยในฟังก์ชัน Makemove นั้นจะต้องสามารถเก็บข้อมูลได้ดังนี้ว่า

1. เดินมาจากตำแหน่งไหน
2. เดินไปตำแหน่งไหน
3. มีการกินตัวอะไรไป
4. มีการหงายเบี้ยหรือไม่ (สำหรับเบี้ยในหมากรุกไทย)
5. คะแนนสำหรับตาเดินนี้คือเท่าไร (ใช้สำหรับเป็นค่าเลือกเพื่อจะใช้ในการค้นหาก่อน)

ซึ่งในฟังก์ชัน Unmakemove ก็จะทำงานคล้าย ๆ กันเพียงแต่เป็นการทำงานย้อนหลังกลับเท่านั้นเอง สิ่งที่สำคัญก็คือในการทดลองเดินนั้นจำเป็นจะต้องตรวจดูด้วยว่าถ้าตัวที่เดินเป็นขุนจะเดินไปตำแหน่งที่ถูกรุกไม่ได้ ต้องมีการตรวจสอบกฎด้วยเสมอ และโครงสร้างข้อมูลที่นำมาใช้กับ Makemove และ Unmakemove ส่วนมากนิยมใช้เป็น Array of structure เพราะทำได้ง่ายและดูแลความผิดพลาดได้ไม่ลำบากมาก

Generate Legal Moves

เป็นฟังก์ชันที่ทำหน้าที่ในการสังเคราะห์ตาเดินโดยดูว่าฝ่ายใดเป็นฝ่ายเดินก็จะทำการสังเคราะห์ตาเดินของฝ่ายนั้นๆ เก็บเอาไว้ในโครงสร้างข้อมูลสักชุดเพื่อเอาไว้ทำการทดลองเดินดู ขอให้พิจารณาจากโปรแกรมภาษา C++ ดังนี้

```
if( SideToMove == WHITE )// ในกรณีที่ขาวเป็นฝ่ายเดิน
{
    while( i < 64 ) // จะทำการตรวจสอบทุกช่องบนกระดานเพื่อดูว่าตัวขาวอยู่ตรงไหนบ้าง
    {
        if( b->get_color(i) == BLACK || b->get_color(i) == EMPTY )
        {
            // ถ้าบนกระดานนั้นเป็นตัวของฝ่ายดำหรือช่องว่างไม่มีตัวเราจะข้ามไปดูช่องใหม่
            i++;
            continue;
        }
        switch( b->get_piece(i) ) // ในกรณีที่เป็นตัวของฝ่ายขาวจะดูว่าเป็นตัวอะไร
        {
            case PAWN : // เบี้ย ก็จะทำการดูตามกฎว่าเบี้ยเดินอย่างไรกินอย่างไร
                if( (b->get_color(i-8) == EMPTY) &&
                    (WPawnAttack[i][i-8]))
                    PushTree(i,i-8,TRUE);
/* ในส่วนนี้คือหากพบว่าสามารถเดินได้ตามกฎเราก็จะนำตาเดินนั้นไปเก็บไว้ในโครงสร้างข้อมูล
โดยในที่นี้จะเรียกฟังก์ชัน Pushtree( from, to, promotion) */
                if( (b->get_color(i-9) == BLACK) &&
                    (COL(i) > 0))
                    PushTree(i,i-9,TRUE);
                if( (b->get_color(i-7) == BLACK) &&
```

```
                (COL(i) < 7))
                PushTree(i,i-7,TRUE);
        break;
case KNIGHT :
{
        int offset[8] = {17,15,10,6,-17,-15,-10,-6};
        /* ในส่วนนี้ Offset นั้นเป็นตาเดินที่ม้าสามารถเดินไปได้ */
        int j = j^j;
        while( j < 8 )
        {
                int to = i+offset[j];
                if( (b->get_color(to) != WHITE) &&
                        (KnightAttack[i][to]) && INBOARD(to) )
                        PushTree(i,to,FALSE);
                j++;
        }
} break;
```

