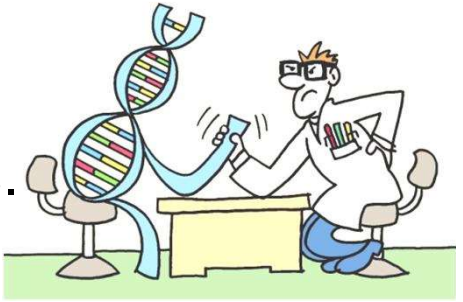
6.1 Game theory

- Game theory
  - –Developed by von Neumann and Morgensten.
  - –Widely studied by economists, mathematicians, financiers.
  - –The action of one player (agent) can significantly affect the utilities of the others.
    - Cooperative or competitive.
    - Deal with the environments with multiple agents.



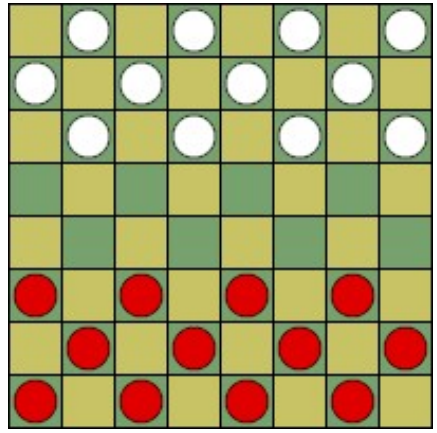
- Most games studied in AI are
  - – Deterministic
  - – Turn-taking
  - – Two-player
  - – Zero-sum
  - – Perfect information

This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the utility values at the end of game are always equal or opposite.

Types of Games

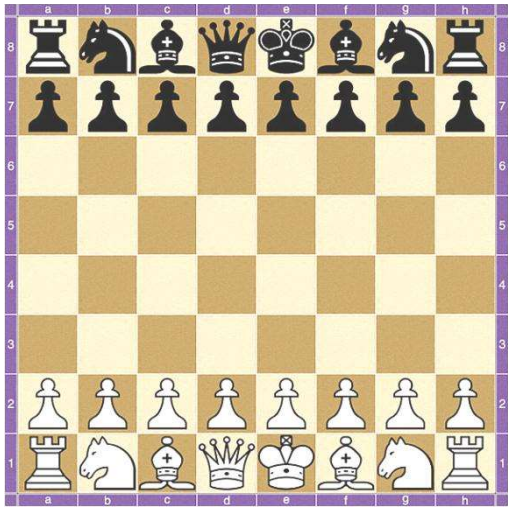
	deterministic	chance
perfect information	Chess西洋棋, Checkers西洋跳棋, Go圍棋, Othello黑白棋	Backgammon 西洋雙陸棋, Monopoly大富翁
imperfect information	Battleship海戰棋, Kriegspiel盲西洋棋, Stratego陸軍棋	Bridge橋牌, Poker撲克, Scrabble拼字遊戲

- Games are one of the first tasks undertaken(從事) in AI.
  - The abstract nature of (nonphysical) games makes them an appealing subject in AI.
- Computers have surpassed(大勝) humans in *Checkers* and *Othello*, and have defeated human champions in *Chess* and *Backgammon*.
- However, in *Go*, computers still perform at the amateur(業餘的) level before 2016.

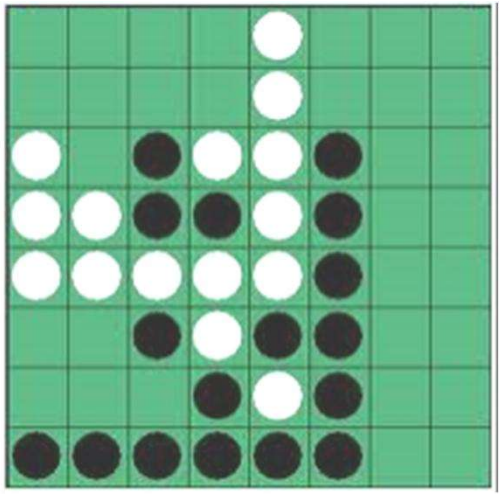


Example Computer Games

- Chess – Deep Blue (World Champion 1997)
  1957: Herbert Simon: "within 10 years a computer will beat the world chess champion".
  1997: Deep Blue beats Kasparov.



- Checkers – Chinook (World Champion 1994)



- Othello – Logistello
  - Beginning, middle, and ending strategy.
  - Generally accepted that humans are no match for computers at Othello.

- Backgammon – TD-Gammon (Top Three)

- Bridge (Bridge Barron 1997, GIB 2000)
  - Imperfect information, multiplayer with two teams of two.
- Go – Goemate and Go4++ (Weak Amateur)

Go:

- The branching factor starts at 361 (19×19), which is too huge for regular search methods.
- 2010年：師大資工所黃士傑同學開發的「Erica」於日本舉辦的2010年 Computer Olympiad榮獲19路電腦圍棋金牌。
- 2016年：世紀人機大賽 Google Deepmind's AlphaGo擊敗棋王李世乭，黃士傑是首席設計師。

186 Game 1	未打劫
211 Game 2	未打劫
176 Game 3	123手造劫，151手提劫
180 Game 4	未打劫 李78手神之一手 黑97入門級錯誤
280 Game 5	194手造劫，201手造劫，230手提劫

棋盤表示法：以象棋為例

一、9*10的二維陣列

- 最直觀的棋盤表示法，速度最慢。
- 走法生成例子:

馬在Y =3, X = 4，可走位置為

Y = 3±2 or 3±1

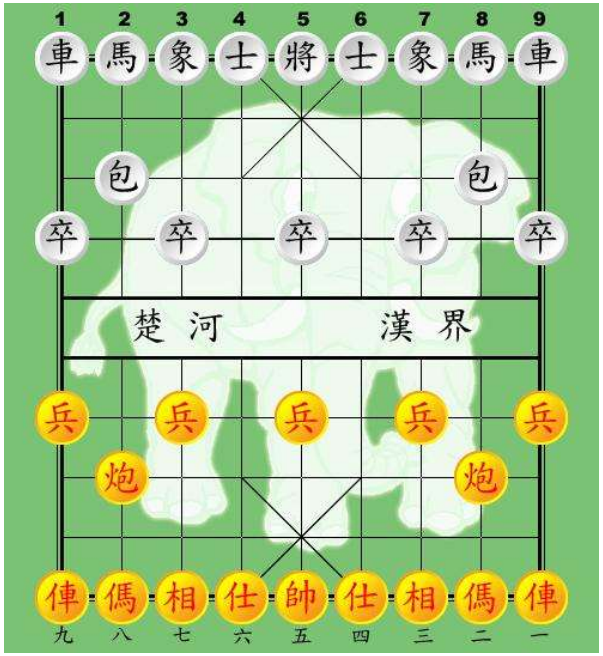
X = 4±2 or 4±1

需將拐馬腳及不在棋盤範圍的座標逐個排除。

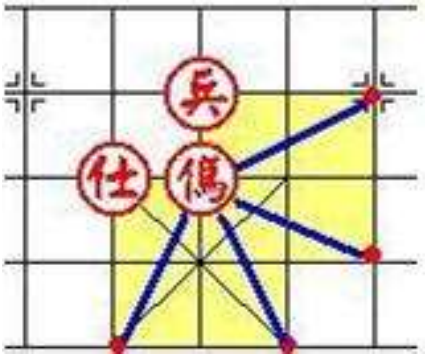
00	01	02	03	04	05	06	07	08
10	11	12	13	14	15	16	17	18
20	21	22	23	24	25	26	27	28
30	31	32	33	34	35	36	37	38
40	41	42	43	44	45	46	47	48
50	51	52	53	54	55	56	57	58
60	61	62	63	64	65	66	67	68
70	71	72	73	74	75	76	77	78
80	81	82	83	84	85	86	87	88
90	91	92	93	94	95	96	97	98

Y

X




```
BYTE Board[10][9]=
{   {2,3,6,5,1,5,6,3,2},
    {0,0,0,0,0,0,0,0,0},
    {0,4,0,0,0,0,0,4,0},
    {7,0,7,0,7,0,7,0,7},
    {0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0},
    {14,0,14,0,14,0,14,0,14},
    {0,11,0,0,0,0,0,11,0},
    {0,0,0,0,0,0,0,0,0},
    {9,10,13,12,8,12,13,10,9}  };
```



二、90的一維陣列

- **BYTE Board[90]**
- 速度慢
- 走法生成例子:

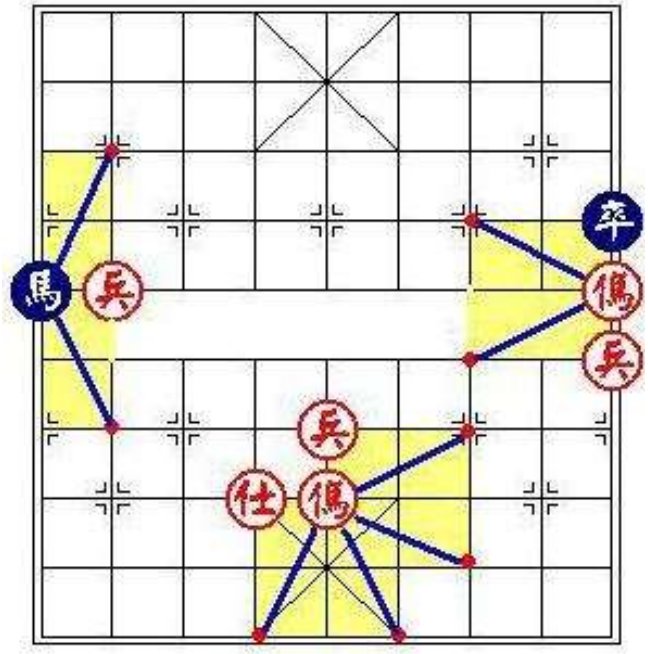
馬在39，可走位置

M = 39 + INC

INC = {-19,-17,-11,-7,+7,+11,+17,+19}

然後將拐馬腳及不在棋盤範圍的座標逐個排除

00	01	02	03	04	05	06	07	08
09	10	11	12	13	14	15	16	17
18	19	20	21	22	23	24	25	26
27	28	29	30	31	32	33	34	35
36	37	38	39	40	41	42	43	44
45	46	47	48	49	50	51	52	53
54	55	56	57	58	59	60	61	62
63	64	65	66	67	68	69	70	71
72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89



三、13*14的一維陣列

00	01	02	03	04	05	06	07	08	09	10	11	12
13	14	15	16	17	18	19	20	21	22	23	24	25
26	27	28	29	30	31	32	33	34	35	36	37	38
39	40	41	42	43	44	45	46	47	48	49	50	51
52	53	54	55	56	57	58	59	60	61	62	63	64
65	66	67	68	69	70	71	72	73	74	75	76	77
78	79	80	81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100	101	102	103
104	105	106	107	108	109	110	111	112	113	114	115	116
117	118	119	120	121	122	123	124	125	126	127	128	129
130	131	132	133	134	135	136	137	138	139	140	141	142
143	144	145	146	147	148	149	150	151	152	153	154	155
156	157	158	159	160	161	162	163	164	165	166	167	168
169	170	171	172	173	174	175	176	177	178	179	180	181

- **BYTE Board[182]**

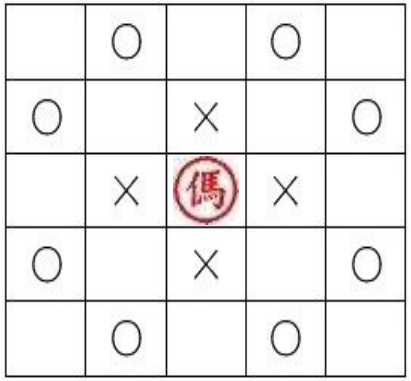
  外圍安排兩層牆壁
- 速度快
- 走法生成例子：

  馬在83，可走位置：

  M = 83 + INC

  INC={-27,-25,-15,-11,+11,+15,+25,+27}
- 可迅速判斷是否在棋盤範圍內：

  Borad[M] != wall



四、16*16的一維陣列

BYTE Board[256]

速度快

走法生成例子:

馬在0xB4位置

可走位置：

M = 0xB4 + INC

INC={-0x21,-0x1F,
-0x12,-0x0E,
+0x0E,+0x12,
+0x1F,+0x21}

可迅速判斷是否在棋盤範圍內：

Borad[M] != wall

求行號及列號很快：

X=M&0x0F;

Y=M>>4;

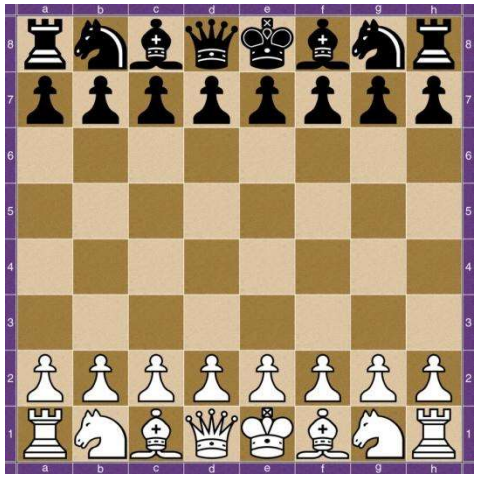
x00	x01	x02	x03	x04	x05	x06	x07	x08	x09	x0A	x0B	x0C	x0D	x0E	x0F
x10	x11	x12	x13	x14	x15	x16	x17	x18	x09	x1A	x1B	x1C	x1D	x1E	x1F
x20	x21	x22	x23	x24	x25	x26	x27	x28	x29	x2A	x2B	x2C	x2D	x2E	x2F
x30	x31	x32	x33	x34	x35	x36	x37	x38	x39	x3A	x3B	x3C	x3D	x3E	x3F
x40	x41	x42	x43	x44	x45	x46	x47	x48	x49	x4A	x4B	x4C	x4D	x4E	x4F
x50	x51	x52	x53	x54	x55	x56	x57	x58	x59	x5A	x5B	x5C	x5D	x5E	x5F
x60	x61	x62	x63	x64	x65	x66	x67	x68	x69	x6A	x6B	x6C	x6D	x6E	x6F
x70	x71	x72	x73	x74	x75	x76	x77	x78	x79	x7A	x7B	x7C	x7D	x7E	x7F
x80	x81	x82	x83	x84	x85	x86	x87	x88	x89	x8A	x8B	x8C	x8D	x8E	x8F
x90	x91	x92	x93	x94	x95	x96	x97	x98	x99	x9A	x9B	x9C	x9D	x9E	x9F
xA0	xA1	xA2	xA3	xA4	xA5	xA6	xA7	xA8	xA9	xAA	xAB	xAC	xAD	xAE	xAF
B xB0	xB1	xB2	xB3	xB4	xB5	xB6	xB7	xB8	xB9	xBA	xBB	xBC	xBD	xBE	xBF
xC0	xC1	xC2	xC3	xC4	xC5	xC6	xC7	xC8	xC9	xCA	xCB	xCC	xCD	xCE	xCF
xD0	xD1	xD2	xD3	xD4	xD5	xD6	xD7	xD8	xD9	xDA	xDB	xDC	xDD	xDE	xDF
xE0	xE1	xE2	xE3	xE4	xE5	xE6	xE7	xE8	xE9	xEA	xEB	xEC	xED	xEE	xEF
xF0	xF1	xF2	xF3	xF4	xF5	xF6	xF7	xF8	xF9	xFA	xFB	xFC	xFD	xFE	xFF