/* ในกรณีที่เป็นขุนเดินตัวขุนนั้นจะสามารถเดินถอยหลังได้และเดินไปด้านข้างได้และเหมือนกับโคน ส่วนโคนก็เหมือนเบี้ยผสมกับเม็ด ดังนั้นจึงไม่มีการเรียกใช้คำสั่ง break */

```
case KING :
{
        if( (b->get_color(i+1) != WHITE) &&
                (KingAttack[i][i+1]) && INBOARD(i+1))
                PushTree(i,i+1,FALSE);
        if( (b->get_color(i-1) != WHITE) &&
                (KingAttack[i][i-1]) && INBOARD(i-1))
```

```
                PushTree(i,i-1,FALSE);
        if( (b->get_color(i+8) != WHITE) &&
                (KingAttack[i][i+8]) && INBOARD(i+8) )
                PushTree(i,i+8,FALSE);
}
case CONE :
{
        if( (b->get_color(i-8) != WHITE) &&
                (WConcAttack[i][i-8]) && INBOARD(i-8))
                PushTree(i,i-8,FALSE);
}
case SMED:
case MED :
{
        int offset[4] = {9,7,-9,-7};
        int j = j^j;
        while( j < 4 )
        {
                int to = i+offset[j];
                if( (b->get_color(to) != WHITE) &&
                (MedAttack[i][to]) && INBOARD(to))
                PushTree(i,to,FALSE);
                j++;
        }
} break;
case ROOK:
{
        int to = i;
```

```
while( COL(to) > 0 )
{
        to--;
        if( b->get_color(to) == EMPTY)
                PushTree(i,to,FALSE);
        else
        {
                if( b->get_color(to) != WHITE)
                        PushTree(i,to,FALSE);
                break;
        }
}
to = i;
while( COL(to) < 7 )
{
        to++;
        if( b->get_color(to) == EMPTY)
                PushTree(i,to,FALSE);
        else
        {
                if( b->get_color(to) != WHITE)
                        PushTree(i,to,FALSE);
                break;
        }
}
to = i;
while( ROW(to) < 7 )
{
```

```
                        to += 8;
                        if( b->get_color(to) == EMPTY)
                                PushTree(i,to,FALSE);
                        else
                        {
                                if( b->get_color(to) != WHITE)
                                        PushTree(i,to,FALSE);
                                break;
                        }
                }
                to = i;
                while( ROW(to) > 0 )
                {
                        to -= 8;
                        if( b->get_color(to) == EMPTY)
                                PushTree(i,to,FALSE);
                        else
                        {
                                if( b->get_color(to) != WHITE)
                                        PushTree(i,to,FALSE);
                                break;
                        }
                }
        } break;  // end cal ROOK move
    } // end switch
    i++;
  } // end while
} // end white tomove
```

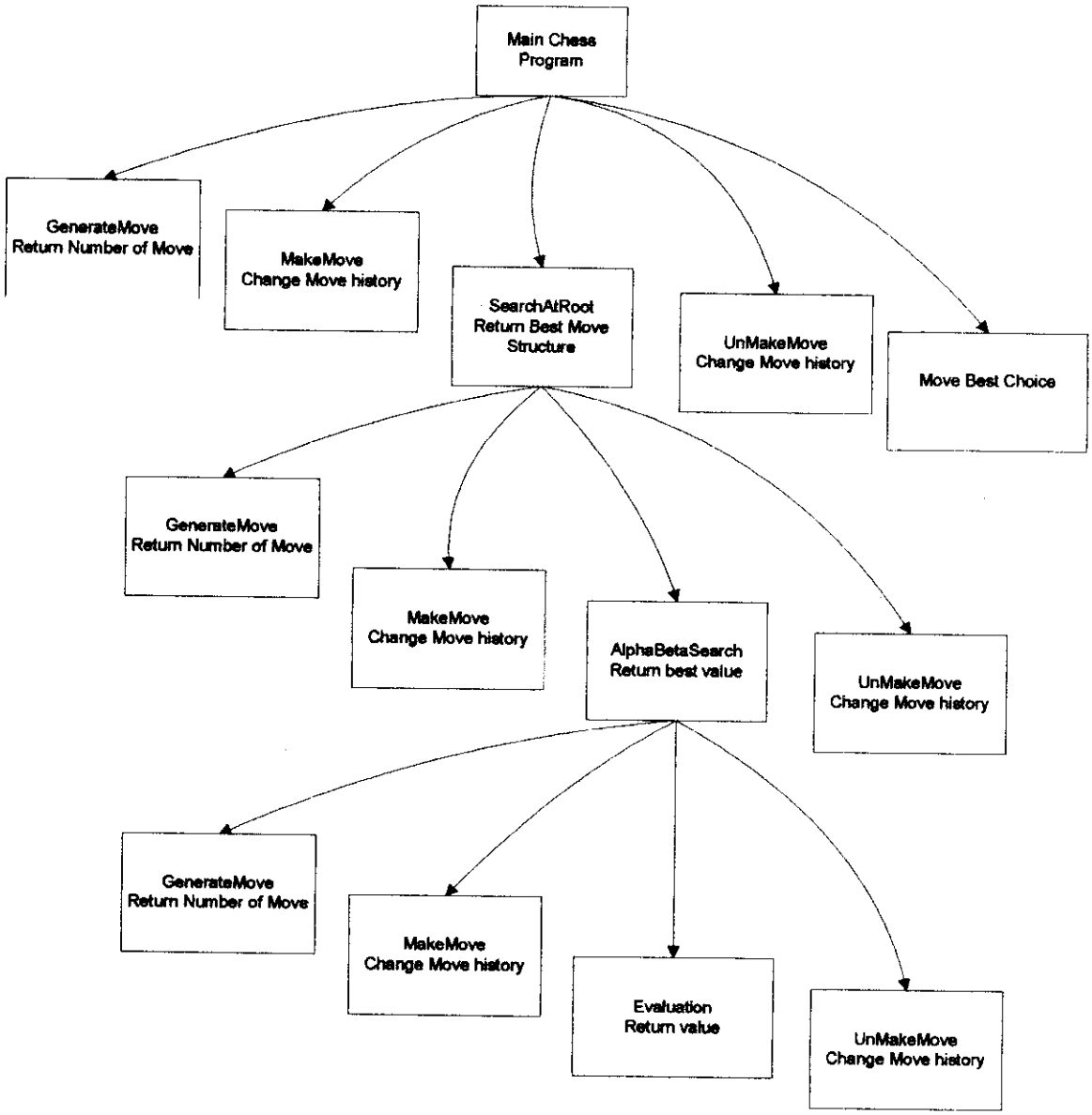
และนี่ก็คือตัวอย่างการสังเคราะห์ตาเดินโดยการสังเคราะห์ตาเดินนั้นจะต้องเป็นไปตามกฎการเล่นของเกมนั้นๆ ด้วย