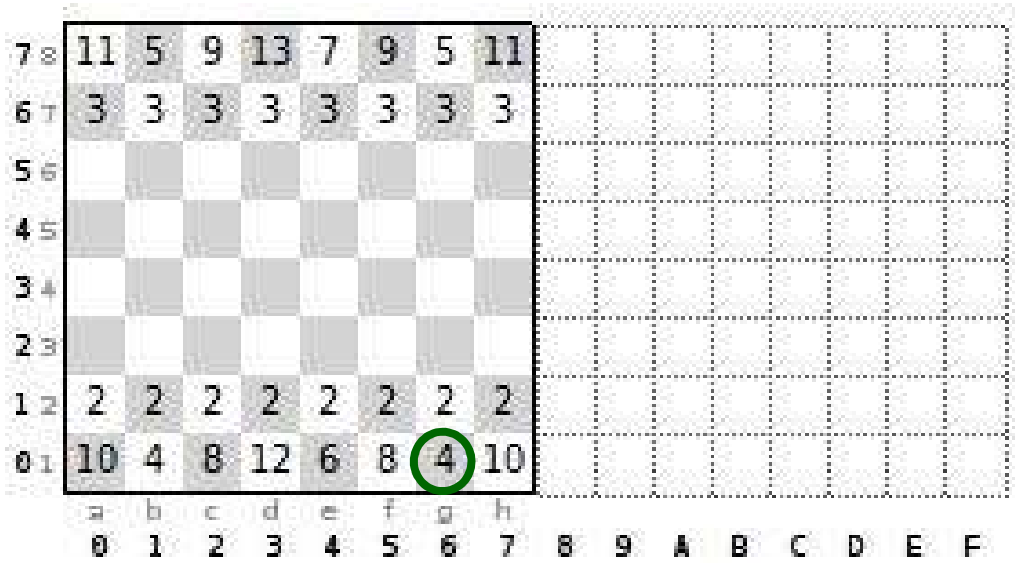
Y

X

4

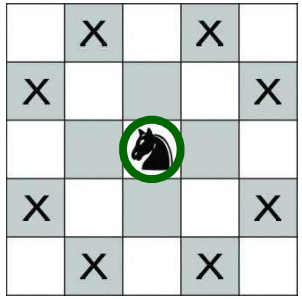
西洋棋0x88棋盤表示法





112	113	114	115	116	117	118	119
96	97	98	99	100	101	102	103
80	81	82	83	84	85	86	87
64	65	66	67	68	69	70	71
48	49	50	51	52	53	54	55
32	33	34	35	36	37	38	39
16	17	18	19	20	21	22	23
0	1	2	3	4	5	6	7

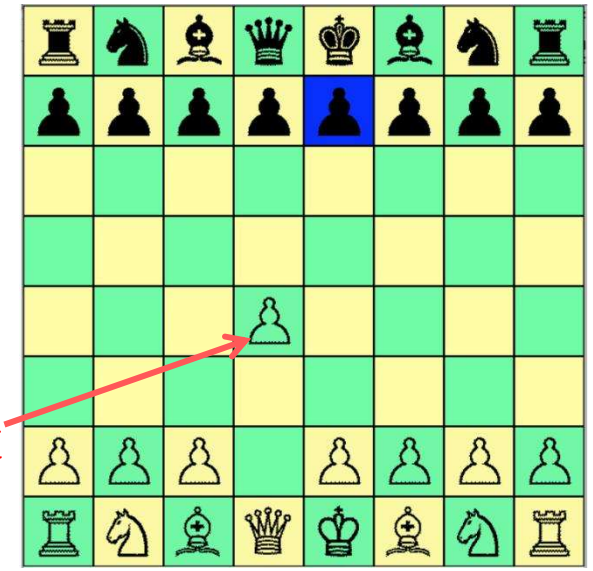
x88Board[0~119] =
[10, 4, 8, 12, 6, 8, 4, 10, 0, 0, 0, 0, 0, 0, 0, 0,
2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, ..., 0, 0,
3, 3, 3, 3, 3, 3, 3, 3, 0, 0, 0, 0, 0, 0, 0, 0,
11, 5, 9, 13, 7, 9, 5, 11, 0, 0, 0, 0, 0, 0, 0, 0]



- 速度更快，走法生成例子：
  馬在6，可走位置：M = 6 + INC　//M用8 bits表示
  INC={-33,-31,-18,-14,+14,+18,+31,+33}
- 可迅速判斷是否在棋盤範圍內：
  if (!(M & 0x88)) then 可走到M

五、Bitboards位元棋盤

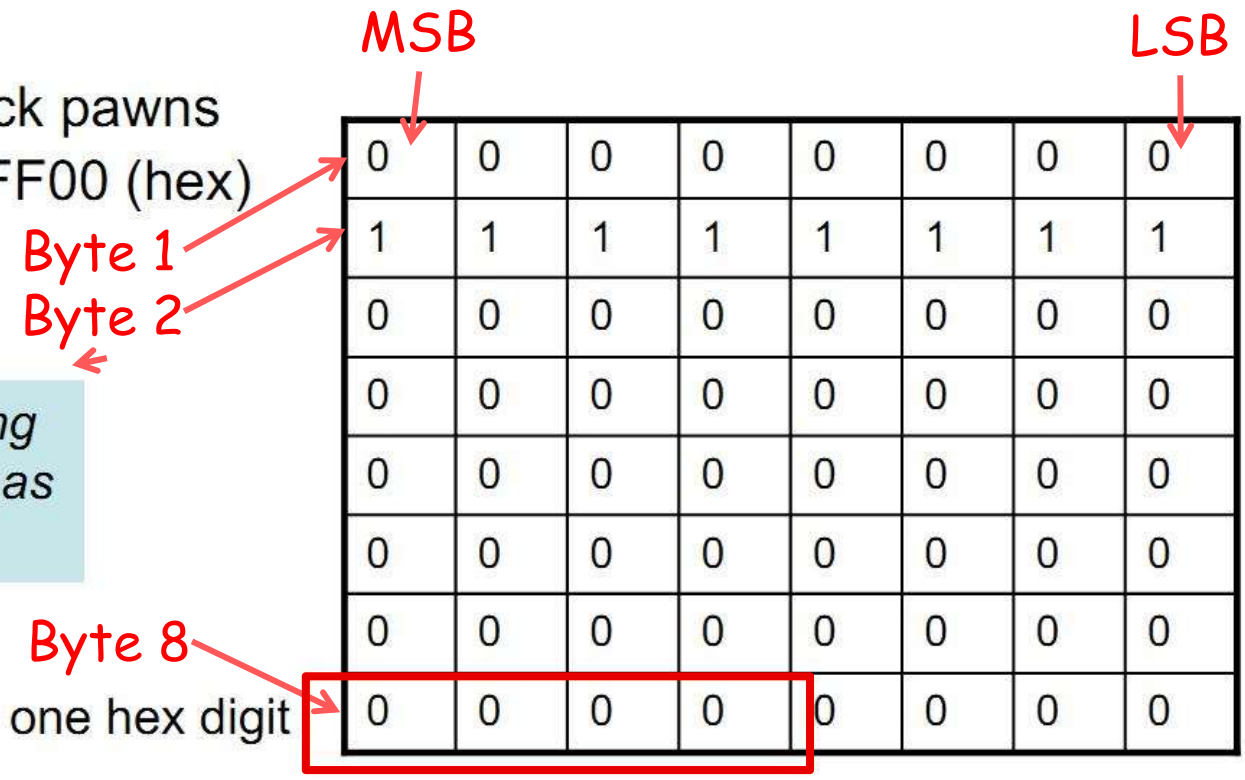
- 用12個8*8的64位數表示12種棋子位置，利用速度更快的AND、OR等動作處理西洋棋盤面。
- 非常快！(1970年蘇聯Kaissa及美國Chess4.5已採用)



64 bits

- integers: typically 32 bits (4 bytes)
- use 64 bit integers: C's (unsigned) long long, uint64_t
- **Example**:
  - initial position: black pawns
  - 0x000000000000FF00 (hex)

use any reasonable mapping you like: of course, as long as you are consistent



- Encoding a 1 step move (***by black pawns***):
  - 0x000000000824C300 –> 0x0000000824C30000
  - left shift by 8 positions (=1 row)
  - i.e. 0x000000000824C300 << 8

byte8	byte7	byte6	byte5	byte4	byte3	byte2	byte1

hex15	hex14	hex13	hex12	hex11	hex10	hex9	hex8	hex7	hex6	hex5	hex4	hex3	hex2	hex1	hex0

0x000000082400C300 <<8 0x0000000824C30000

0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1
0	0	1	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1
0	0	1	0	0	1	0	0
0	0	0	0	1	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

- Integer representation
  - signed (2's complement)
  - 32 bit version

-1	= 0xFFFFFFFF
-2147483648	= 0x80000000
2147483647	= 0x7FFFFFFF
1	= 0x00000001
0	= 0x00000000

- **this usually means we need to be careful with right shifts**
  - because of sign bit preservation
  - example:
    - 0x80000000 >> 1
    - =
    - 0xC0000000

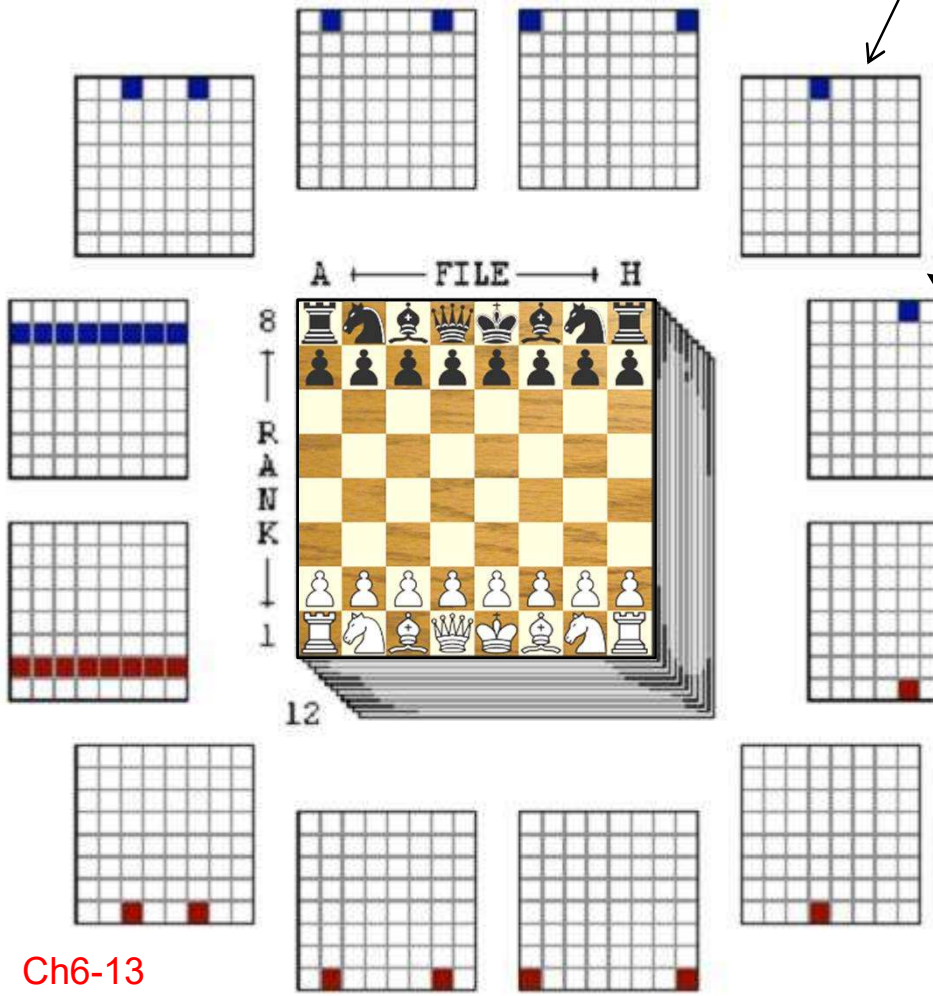
除2的意思

C 語言視這個常數為 signed integer

```
typedef unsigned __int64 U64;    // for the old microsoft compilers
typedef unsigned long long  U64; // supported by MSC 13.00+ and C99
#define C64(constantU64) constantU64##ULL
```

How many bitboards?

- one for each type of piece K, Q, R, B, N, P
- for each side
- = 12



Initially:

K ♚	0x0000000000000008
Q ♛	0x0000000000000010
R	0x0000000000000081
B	0x0000000000000024
N	0x0000000000000042
P	0x000000000000FF00
k	0x0800000000000000
q	0x1000000000000000
r	0x8100000000000000
b	0x2400000000000000
n	0x4200000000000000
p	0x00FF000000000000

有黑子 black = K | Q | R | B | N | P (*always*)

black = 0x000000000000FFFF

有白子 white = k | q | r | b | n | p (*always*)

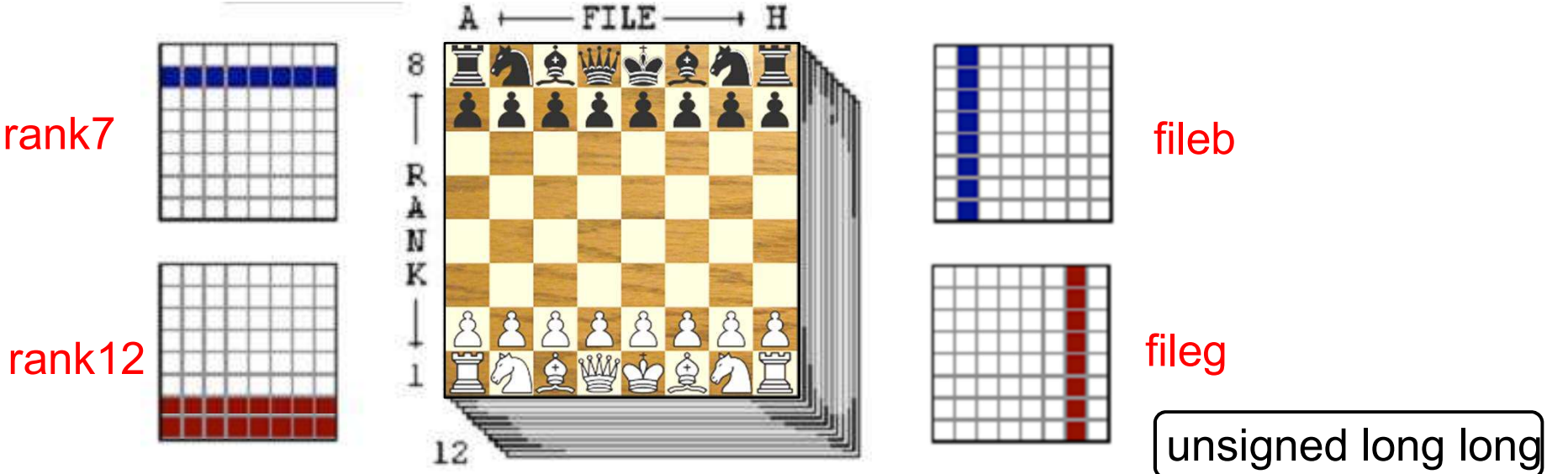
white = 0xFFFF000000000000

有子 occupied = white | black

空格 empty = ~ (white | black) (*always*)

empty = ~ 0xFFFF00000000FFFF

= 0x0000FFFFFFFF0000

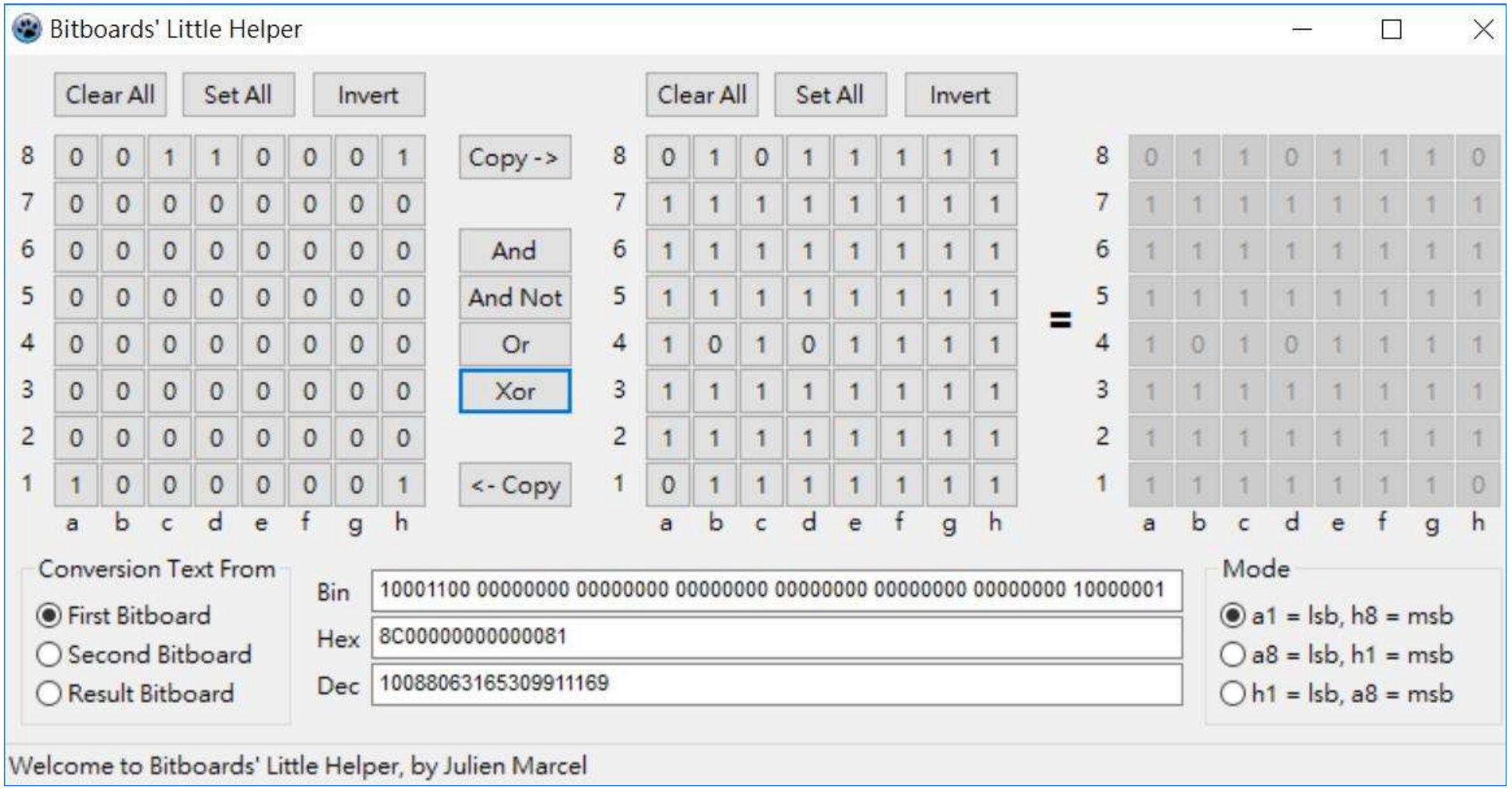


unsigned long long

- **Bitboard Masks** 這些均為常數 **Const U64 filea=0x8080808080808080ULL**

rank8 = 0x00000000000000FF	filea = 0x8080808080808080
rank7 = 0x000000000000FF00	fileb = 0x4040404040404040
rank6 = 0x0000000000FF0000	filec = 0x2020202020202020
rank5 = 0x00000000FF000000	filed = 0x1010101010101010
rank4 = 0x000000FF00000000	filee = 0x0808080808080808
rank3 = 0x0000FF0000000000	filef = 0x0404040404040404
rank2 = 0x00FF000000000000	fileg = 0x0202020202020202
rank1 = 0xFF00000000000000	fileh = 0x0101010101010101
rank12 = 0xFFFF000000000000	fileab = 0xC0C0C0C0C0C0C0C0
rank78 = 0x000000000000FFFF	filegh = 0x0303030303030303

Bitboards' Little Helper v1



http://www.chess2u.com/t2159-new-free-tool-bitboards-helper

- **(Legal) pawn moves**
  - forward one rank (*if not blocked*)
  - forward two ranks if not moved (*and not blocked*)
  - capture diagonally
  - *en passant* 吃過路兵
  - **promote**若兵到達敵方的底線，則可升變為其他棋子。

- **Bitboard implementation:** 黑兵的非吃子步

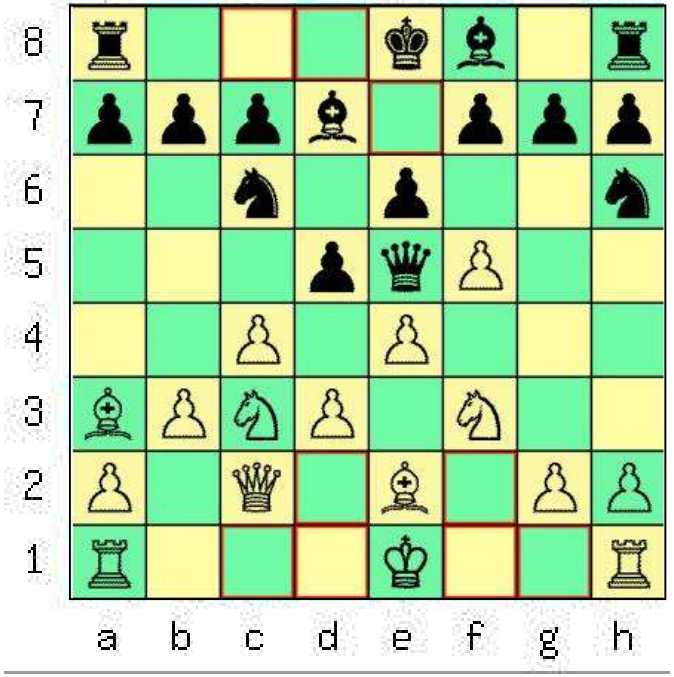
  - **forward one rank (*if not blocked*)**

    we can operate on all the pawns at a time (***bitparallel***) P=0x0000000010 08E700

    P << 8 =0x0000001008E70000

  - need to mask out occupied squares empty=0x76540BD7E3D60872

P_move1 = P << 8 & empty =0x0000001000C60000



  - **forward two ranks if not moved (*and not blocked*)**
  - define P=0x0000000010 08E700
    - rank7 = 0x000000000000FF00
  - P_move2 = ((P & rank7) << 8 & empty) << 8 & empty

P&rank7=0x0000000000 00E700

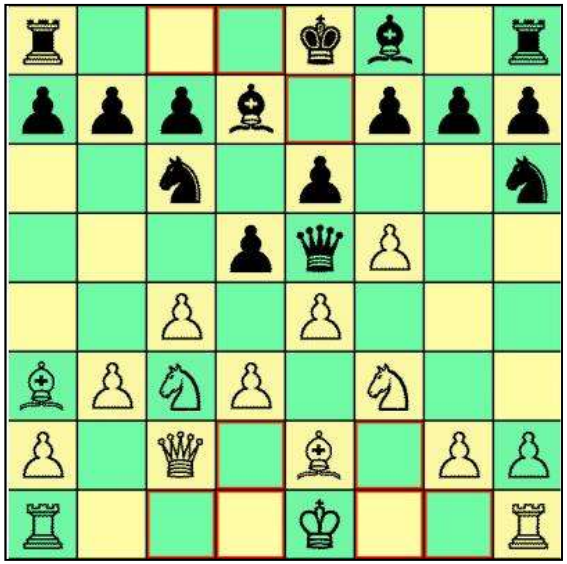
(P&rank7)<<8 = 0x0000000000E70000

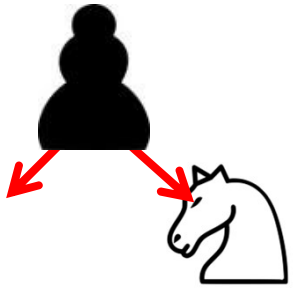
(P&rank7)<<8 &empty= 0x0000000000C60000

((P&rank7)<<8 &empty)<<8 &empty= 0x0000000 0C2000000

- **Pawn moves**
  - capture diagonally
  - define
    - filea = 0x8080808080808080
    - fileh = 0x0101010101010101
  - for black
  - attacks down and to the right
    (P & ~fileh) << 7
  - Attacks down and to the left
    (P & ~filea) << 9

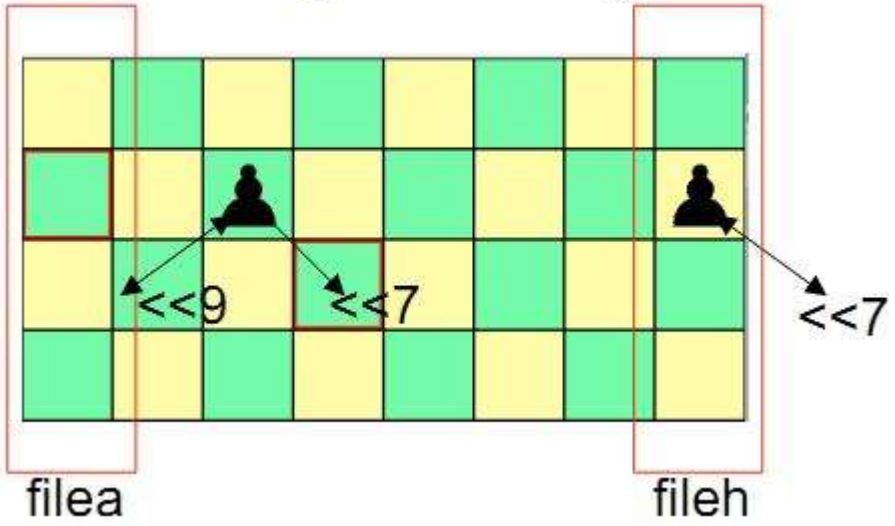
黑兵的吃子步





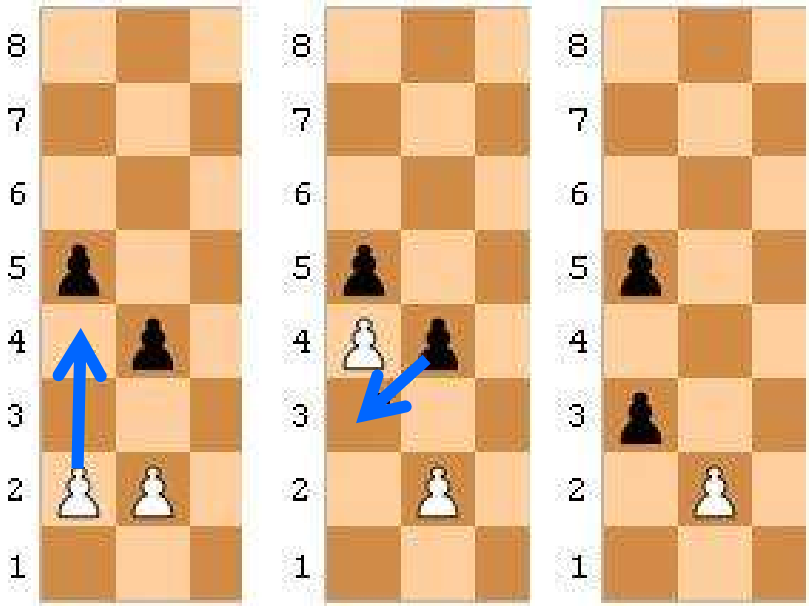
P_captures = (((P & ~fileh) << 7 | (P & ~filea) << 9))& white

- Masking at the edges



◎另一類吃子步：吃過路兵

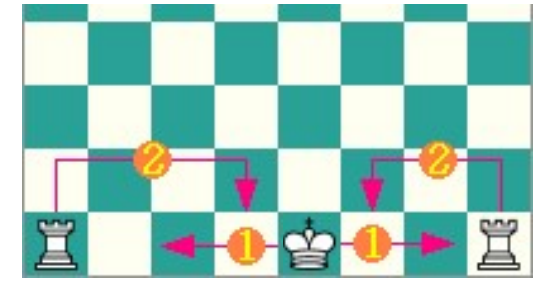
白兵走兩步，恰巧和黑兵並排在一起。黑兵此時可向前斜吃白兵！

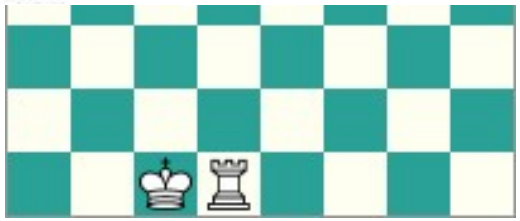


- **King moves**
  - one step in any direction to an empty square or opponent's square (*that is not attacked*)

    (excluding castling) 入堡:王車易位
  - can pre-compute king moves for each possible bitboard bb

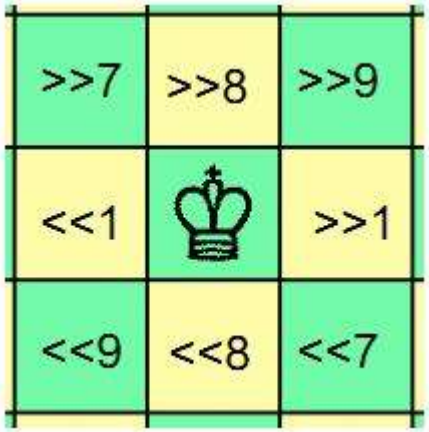
    (set_k_moves)
  - so k_moves function becomes an array lookup
  - 64 possible single bit bitboards





- **Bit Logic:**
  - let **bb** be the bitboard associated with a king (black or white)
  - **define**:

    k_moves(bb) = (bb & ~ rank8) >> 8 | (bb & ~ rank1) << 8 | (bb & ~ fileh) >> 1 | (bb & ~ filea) << 1 | (bb & ~ (rank8 | filea)) >> 7 | (bb & ~ (rank8 | fileh)) >> 9 | (bb & ~ (rank1 | fileh)) << 7 | (bb & ~ (rank1 | filea)) << 9

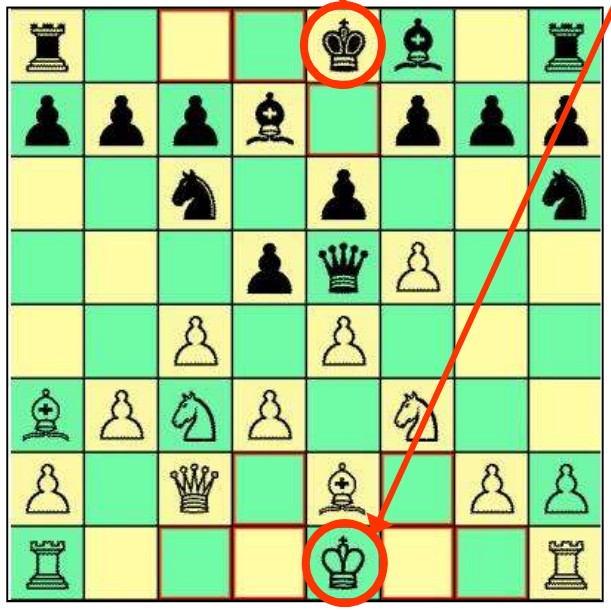


k_moves(bb) array

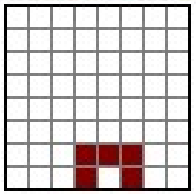
bb=0x0000000000000001～0x8000000000000000
64位元範圍過大，實作上有困難 (陣列過大： 2^{64})