บทสรุป

หลักที่สำคัญในการสร้างโปรแกรมหมากรุกก็คือจะต้องมีโครงสร้างข้อมูลดังนี้

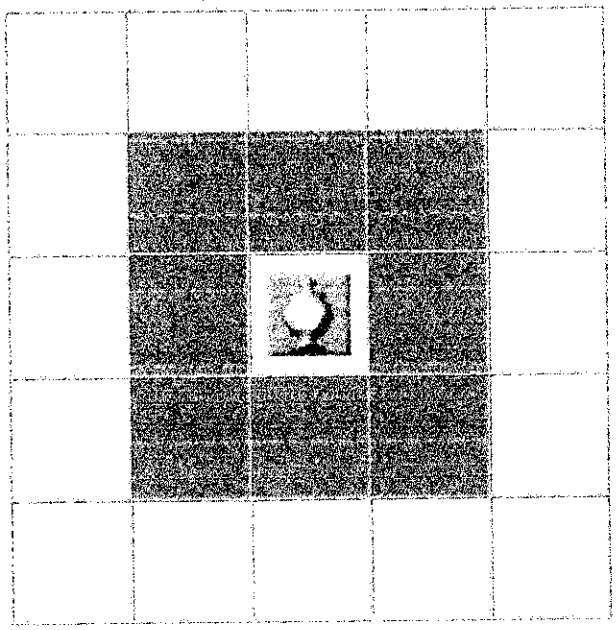
1. บอร์ด (นิยมใช้เป็น Array) เพื่อใช้เก็บค่าตัวหมากรุก
2. โครงสร้างข้อมูลในการเก็บตำแหน่งที่สามารถเดินได้ สำหรับกรณีเรียกใช้ฟังก์ชัน GenerateLegalMoves และ GenerateLegalCapture ซึ่งนิยมใช้เป็น Array อาจจะต้องมีจำนวนมากเพียงพอด้วยเพราะในการค้นลึกการเติบโตจะเป็นแบบ Exponential
3. ฟังก์ชันในการสังเคราะห์ตาเดิน GenerateLegalMoves และ GenerateLegalCapture ซึ่งจะใช้ควบคู่กับบอร์ดเพื่อดูว่าตัวอะไรอยู่ในกระดานและตัวนั้นสามารถเดินไปไหนได้บ้างหรือกินอะไรได้บ้าง ซึ่งจะต้องสังเคราะห์ตามกฎการเล่นหมากรุก เช่น เบี้ยจะเดินถอยหลังไม่ได้ หรือเบี้ยจะกินตรงไม่ได้ และกินกันเองก็ไม่ได้ ส่วนนี้ต้องตรวจดูให้ดี
4. ฟังก์ชันในการตรวจดูว่าขุนถูกรุกอยู่หรือไม่ Incheck ซึ่งหากว่าขุนถูกรุกเราจะต้องเดินขุนอย่างเดียวเท่านั้นต้องห้ามให้เดินตัวอื่น
5. ฟังก์ชันในการ Makemove และ Unmakemove เป็นฟังก์ชันเพื่อใช้ในการทดลองเดินดู โดยจะเดินจากตาเดินที่สังเคราะห์ได้เพื่อดูว่าเมื่อเดินแล้วจะเกิดผลอย่างไร
6. ฟังก์ชันในการค้นหา ในการที่เราทดลองเดินไปนั้นก็เพื่อเป็นการค้นหาซึ่งถือว่าเป็นหัวใจหลักส่วนหนึ่ง เพราะการค้นหานั้นยิ่งค้นลึกเท่าไรก็จะยิ่งเสียเวลามากขึ้นเป็นทวีคูณ ดังนั้นจึงต้องมีวิธีการในการตัดเส้นทางที่ไม่จำเป็นในการค้นออกไป นั้นก็คือ Alpha-Beta Search
7. ฟังก์ชันค้นหาทางเลือกในการกินกัน หรือที่เรียกว่า Quiescence Search เมื่อค้นลึกไปถึงบัพสุดท้ายแล้วก่อนจะมีการประเมินค่านั้นจะต้องพิจารณาดูก่อนว่าการกินกันหรือไม่กินกันสิ่งไหนจะคุ้มที่สุด ซึ่งฟังก์ชันนี้จะเป็นฟังก์ชันก่อนที่จะเรียกใช้ Evaluation
8. ฟังก์ชัน Evaluation หรือฟังก์ชันประเมินค่า เป็นฟังก์ชันที่จะให้ค่ากลับมาว่าผู้เล่นในฝ่ายที่เดินนั้นจะได้ค่าเท่าไรเมื่อเดินไปตำแหน่งนั้น ๆ

และนี่ก็คือโครงสร้างและฟังก์ชันที่จำเป็นสำหรับโปรแกรมหมากรุกคอมพิวเตอร์แบบเบื้องต้น

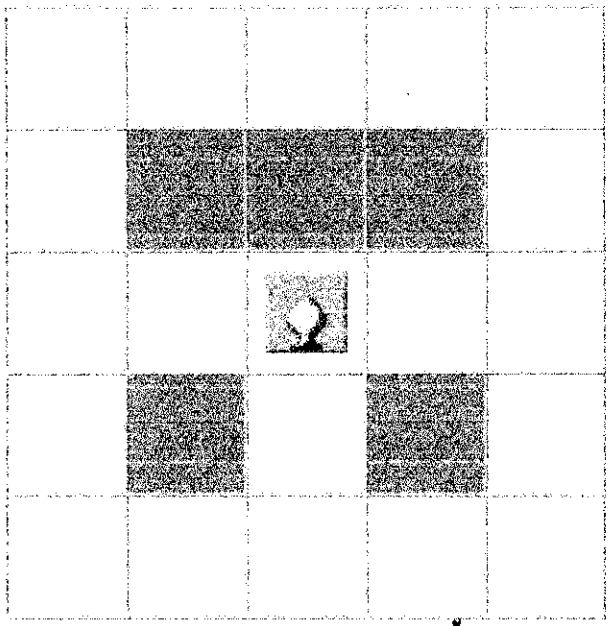


ภาพแสดงการทำงานของโปรแกรม

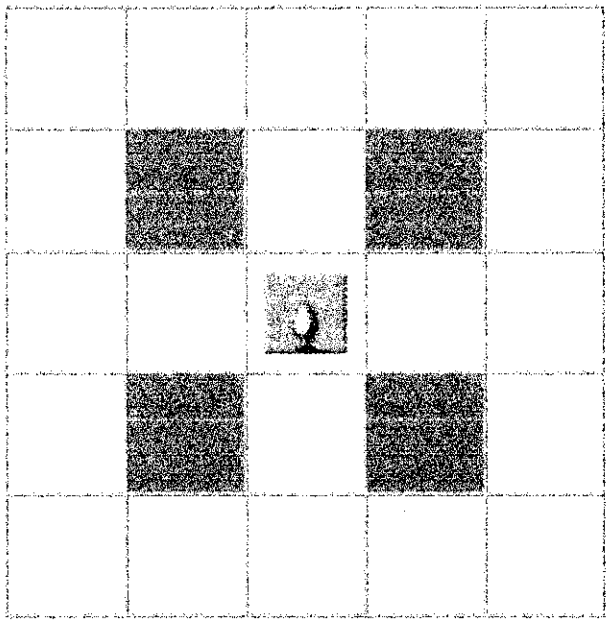
ขุน - แม่ทัพไทย เป็นนักรบที่มีความสามารถ มักเป็นผู้นำในการรบเปรียบเสมือน "ขุน" ในหมากรุกไทย ซึ่งต้องบงการการรบทั้งกระดาน ลักษณะการเดิน และการกินคู่ต่อสู้ของ ขุน



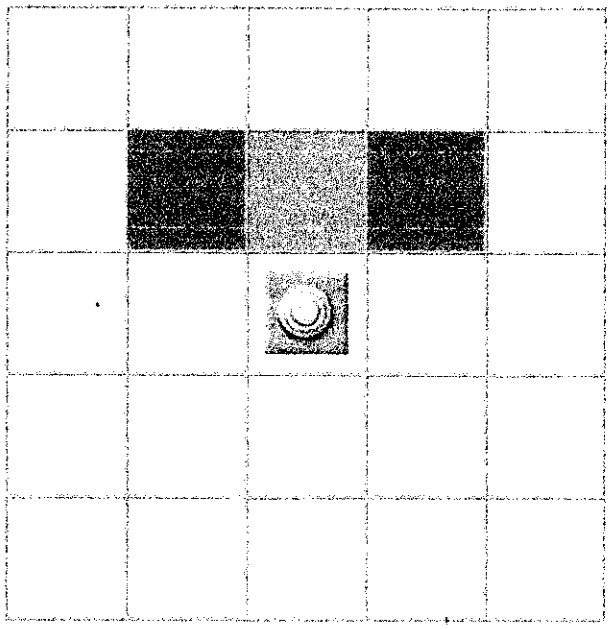
โคน - ทหารเอกขนาบข้างซ้าย และขวาของขุน มีความสำคัญในการเล่นหมากรุกเป็นอย่างมาก ดังคำกล่าวที่ว่า "ถ้ามีโคนคู่แล้ว เปรียบเสมือนหนึ่งมีขุนถึง 3 ขุนด้วยกัน" ลักษณะการเดิน และการกินคู่ต่อสู้ของ โคน