0x0000000000000001 -> 0x0000000000000302
0x0000000000000002 -> 0x0000000000000705
0x0000000000000004 -> 0x0000000000000e0a
0x0000000000000008 -> 0x0000000000001c14
0x0000000000000010 -> 0x0000000000003828
0x0000000000000020 -> 0x0000000000007050
0x0000000000000040 -> 0x000000000000e0a0
0x0000000000000080 -> 0x000000000000c040
0x0000000000000100 -> 0x0000000000030203
0x0000000000000200 -> 0x0000000000070507
0x0000000000000400 -> 0x00000000000e0a0e
0x0000000000000800 -> 0x00000000001c141c
0x0000000000001000 -> 0x0000000000382838
0x0000000000002000 -> 0x0000000000705070
0x0000000000004000 -> 0x0000000000e0a0e0
0x0000000000008000 -> 0x0000000000c040c0
0x0000000000010000 -> 0x0000000003020300
0x0000000000020000 -> 0x0000000007050700
0x0000000000040000 -> 0x000000000e0a0e00
0x0000000000080000 -> 0x000000001c141c00
0x0000000000100000 -> 0x0000000038283800
0x0000000000200000 -> 0x0000000070507000
0x0000000000400000 -> 0x00000000e0a0e000
0x0000000000800000 -> 0x00000000c040c000

0x0000000100000000 -> 0x0000030203000000
0x0000000200000000 -> 0x0000070507000000
0x0000000400000000 -> 0x00000e0a0e000000
0x0000000800000000 -> 0x00001c141c000000
0x0000001000000000 -> 0x0000382838000000
0x0000002000000000 -> 0x0000705070000000
0x0000004000000000 -> 0x0000e0a0e0000000
0x0000008000000000 -> 0x0000c040c0000000
0x0000010000000000 -> 0x0003020300000000
0x0000020000000000 -> 0x0007050700000000
0x0000040000000000 -> 0x000e0a0e00000000
0x0000080000000000 -> 0x001c141c00000000
0x0000100000000000 -> 0x0038283800000000
0x0000200000000000 -> 0x0070507000000000
0x0000400000000000 -> 0x00e0a0e000000000
0x0000800000000000 -> 0x00c040c000000000



0x0000100000000000 -> 0x0038283800000000
0x0000200000000000 -> 0x0070507000000000
0x0000400000000000 -> 0x00e0a0e000000000
0x0000800000000000 -> 0x00c040c000000000
0x0001000000000000 -> 0x0302030000000000
0x0002000000000000 -> 0x0705070000000000
0x0004000000000000 -> 0x0e0a0e0000000000
0x0008000000000000 -> 0x1c141c0000000000
0x0010000000000000 -> 0x3828380000000000
0x0020000000000000 -> 0x7050700000000000
0x0040000000000000 -> 0xe0a0e00000000000
0x0080000000000000 -> 0xc040c00000000000
0x0100000000000000 -> 0x0203000000000000
0x0200000000000000 -> 0x0507000000000000
0x0400000000000000 -> 0x0a0e000000000000
0x0800000000000000 -> 0x141c000000000000
0x1000000000000000 -> 0x2838000000000000
0x2000000000000000 -> 0x5070000000000000
0x4000000000000000 -> 0xa0e0000000000000
0x8000000000000000 -> 0x40c0000000000000



白方國王的非吃子步

k_move=
k_moves(k)&empty

白方國王的吃子步

K_captures=
k_moves(k)&black

黑方國王的非吃子步

K_move=
k_moves(K)&empty

黑方國王的吃子步

K_captures=
k_moves(K)&white

Perfect hashing函數(bitscan_magic)

bb=0x0000000000000001～0x8000000000000000範圍過大，實作上有困難(陣列過大)。
下法由**Leiserson, Prokop, Randall**提出，先根據最低位的**1** 的位置，再轉換到 **0～63** 的**64**個不同的值(即做 **perfect hashing)**，則只需一個**64**個元素(各佔**64 bits)**的陣列 。

perfect_hashing(bb) = (bb * 0x07EDD5E59A4E28C2) >> 58

留下左邊6個位元!

Magic number!

0000 011*1 1110 1*101 1101 0101 1110 0101 1001 1010 0100 1110 0010 1000 1100 0010

則(0x0000000000000001*0x07EDD5E59A4E28C2)>>58=0x01
(0x0000000000000002*0x07EDD5E59A4E28C2)>>58=0x03
(0x0000000000000004*0x07EDD5E59A4E28C2)>>58=0x07
(0x0000000000000008*0x07EDD5E59A4E28C2)>>58=0x0F
(0x0000000000000010*0x07EDD5E59A4E28C2)>>58=0x1F
(0x0000000000000020*0x07EDD5E59A4E28C2)>>58=0x3F
(0x0000000000000040*0x07EDD5E59A4E28C2)>>58=0x3E
(0x0000000000000080*0x07EDD5E59A4E28C2)>>58=0x3D

反向檢索的表格，也事先建立：

```
const int index64[64]={
63,0,58,1,59,47,53,2,
60,39,48,27,54,33,42,3,
61,51,37,40,49,18,28,20,
55,30,34,11,43,14,22,4,
62,57,46,52,38,26,32,41,
50,36,17,19,29,10,13,21,
56,45,25,31,35,16,9,12,
44,24,15,8,23,7,6,5};
```

k_moves(bb) array ⇒ 改用以下一個**64**個元素(各佔**64 bits)**的陣列 。

0x0000000000000001 -> 0x0000000000000302
0x0000000000000002 -> 0x0000000000000705
0x0000000000000004 -> 0x000000000000e0a
0x0000000000000008 -> 0x0000000000001c14
0x0000000000000010 -> 0x0000000000003828
0x0000000000000020 -> 0x0000000000007050
0x0000000000000040 -> 0x000000000000e0a0
0x0000000000000080 -> 0x000000000000c040

k_moves[0x01] = 0x0000000000000000000000000000302
k_moves[0x03] = 0x0000000000000000000000000000705
k_moves[0x07] = 0x000000000000000000000000000e0a
k_moves[0x0F] = 0x000000000000000000000000001c14
k_moves[0x1F] = 0x000000000000000000000000003828
k_moves[0x3F] = 0x000000000000000000000000007050
k_moves[0x3E] = 0x00000000000000000000000000e0a0
k_moves[0x3D] = 0x00000000000000000000000000c040
...

/*Generate white king moves*/ 產生白方國王的非吃子步

```
from = index64[perfect_hashing(k)];      // 找到白方國王所在的位置
                    // 也就是 from = index64[(k*0x07EDD5E59A4E28C2)>>58]
targets = k_moves[from] & empty;   // Mask out illegal moves
while (targets ) {
   mask=LS1B(targets);    to=index64[(mask*0x07EDD5E59A4E28C2)>>58];
     //LS1B留下最低位的1:Least Significant One Bit function = targets & -targets
   targets ^= mask;    //去除最低位的1 ，也就是去除一個目的地
   Add_k_moves_ToList(from, to);    // Add white king moves to global movelist
}
```

二補數

例： **k=0x0800000000000000** 位置59

perfect_hashing(k)=(k*0x07EDD5E59A4E28C2)>>58=0x04

from = **index64**[0x04]=**index64**[4]=59

k_moves[59]=0x141C000000000000

targets = **k_moves**[59] & empty

= **0x1414000000000000**

mask=LS1B(targets)

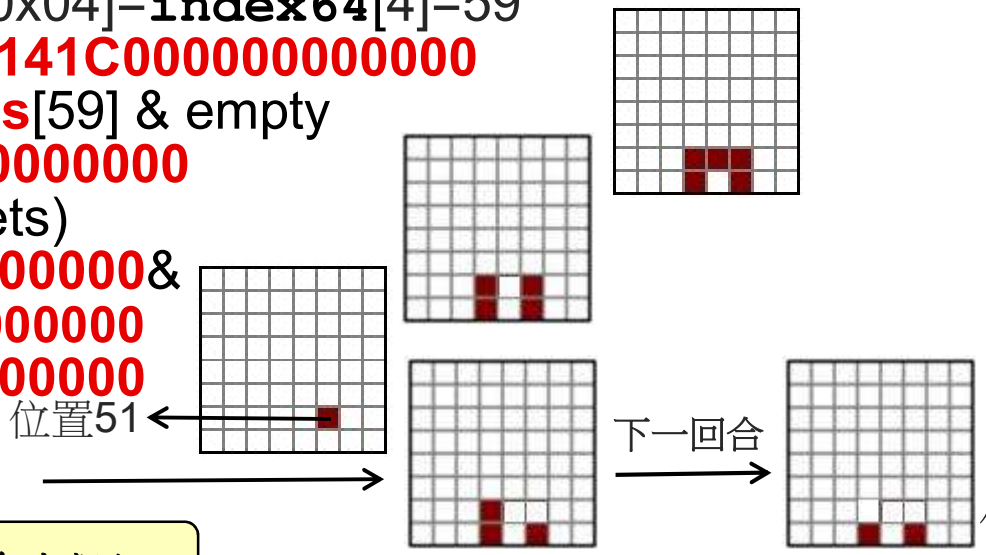
=**0x1414000000000000**&

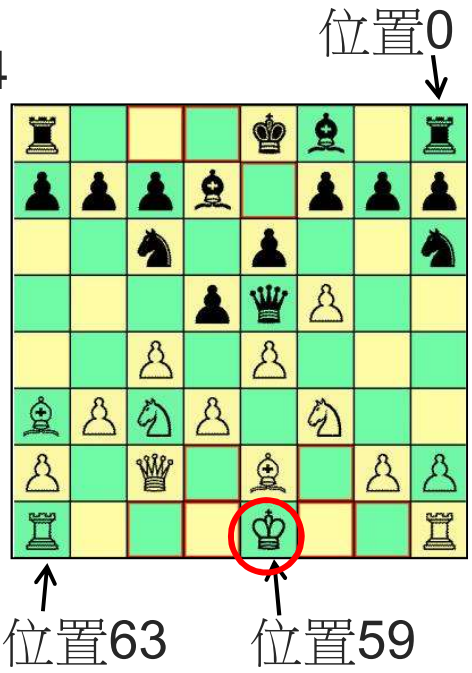
0xEBEC000000000000

=**0x0004000000000000**

to=51

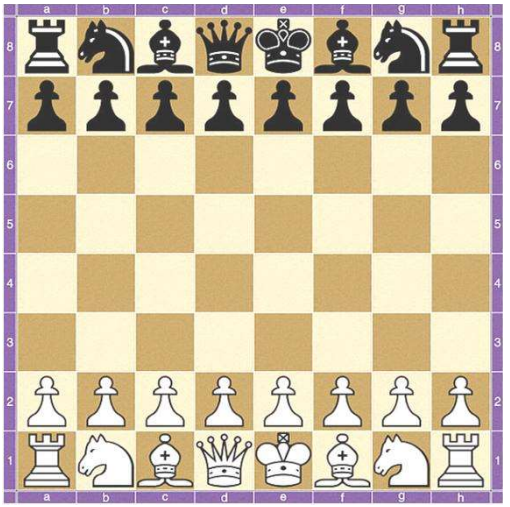
targets^=mask





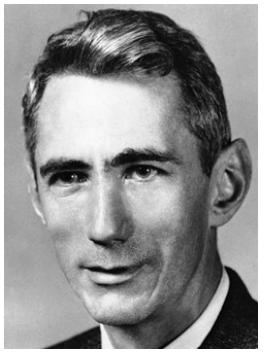
產生白方國王的吃子步作法類似

6.2 Optimal Decisions in Games



- Games are usually too hard to solve.
  - E.g., a chess game
    - Average branching factor: 35.
    - Average moves by each player: 50.
    - Total number of nodes in the search tree: 35^{100} or 10^{154}.
    - Total number of distinct states:10^{40}.
- The solution is a *strategy* that specifies a move for every possible opponent reply.
  - Time limit: how to make the best possible use of time?
    - Calculate the optimal decision may be infeasible.
    - Pruning is needed.
  - Uncertainty: due to the opponent's actions and game complexity.
    - Imperfect information
    - Chance

6.2.1 Minimax Search(1950年Shannon提出)

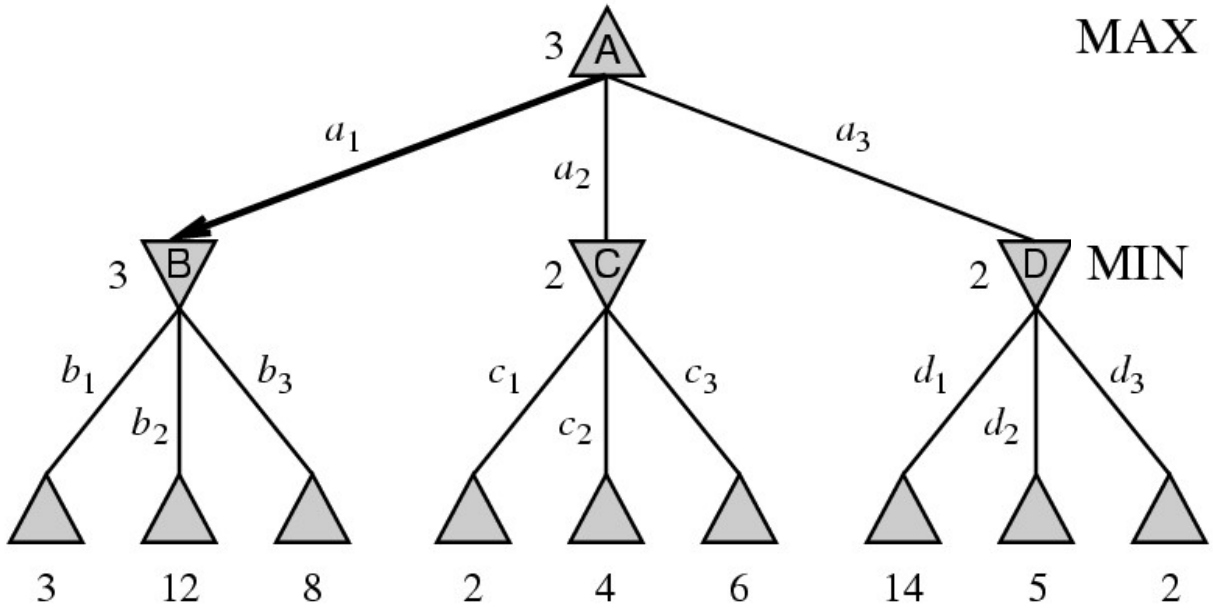


- A strategy/solution for optimal decisions.

  MINIMAX-VALUE(n)=

  $$\begin{cases} \text{UTILITY}(n) & \text{If } n \text{ is a terminal} \\ \max_{s \in successors(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a max node} \\ \min_{s \in successors(n)} \text{MINIMAX-VALUE}(s) & \text{If } n \text{ is a min node} \end{cases}$$

- Example: a trivial 2-ply(層) game.



  - Minmax value of a terminal state is just the utility from the point of view of MAX.
  - Assume two players (MAX and MIN) play optimally (infallibly 無誤地) from the current node to the end of the game.

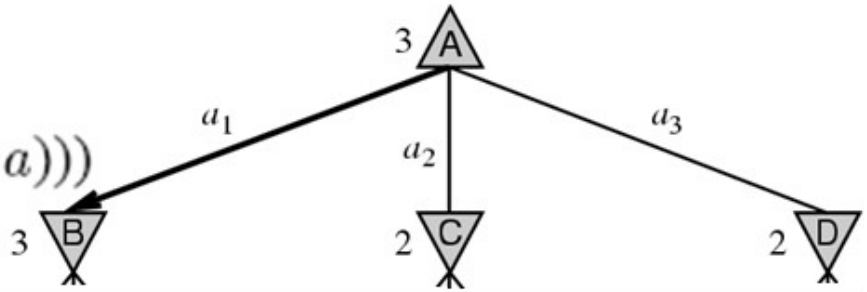
Minimax Search: Algorithm

function MINIMAX-DECISION(*state*) **returns** *an action*
  return $\arg\max_{a \in \text{ACTIONS}(s)}$ MIN-VALUE(RESULT(*state*, *a*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s, a)))
  return v

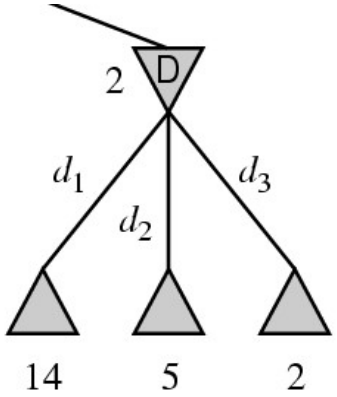
For MAX Node



---

function MIN-VALUE(*state*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow \infty$
  for each *a* **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s, a)))
  return v

For MIN Node

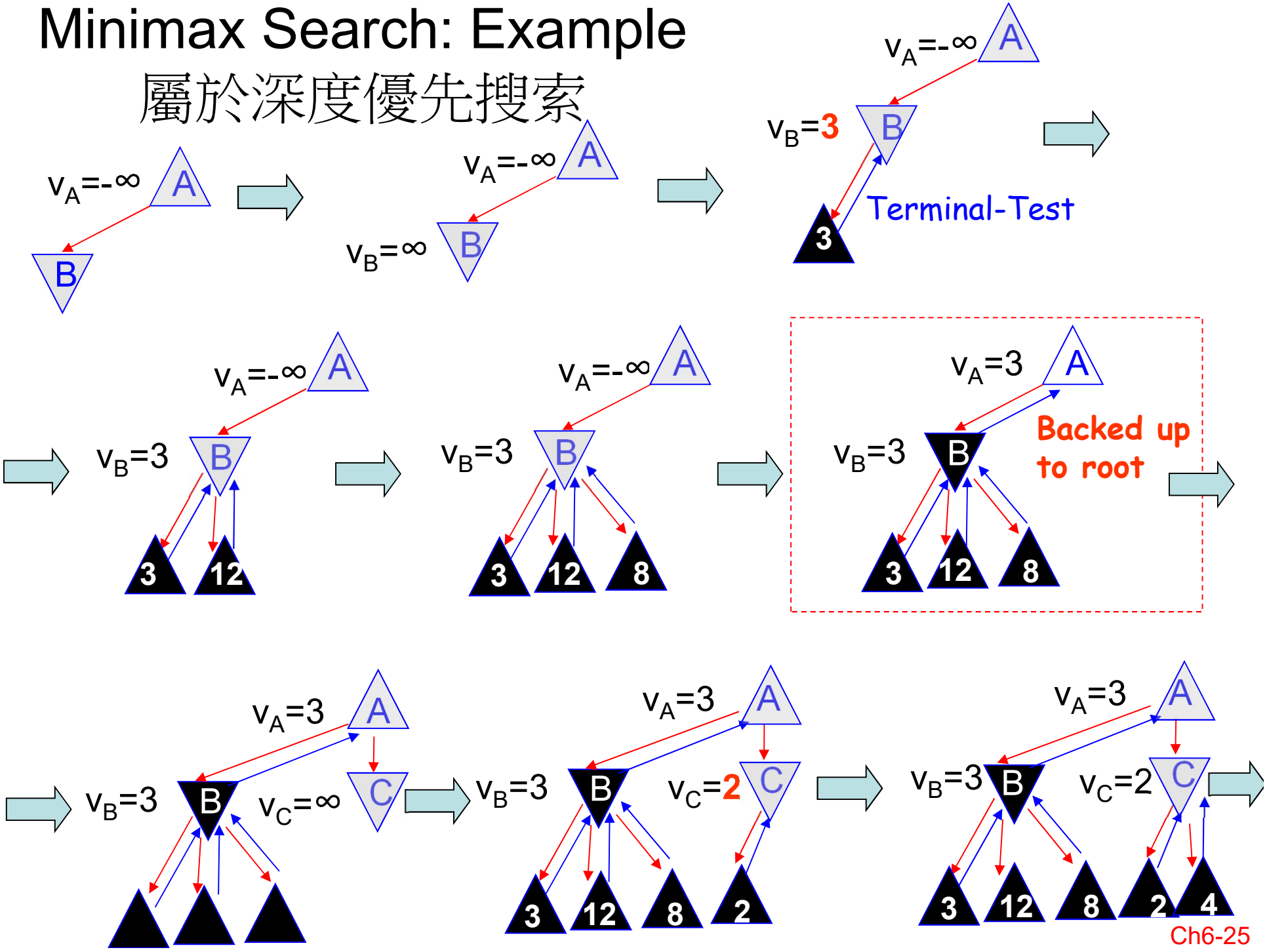


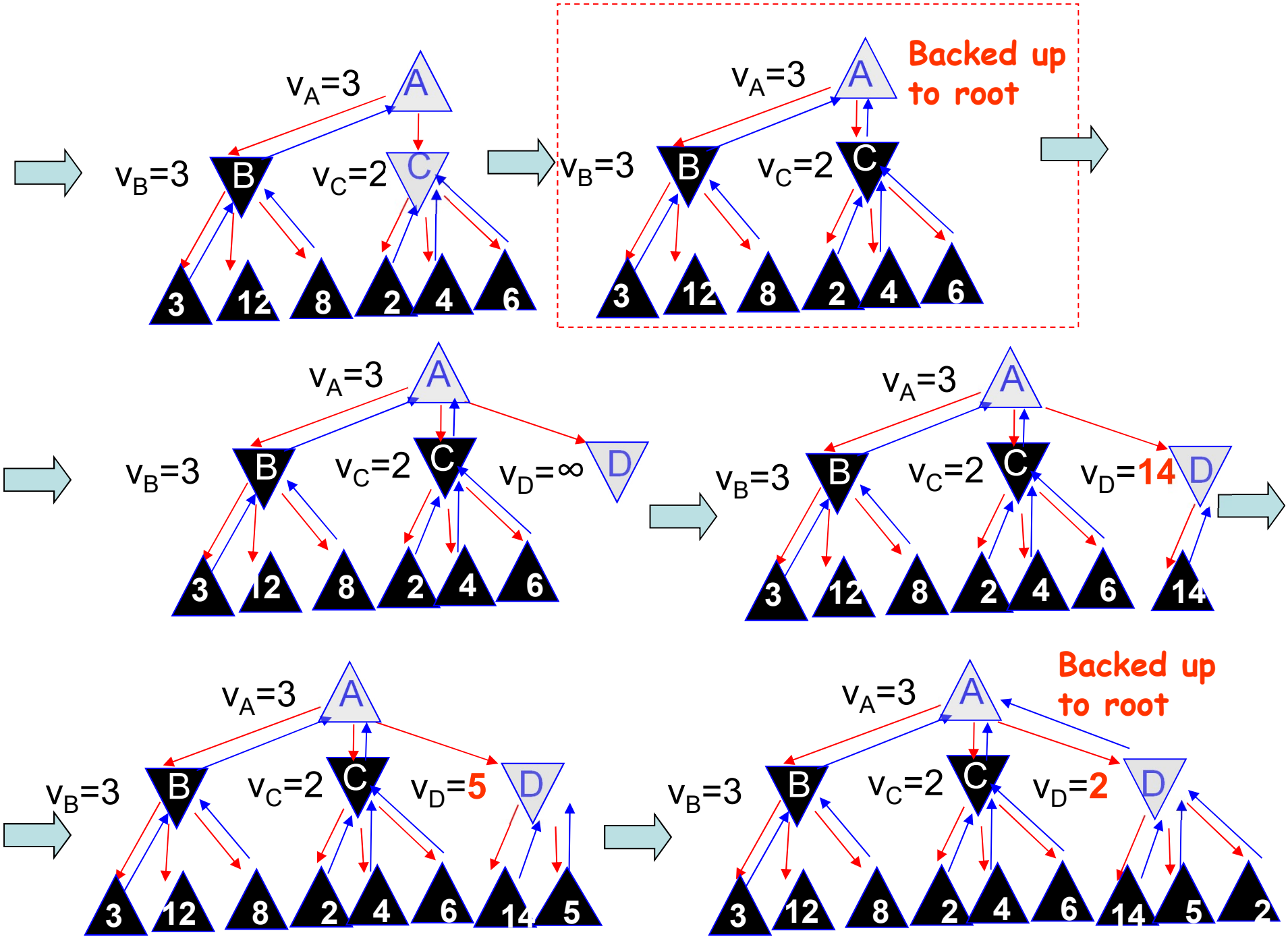
DEMO

https://www.yosenspace.com/posts/computer-science-game-trees.html

Minimax Search: Example

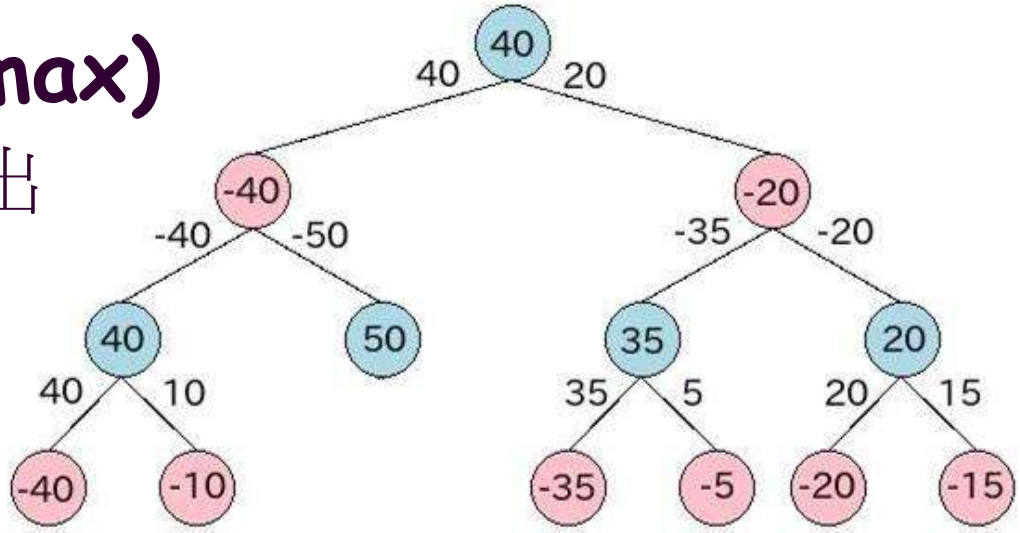
屬於深度優先搜索



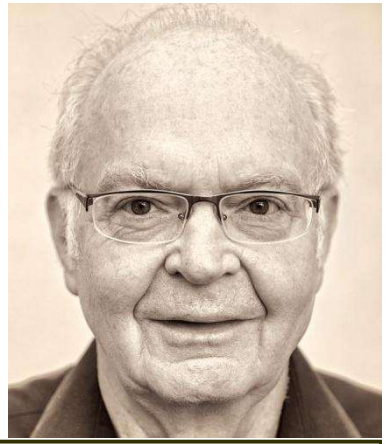

v_A=3
A
v_B=3
B
v_C=2
C
3
12
8
2
4
6
Backed up to root
v_D=∞
D
v_D=14
14
v_D=5
5
Backed up to root

負最大值演算法(Negamax)

1975年Knuth和Moore提出




```
int NegaMax(int depth) {
  int value;
  int best = -INFINITY;
  if (depth <= 0 || 棋局结束){
    return Evaluate();
  }
  GenerateLegalMoves(); //就當前局面，生成並排序一系列走步
  while (MovesLeft()) { //仍有剩餘的走步
    MakeNextMove();     //執行下一個走步
    value = -NegaMax(depth-1); //注意這裡有個負號
    UnmakeMove();       //撤銷這一個走步
    if (value > best) {
      best = value;
    }
  }
  return best;
}
```

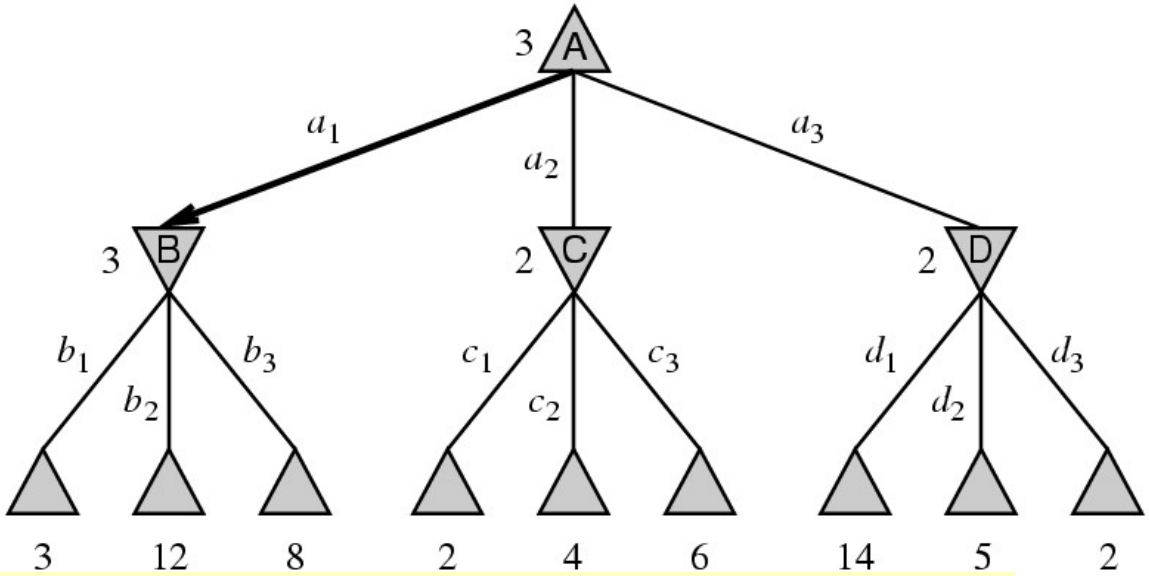


負最大值演算法形式更加簡潔優雅。
每個節點的Evaluate函數評估值為站在子節點的立場上估值後，再取負值。

Properties of Minimax Search

- A complete depth-first, recursive exploration of the game tree.
- Is optimal if the opponent acts optimally.
- Space complexity: $O(bm)$ or $O(m)$ (when successors are generated one at a time.)
- Time complexity=$O(b^m)$, m = max. depth of the tree.