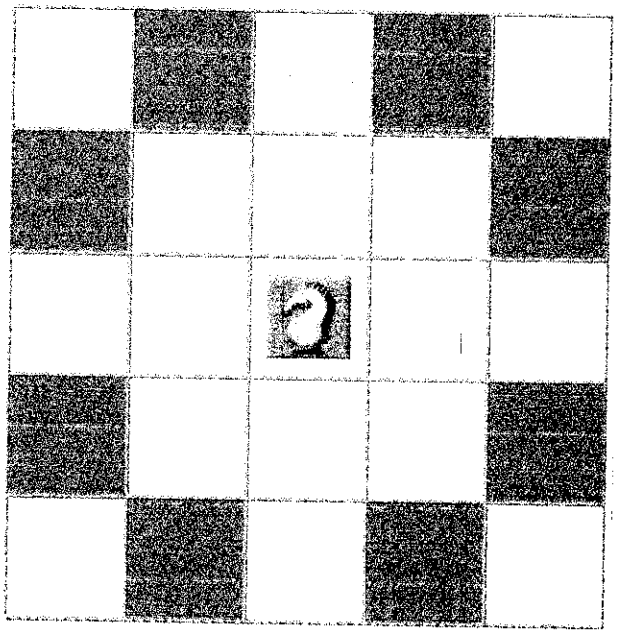
เม็ด - คนไทยถือว่า "ขุน" เป็นจอมทัพ / เทพยดา ดังนั้นจึงต้องมี องครักษ์พิทักษ์ขุน ซึ่งก็คือ "เม็ด" ใน หมากรุกไทยนั่นเอง ลักษณะการเดิน และการกินคู่ต่อสู้ของ เม็ด



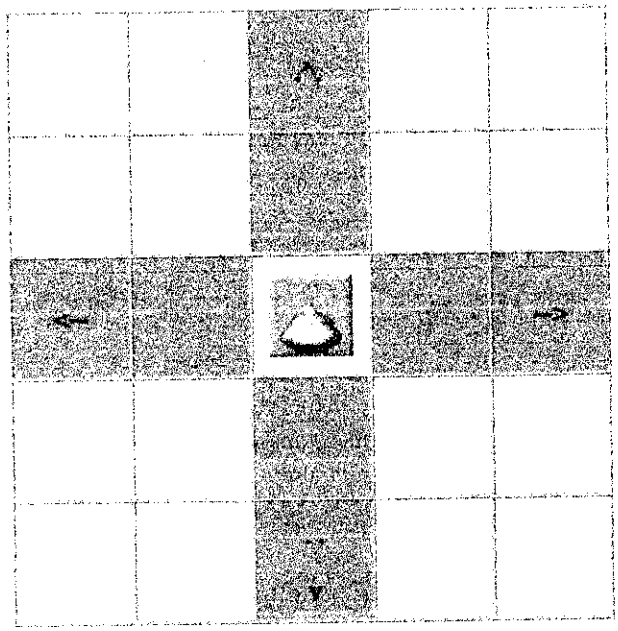
เบี้ย - เป็นด่านหน้าที่มีจำนวนมากที่สุด มีหน้าที่พิทักษ์รักษาจอมทัพ หากพลทหารใดสามารถเดินเข้าไปถึงด่านหน้าของข้าศึกได้อย่างปลอดภัย จะได้เลื่อนตำแหน่งขึ้นเป็นเม็ด นั่นก็คือ "เบี้ยหงาย" นั่นเอง ลักษณะการเดิน และการกินคู่ต่อสู้ของ เบี้ย (เดินตรง กินเฉียง)



ม้า เป็นกำลังรบที่สำคัญเปรียบได้กับอัศวินม้าในกองทัพ เพราะสามารถพลิกเกมที่เพลี่ยงพล้ำให้กลับมาเป็นฝ่ายคุม สถานการณ์ได้ด้วยการ "รุกฆาต" มีคำกล่าวที่ว่า " คนที่จะเดินม้าให้เก่งแล้ว ต้องสามารถเดินม้า ให้ครบทั้ง 64 ตาของกระดานโดยไม่ซ้ำกันเลยแม้แต่ครั้งเดียว" ลักษณะการเดิน และการกินคู่ต่อสู้ของ ม้า