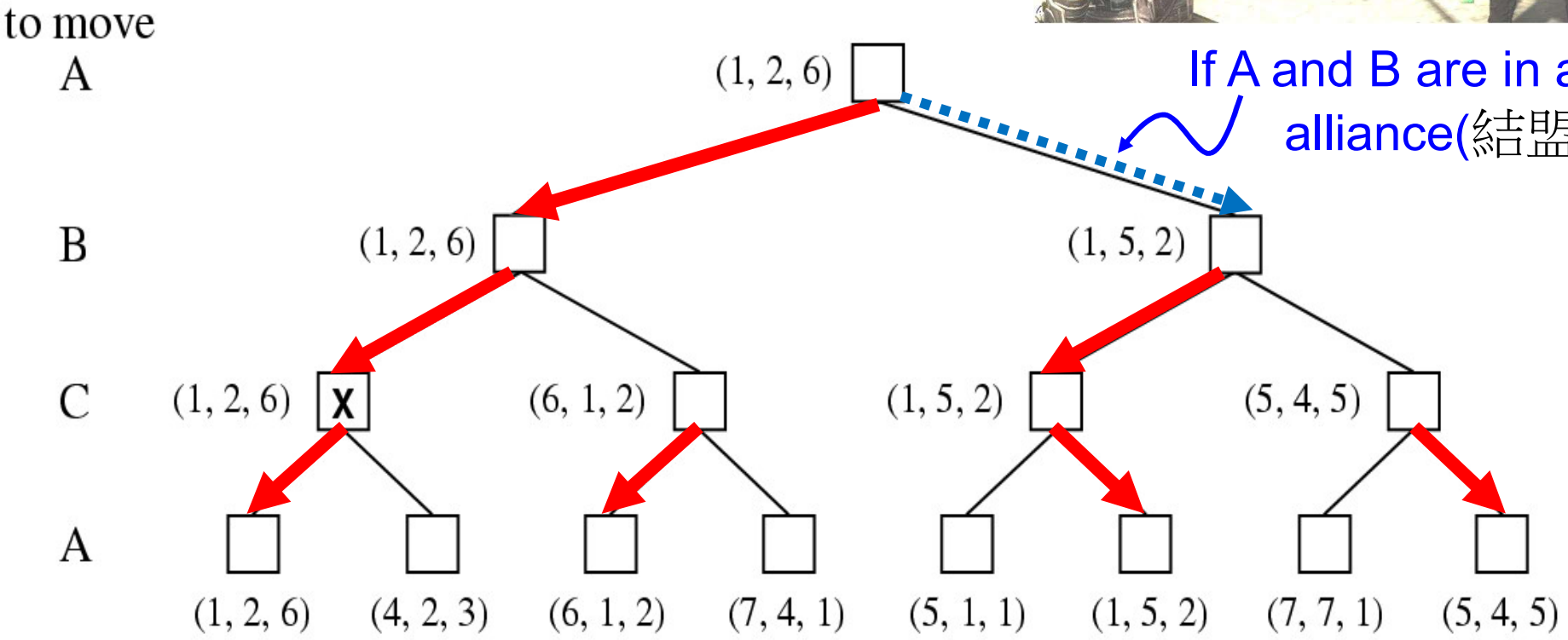
The number of nodes to examine is exponential.



For chess, b ≈ 35, m ≈ 100 for "reasonable" games.
i.e., exact solution is completely infeasible.

6.2.2 Optimal Decisions in Multiplayer Games

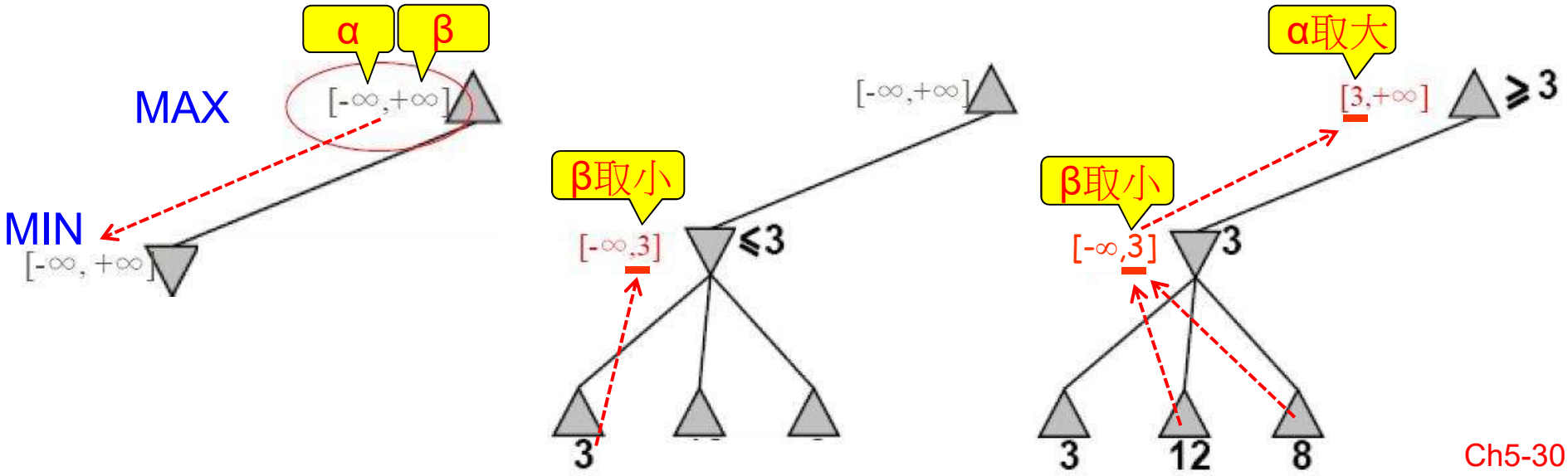
- Extend the minimax idea to multiplayer games.
- Replace the single value for each node with a vector of values (utility vector).

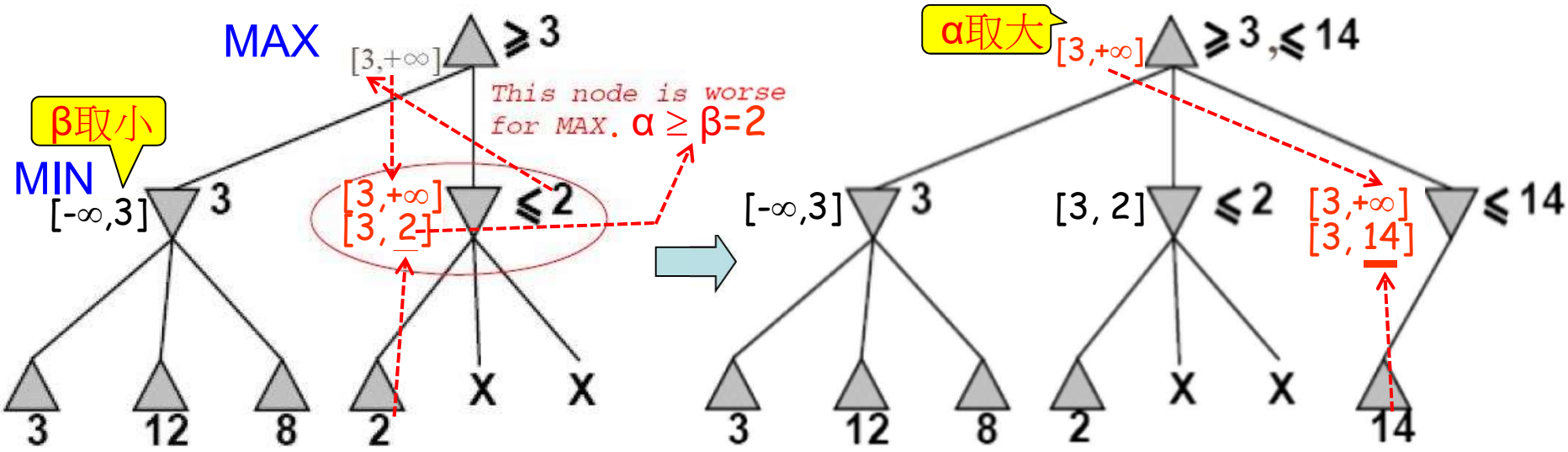


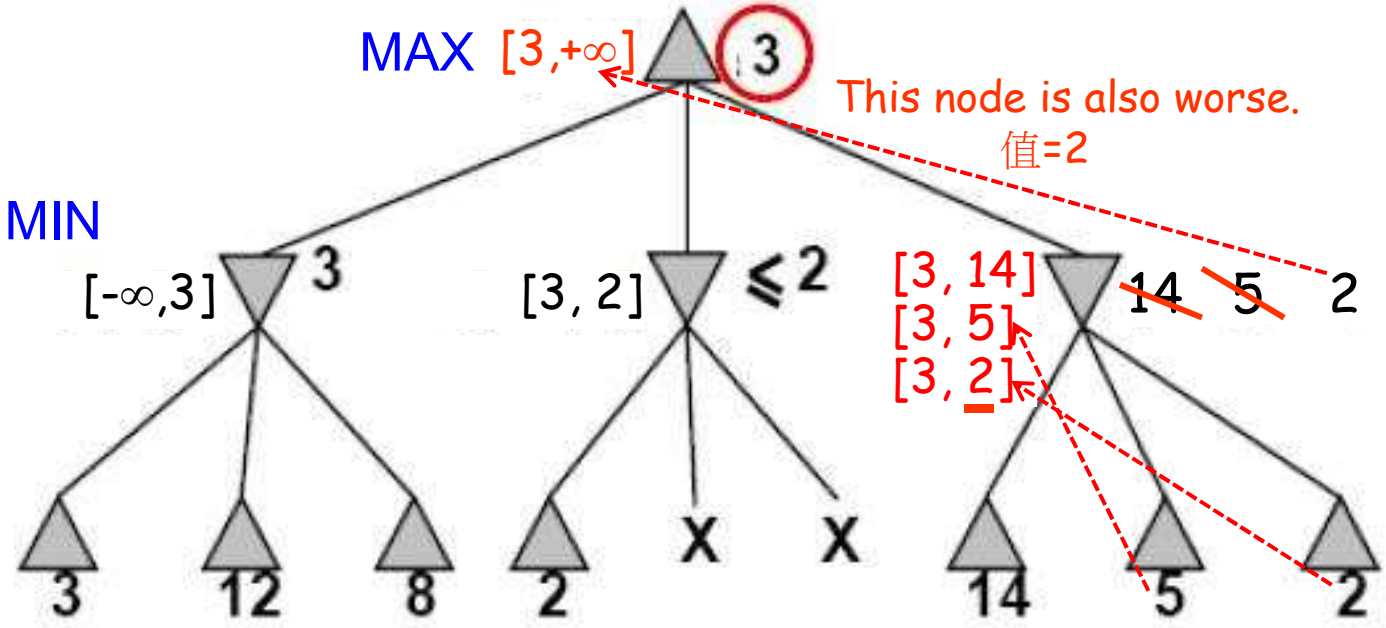
- Alliances(結盟) among players would be involved sometimes.
  - e.g., A and B form an alliance to attack C.

6.2.3 α-β Pruning(Alexander Brudno在1963 年提出)

- Applied to the minimax tree.
- Return the same moves as minimax would, but prune away branches that can't possibly influence the final decision.
- α: the value of best (highest-value) choice so far in search of MAX.
- β: the value of best (lowest-value) choice so far in search of MIN.







$$\text{Value}(root) = \max(\min(3,12,8), \min(2,x,y), \min(14,5,2))$$
$$= \max(3, \min(2,x,y), 2) = \max(3,z,2) = 3 \qquad \text{where } z \le 2$$

DEMO

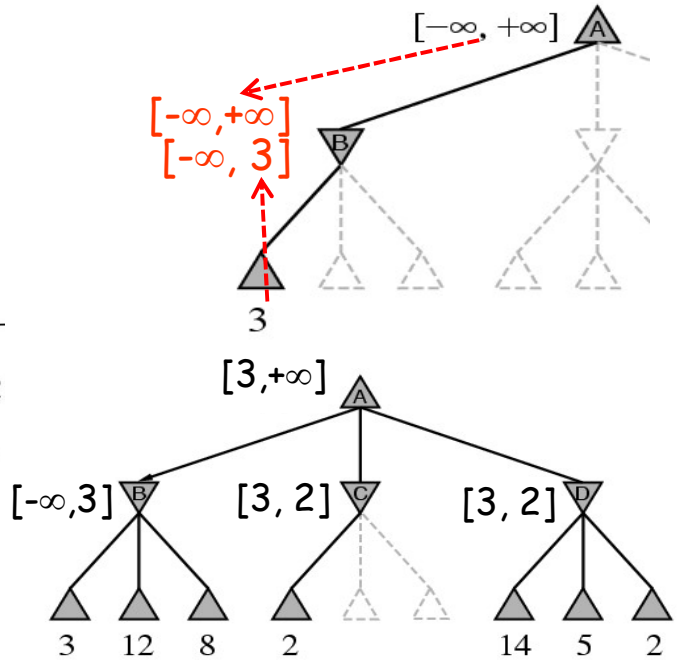
https://www.yosenspace.com/posts/computer-science-game-trees.html

α-β Pruning Algorithm

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  $v \leftarrow$ MAX-VALUE($state, -\infty, +\infty$)
  return the *action* in ACTIONS(*state*) with value v

For MAX Node

function MAX-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow -\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if $v \geq \beta$ **then return** v
    $\alpha \leftarrow$ MAX(α, v)
  return v



Pruning: If one of its children has value larger than that of its best MIN predecessor node , return immediately.

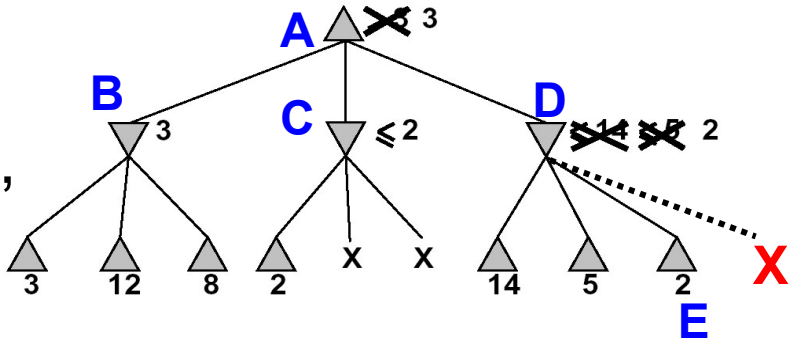
For MIN Node

function MIN-VALUE($state, \alpha, \beta$) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  $v \leftarrow +\infty$
  for each a **in** ACTIONS(*state*) **do**
    $v \leftarrow$ MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if $v \leq \alpha$ **then return** v
    $\beta \leftarrow$ MIN(β, v)
  return v

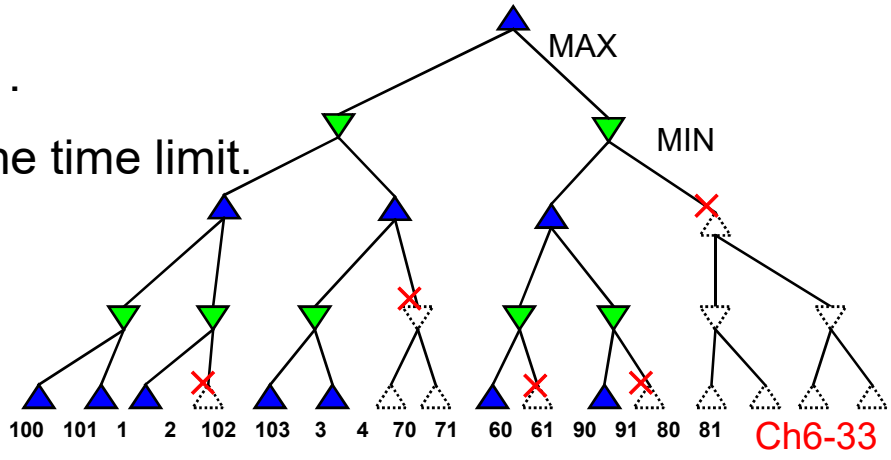
Pruning: If one of its children has value lower than that of its best MAX predecessor node, return immediately.

Properties of α-β Pruning

- Pruning does not affect the final result.
- The effectiveness of α-β pruning is highly dependent on the order in which the successors are examined.
  - Worthwhile to try to examine first the successors that are likely to be best
  - e.g., If the third successor "E" of node D has been generated first, the other three successors can be pruned.

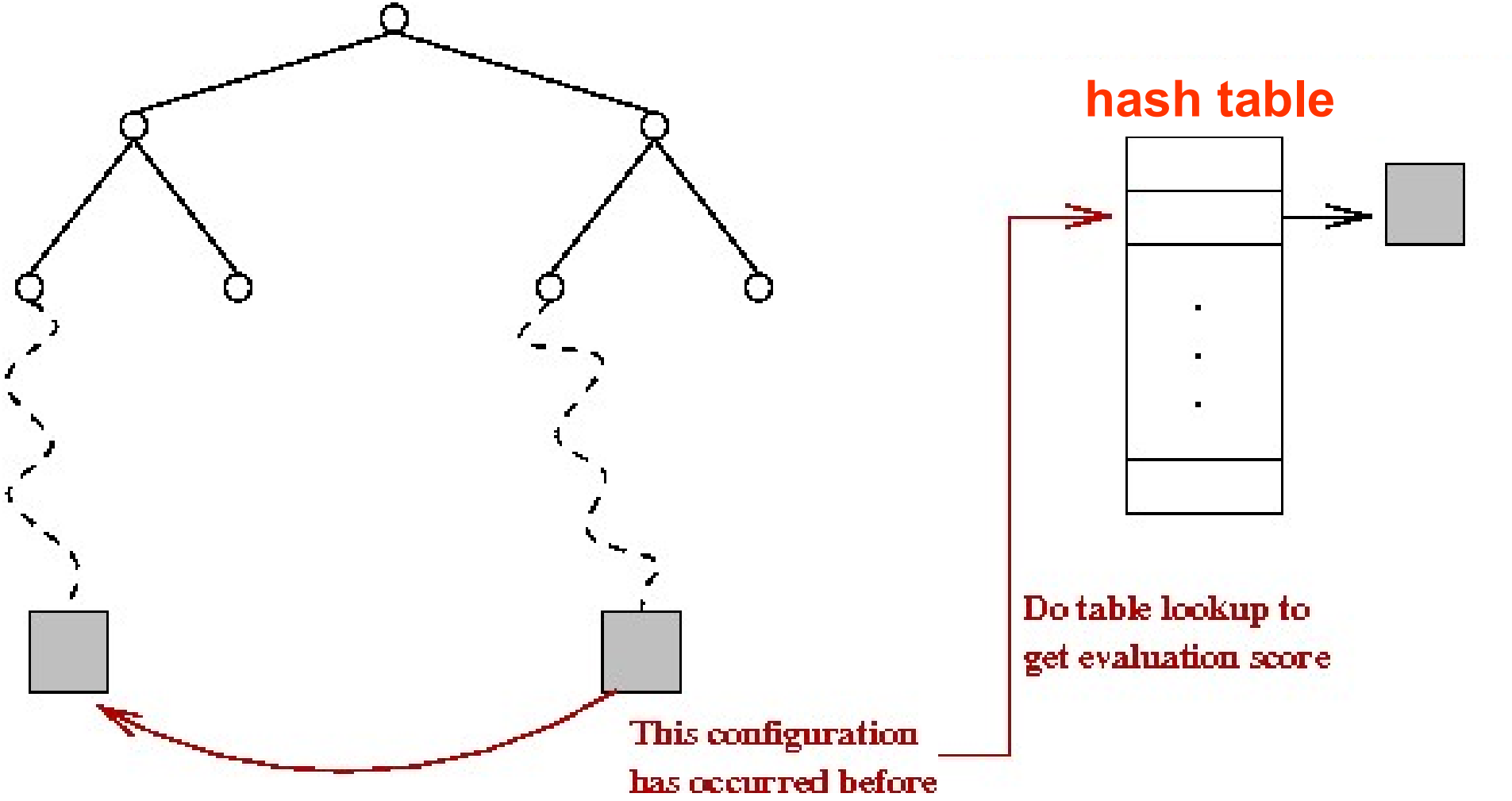


- If "perfect ordering" can be achieved.
  - Time complexity: $O(b^{m/2})$.
  - Effective branching factor becomes: $b^{1/2}$.
  - Can *double* the depth of search within the time limit.
- If "random ordering"
  - Time complexity ≈ $O(b^{3m/4})$.



同形表/置換表(Transposition Tables)

使用同形表（以hash table實現）是為了解決重複節點（不同著法順序導致相同局面）的重複搜索問題。

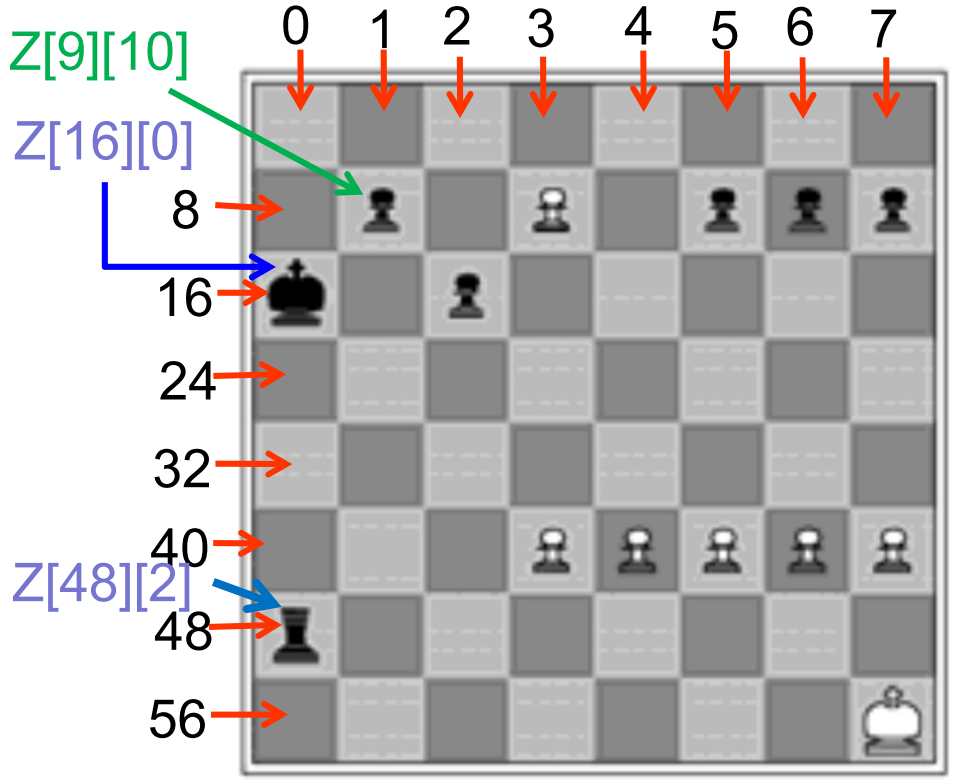


Transposition Tables & Zobrist Hashing

rand()傳回0~32767(15位元)，要得到64位元的亂數rand64()需再加工一下。

Zobrist hash的優點在於可逐步更新(incremental update)，速度快。

- 產生**64**位元亂數: U64 rand64(void) {
  return rand() ^ ((U64)rand() << 15) ^
  ((U64)rand() << 30) ^ ((U64)rand() << 45) ^
  ((U64)rand() << 60); }
- 初始化整個棋盤的Hash value:
  for(Num = 0; Num<31;++Num)
  {hash_value ^= Z[Pos][Num];}
- 走子更新： Num棋子由OldPos移到NewPos，吃了EatNum棋子
  hash_value ^= Z[OldPos][Num];
  hash_value ^= Z[NewPos][Num];
  hash_value ^= Z[NewPos][EatNum];

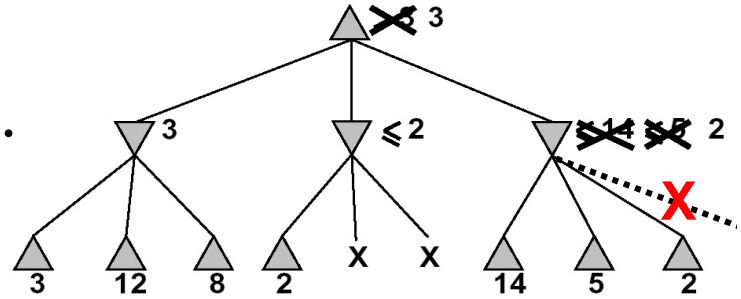


64格x32顆棋子=2048個亂數
U64 Z[Pos] [Num]
Pos是棋子的位置
Num是棋子的編號
Z[0..63][0..31]

hash_value=
Z[9][10]⊕Z[11][26]⊕
Z[13][13]⊕Z[14][14]⊕
Z[15][15]⊕Z[16][0]⊕
Z[18][10]⊕Z[43][27]⊕
Z[44][28]⊕Z[45][29]⊕
Z[46][30]⊕Z[47][31]⊕
Z[48][2]⊕Z[63][16]

6.3 Heuristic α-β Pruning

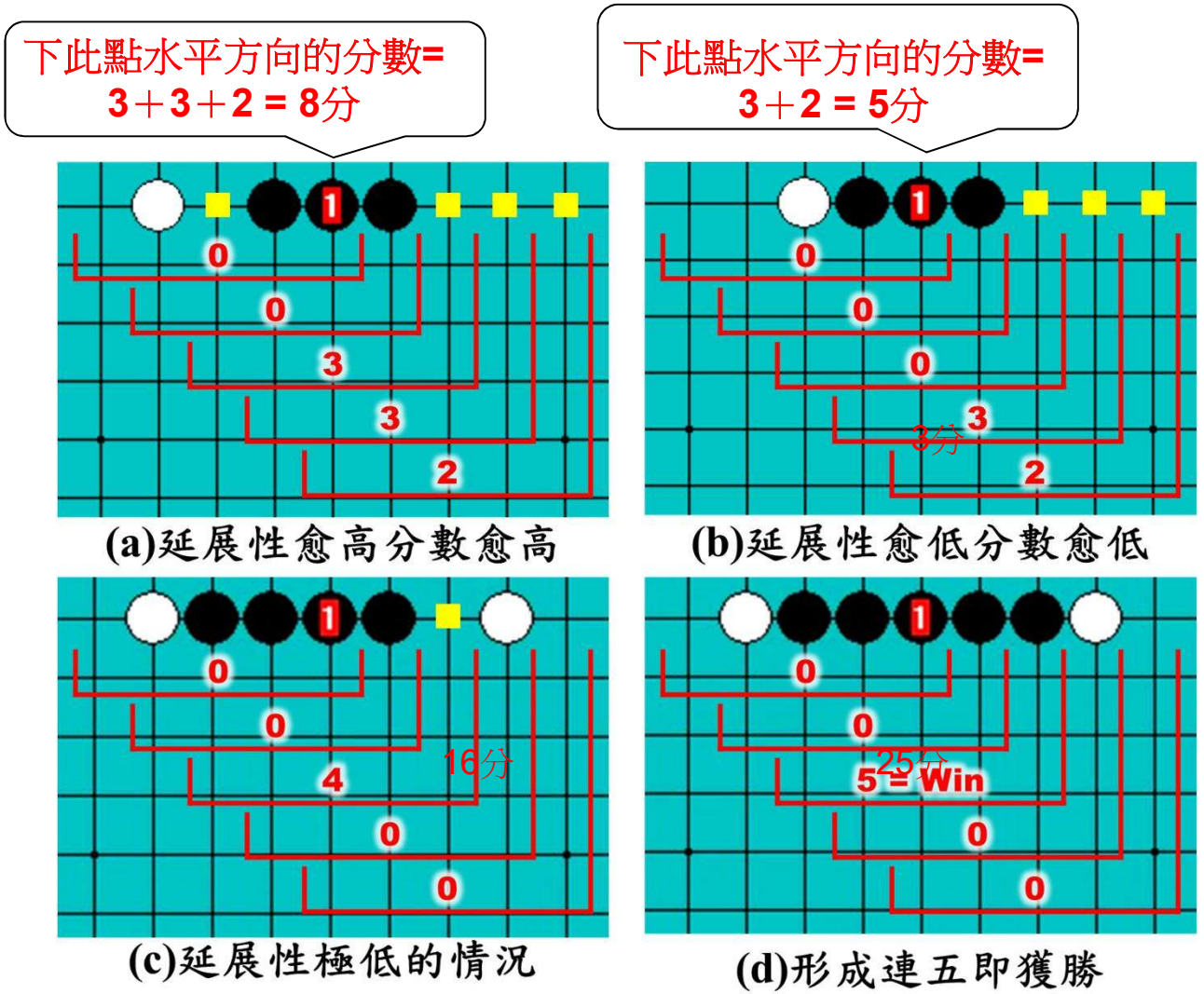
- One problem of alpha-beta is that it still has to search to terminal states.
  ⇒ The depth is usually not practical.
- We should cut off the search earlier and apply a heuristic evaluation function.
  - terminal test → cut-off test
  - utility function → heuristic evaluation function審局函數




```
function MAX-VALUE(state, α, β) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for each a in ACTIONS(state) do
        v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v
```

6.3.1 Evaluation function(審局函數)

範例：五子棋，陳志宏的Sliding Window



審局分數 = 4個方向的分數平方和。

例：象棋的審局函數

- 通常會將不同兵種給予不同的子力分數。
- 審局時只要將雙方的子力分數總分相減，就可以得到一個估計值。
- 其它常見的考慮因素如下：
  - 位置重要性
  - 棋子靈活度
  - 威脅與保護


帥 10000


仕 200


相 200


俥 1000


傌 420


炮 450


兵 100

車

8	9	8	10	9	10	8	9	8
9	9	8	11	12	11	8	9	9
8	8	7	11	11	11	7	8	8
7	8	10	11	11	11	10	8	7
6	7	6	10	10	10	6	7	6
6	9	6	9	9	9	6	9	6
5	6	5	8	8	8	5	6	5
2	6	5	8	8	8	5	6	2
4	4	3	8	1	8	3	4	4
1	4	2	7	7	7	2	4	1

馬

4	3	3	6	3	6	3	3	4
3	8	8	6	3	6	8	8	3
7	6	8	8	8	8	8	6	7
4	9	6	9	6	9	6	9	4
4	7	6	7	8	7	6	7	4
0	3	6	4	4	4	6	3	0
0	3	3	2	4	2	3	3	0
0	2	2	2	0	2	2	2	0
0	0	0	1	10	1	0	0	0
0	2	0	0	0	0	0	2	0