เรือ - เป็นหมากที่สำคัญมาก เปรียบเสมือนกองทัพเรือ หรือกองทัพอากาศที่มีอาวุธยาวยิงไกลได้ สามารถทำลายศัตรูได้มาก หากเสียเรือ ก็จะเป็นการเสียอาวุธที่สำคัญไป ลักษณะการเดินและการกินคู่ต่อสู้ของ เรือ



คำศัพท์และความหมาย

Abstraction	นามธรรม
Algorithm	ขั้นตอนวิธี
Alphabet	ตัวอักขระ
Argument	อาร์กิวเมนท์,ส่วนในวงเล็บ,ข้อโต้แย้ง
Arthificial intelligence	ปัญญาประดิษฐ์
Artificial Neural Networks	รูปแบบคอมพิวเตอร์ที่พยายามเลียนแบบเครือข่ายใยประสาททางชีววิทยา การประมวลแบบ Neural
Axon	แกนประสาท
backtracking	เทคนิคที่ใช้พิจารณาเรื่องการย้อนกลับไปเริ่มต้น
Backus-Naur Form (BNF)	รูปแบบบรรทัดฐานแบกคัส
Behavior	ทำนายพฤติกรรม
black box	กล่องดำ
bottom-up	ล่างขึ้นบน
Class	สมาชิกของกลุ่ม
Closed Loop of Neurons	วงรอบของเซลล์ประสาท
Cognitive Processor	ตัวประมวลผลการรับรู้
compile	แปลโปรแกรม
Condition	เงื่อนไข
conditional part	ส่วนของเงื่อนไข
Connected graph	กราฟที่มี NODE 2 ตัวมี PATH เชื่อมกันไว้
Connectioness	การติดต่อกัน
Consistency	ประโยคที่มีค่าเป็นจริง หรือ เท็จ
Cut	ทำหน้าที่ป้องกันมิให้มีการ backtracking
data abstraction	ข้อมูลที่เป็นความจริง
Decision Support System	ระบบสนับสนุนการตัดสินใจ

Declarative Programming	โปรแกรมการอธิบาย
Deductive apparatus	กลไกการสืบสมมติฐาน
Dendrite	เส้นใยประสาท
Derivations	การสืบสมุฏฐาน
Different engine	เครื่องผลต่าง
Directed graph	กราฟที่มีการให้ทิศทางกับ ARC
Distributed system	ระบบแจกแจง
Efficiency	ประสิทธิภาพ
Enrichment	การเพิ่มความสมบูรณ์
experiential knowledge	ความรู้ประสบการณ์
Expert system	ระบบผู้เชี่ยวชาญ
Expressions	นิพจน์
Expressions	การแสดงออก
Expressiveness	การแสดงออก
Fail	เมื่อโปรลอกพบ fail ในอนุประโยคใด ก็จะทำให้อนุประโยคนั้นเป็นเท็จ
Formal	แบบแผน
Formal language	ภาษาแบบแผน
Formal system	ระบบแบบแผน
Free variable	ตัวแปรอิสระ
Fuzzy Logic	ตรรกศาสตร์คลุมเครือ
Fuzzy Sets	กลุ่มคลุมเครือ
Generating	การสร้าง
Goal	เป้าประสงค์
Graph theory	ทฤษฎีกราฟ

Handbook of Artificial Intelligence	หนังสือคู่มือปัญญาประดิษฐ์
Head	ส่วนหัว
Heirarchical problem decomposition	คือ การแตกปัญหาย่อยตามลำดับขั้น
Imperative language	ภาษาเชิงคำสั่ง
Implement pattern	จัดเก็บรูปแบบ
Inconsistent	ประโยคที่มีค่าเป็นความเป็นเท็จเสมอ
Inference Engine	ส่วนของประมวลผลสรุป
Inference systems	ระบบอนุมาน
Inherit	ถ่ายทอด
Intelligent Systems	ระบบความชาญฉลาด
Interpretation	การตีความ
Interpreter	ตัวแปลภาษา
Knowledge Domain	ขอบเขตความรู้
Knowledge representation	การแสดงความรู้
Knowledge-base container	ส่วนบรรจุข้อมูลพื้นฐาน
knowledge-base expert system	ระบบผู้เชี่ยวชาญความรู้พื้นฐาน
knowledge-base system	ระบบความรู้พื้นฐาน
Knowlegde representation language	ภาษาแสดงความรู้
Languages and environments for AI	ภาษาและสภาพแวดล้อมสำหรับปัญญาประดิษฐ์
Laws of Thought	กฎแห่งความคิด
Lemma	ข้อเสนอแทรก
Link	เชื่อมโยง
List	จำนวนเทอม
Logic	ตรรก

Loop	วงวน
Machine learning	การเรียนรู้ของเครื่องคอมพิวเตอร์
Medium	ตัวกลาง
Membership Value	มีค่าความเป็นสมาชิก
Meta language	อภิภาษา
Modelling human performance	การสร้างรูปแบบการทำงานของมนุษย์
Modelling the real world	การสร้างแบบจำลองของจริง
Modularity	สภาพเป็นส่วนจำเพาะ
Module	ส่วนจำเพาะ
Natural language understanding and semantic modeling	การเข้าใจภาษาธรรมชาติและการสร้างรูปแบบความหมาย
Neuron Network	ระบบเครือข่ายของเซลล์ประสาท
Node	จุดต่อ
Object-oriented	เชิงวัตถุ
Object-Oriented programming	โปรแกรมเชิงวัตถุ
Objects	วัตถุ
Optimal solution	การค้นหาข้อสรุปที่ดีที่สุด
Parallel Computation	การคำนวณแบบขนาน
Parallel Computing	การคำนวณแบบขนาน
Path	ทางเดินของกราฟจากจุดกำหนดไปยังจุดปลายทาง
pattern part	ส่วนแบบแผน
Planning and robotics	การวางแผนและหุ่นยนต์
Predicate	คำที่แสดงไว้ในรูปแบบ
Premises	ข้อกำหนด

Probability	ความน่าจะเป็น
Problem solving as search	การค้นหาวิธีการแก้ปัญหา
Problem state	ขั้นตอนของปัญหา
Proofs and theorems	การพิสูจน์และทฤษฎี
Proposition	ความเป็นจริงและเท็จ
Quantify	กำหนดปริมาณจำนวน
Rapid Prototyping	การเพิ่มระดับความรู้อย่างรวดเร็ว
recognize-act cycle	วงจรรู้จักการกระทำ
Recursion	เรียกซ้ำ
Recursive function	ฟังก์ชั่นเรียกตัวเอง
re-inventing the wheel	กลับมาประดิษฐ์อีกครั้ง
Relations	ความสัมพันธ์
Representation language	ภาษาการแสดง
Representation scheme	รูปแบบการแสดง
Robot	หุ่นยนต์
Rooted graph	กราฟที่มี NODE หนึ่งทำหน้าที่เป็น ROOT
run	ดำเนินงาน
Scope	ขอบเขต
Search	การลำดับความรู้,การสืบค้น
select-execute cycle	วงจรเลือกการปฏิบัติ
Semantics	ความหมาย
situation-action cycle	วงจรสถานการณ์การกระทำ
situation-response cycle	วงจรสถานการณ์ตอบสนอง
Software engineering	วิศวกรรมซอฟต์แวร์
Solution	การค้นหาข้อสรุป

Solution path	
Space	การกำหนดขอบเขต
Specifying artifacts	การกำหนดวัตถุจำลอง
State	สถานะ
Structure	รูปแบบ
Success of expert systems	ความสำเร็จ ของระบบผู้เชี่ยวชาญ
super classes	
Symbol pattern	สัญลักษณ์รูปแบบ
Symbolic computing	การคำนวณเชิงสัญลักษณ์
Synapse	จุดรวมเซลล์ประสาท
Syntactic level	ระดับโครงสร้าง
Systax	วากยสัมพันธ์
Tail	ส่วนหาง
Tautology	ประโยคที่มีค่าเป็นความจริงเสมอ
Theories	ทฤษฎี
Tool	เครื่องมือ
Top level	ขั้นสูงสุด
top-down	บนลงล่าง
Tree	กราฟซึ่งมี ARC เพียงเส้นเดียวเชื่อมระหว่าง NODE 2 ตัว
Well-defined language	ภาษาที่ถูกกำหนดไว้เป็นอย่างดีแล้ว