包

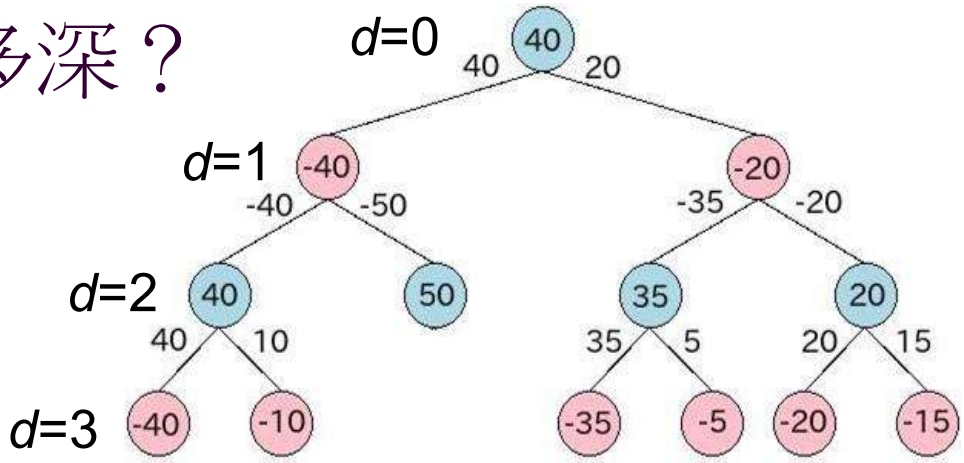
6	5	0	0	0	0	0	5	6
1	1	0	1	0	1	0	1	1
1	1	0	0	0	0	0	1	1
0	2	0	0	10	0	0	2	0
0	0	0	0	9	0	0	0	0
0	1	1	0	8	0	1	1	0
0	0	0	0	7	0	0	0	0
1	0	1	2	6	2	1	0	1
0	1	2	2	5	2	2	1	0
0	0	0	2	1	2	0	0	0

卒

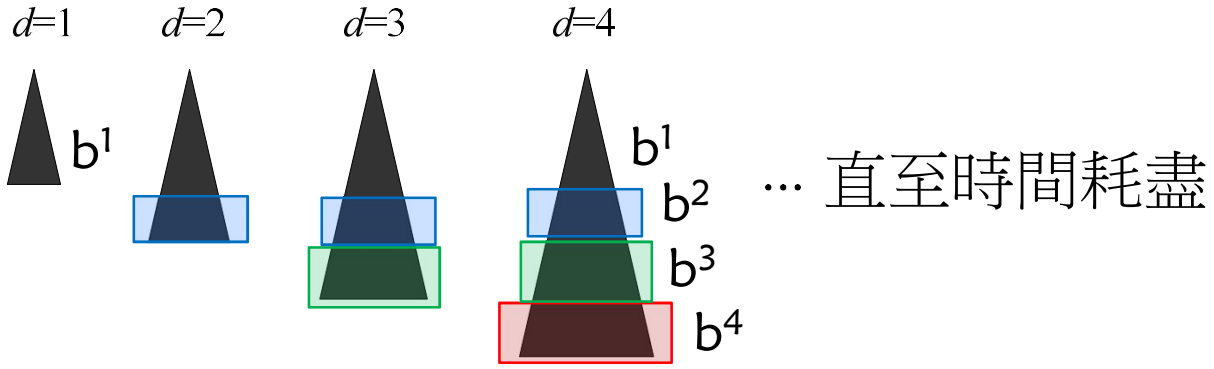
-30	-30	-30	-20	-20	-20	-30	-30	-30
7	9	14	18	30	18	14	9	7
6	8	12	14	14	14	12	8	6
5	7	10	12	12	12	10	7	5
4	6	8	10	10	10	8	6	4
1	0	2	0	3	0	2	0	1
1	0	1	0	1	0	1	0	1
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

深度優先搜索(DFS)要多深？

- DFS 難以預測解的深度；
- DFS 難以控制時間，深度太深會超時，太淺會找不到解。

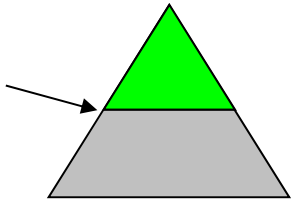


解法：**DFID逐層加深搜索**(Depth First Iterative Deepening)

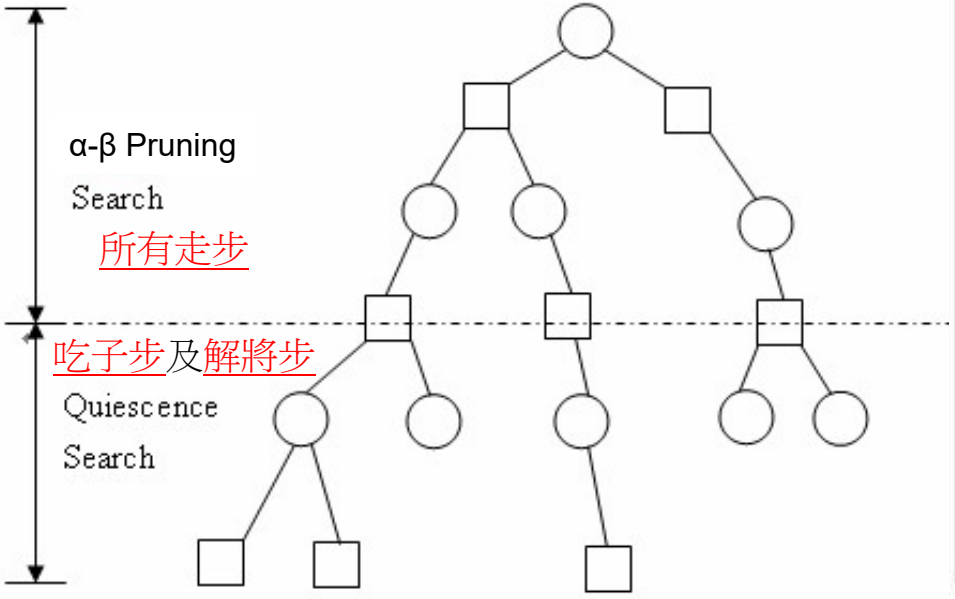


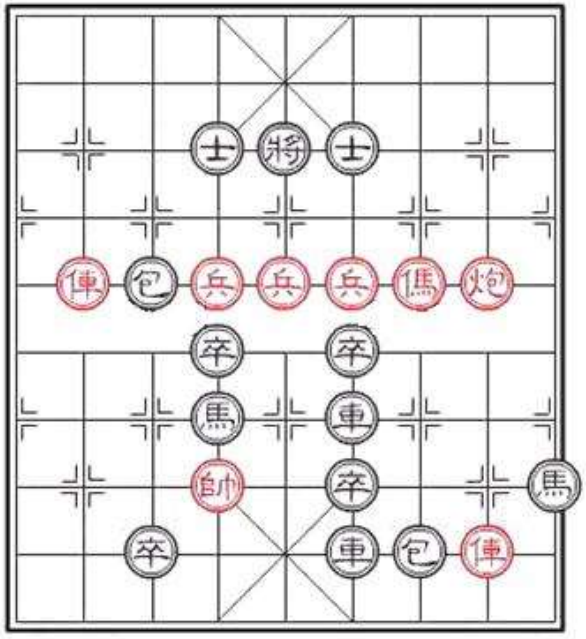
- 以深度優先的方式先搜1層、再搜2層、再搜3層、…，直到找到解or逾時。
  - DFID逐層加深的額外代價並不高。if b=10 and d=4,
    N(DFS) =10+100+1000+10000=11110
    N(DFID)=40+300+2000+10000=12340

6.3.2 Cutting off search



- 如果在搜尋最後一層發生兌子的情況，因為搜尋層數不足以看到下一手對方吃回我方子力，會誤以為己方佔優，而對盤面形勢造成誤判。這種稱為**水平效應(horizon effect)**。
- 解法：**寧靜搜尋(quiescence search)**，為一種選擇性搜尋，只針對吃子步及解將步繼續展開，最後到達寧靜盤面(沒有子力交換的盤面)為止。



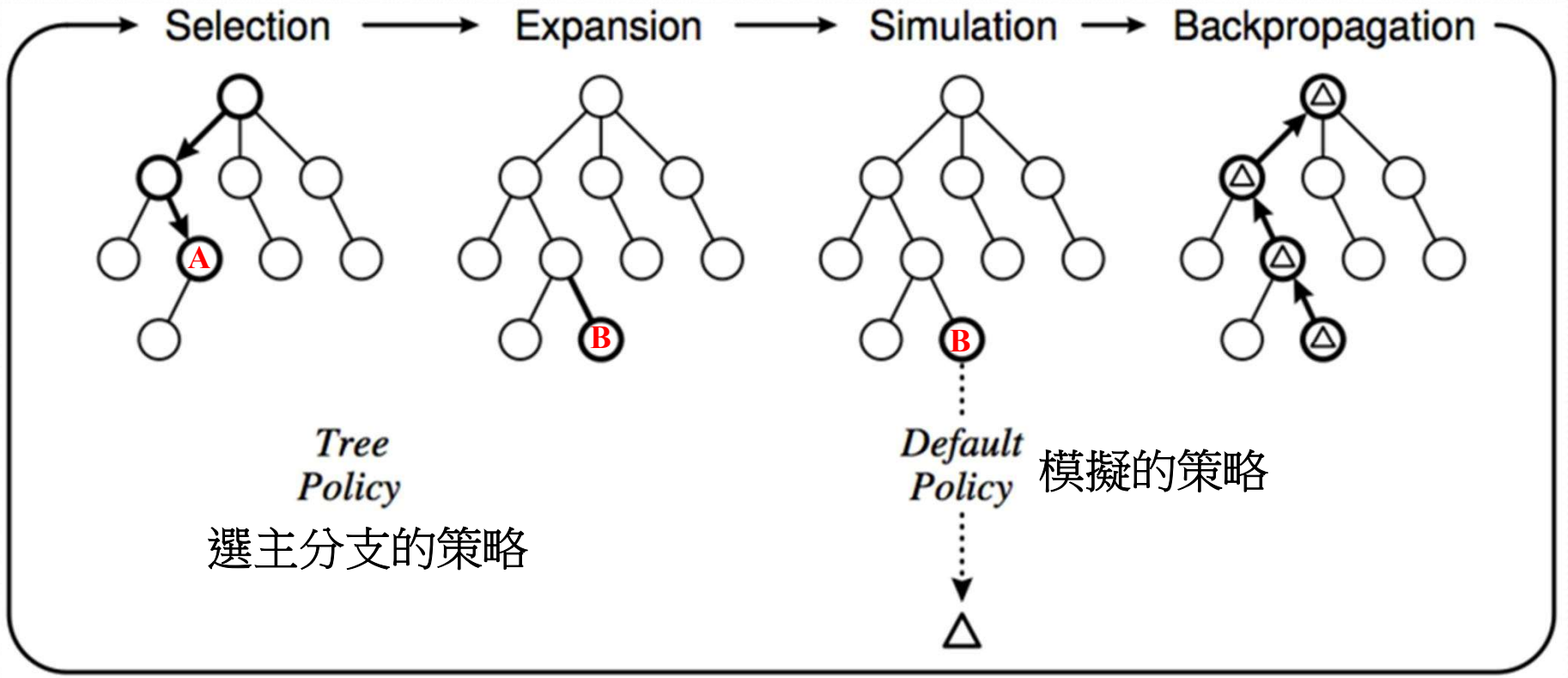


象棋殘局：
一元

6.4 蒙地卡羅樹搜索(Monte Carlo Tree Search)

2006年Rémi Coulom提出。是一個通用型的算法。

1. **Selection**：從根節點往下選主路徑，直到某個節點**A**，其下至少還有子節點沒有展開。
2. **Expansion**：選節點**A**的一個尚未展開的節點**B**產生新節點。
3. **Simulation**：從節點**B**開始隨機模擬對局，直到得出勝負。
4. **Backpropagation**：從節點**B**往上更新分數回溯到根節點。



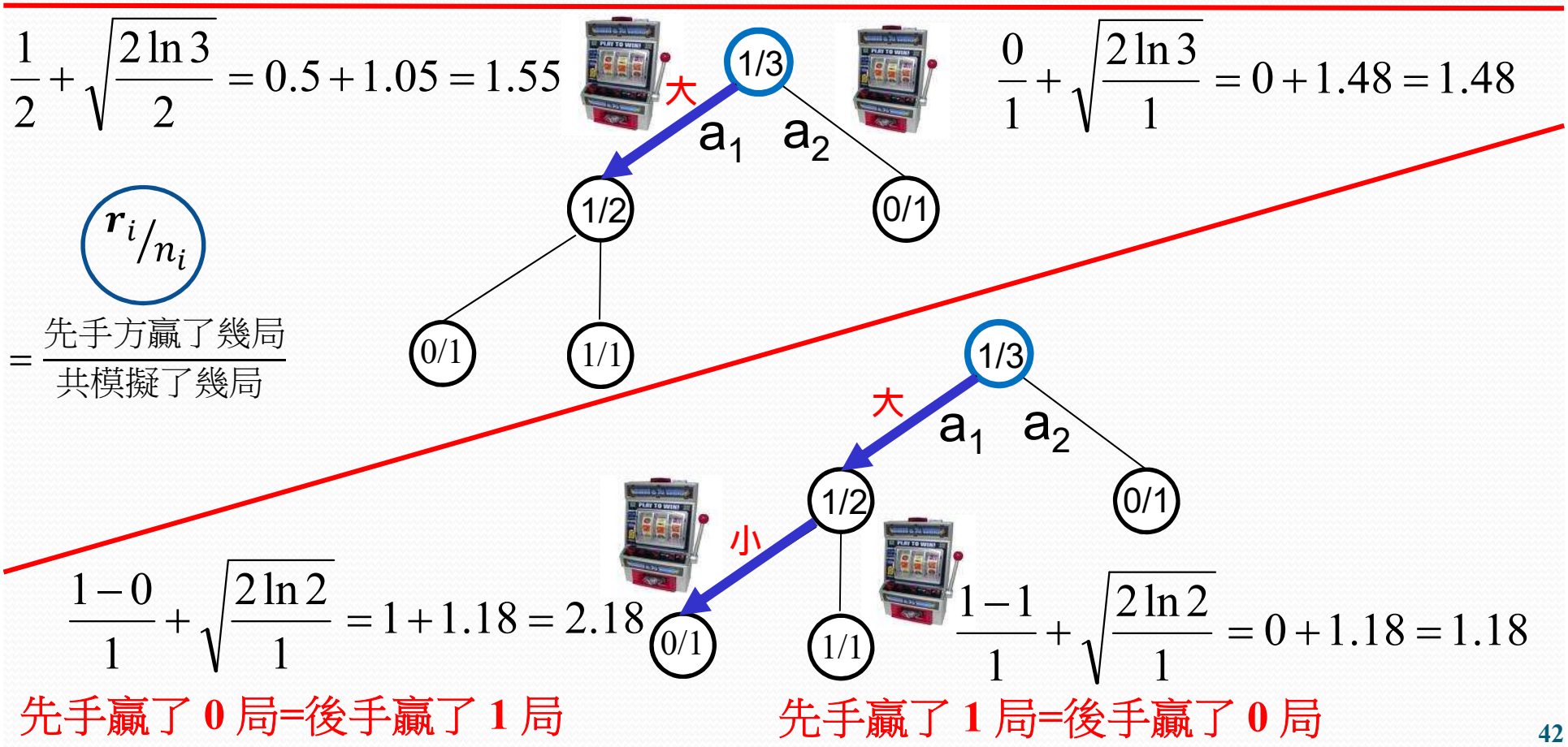
信賴上限應用樹演算法(UCB applied to Trees)

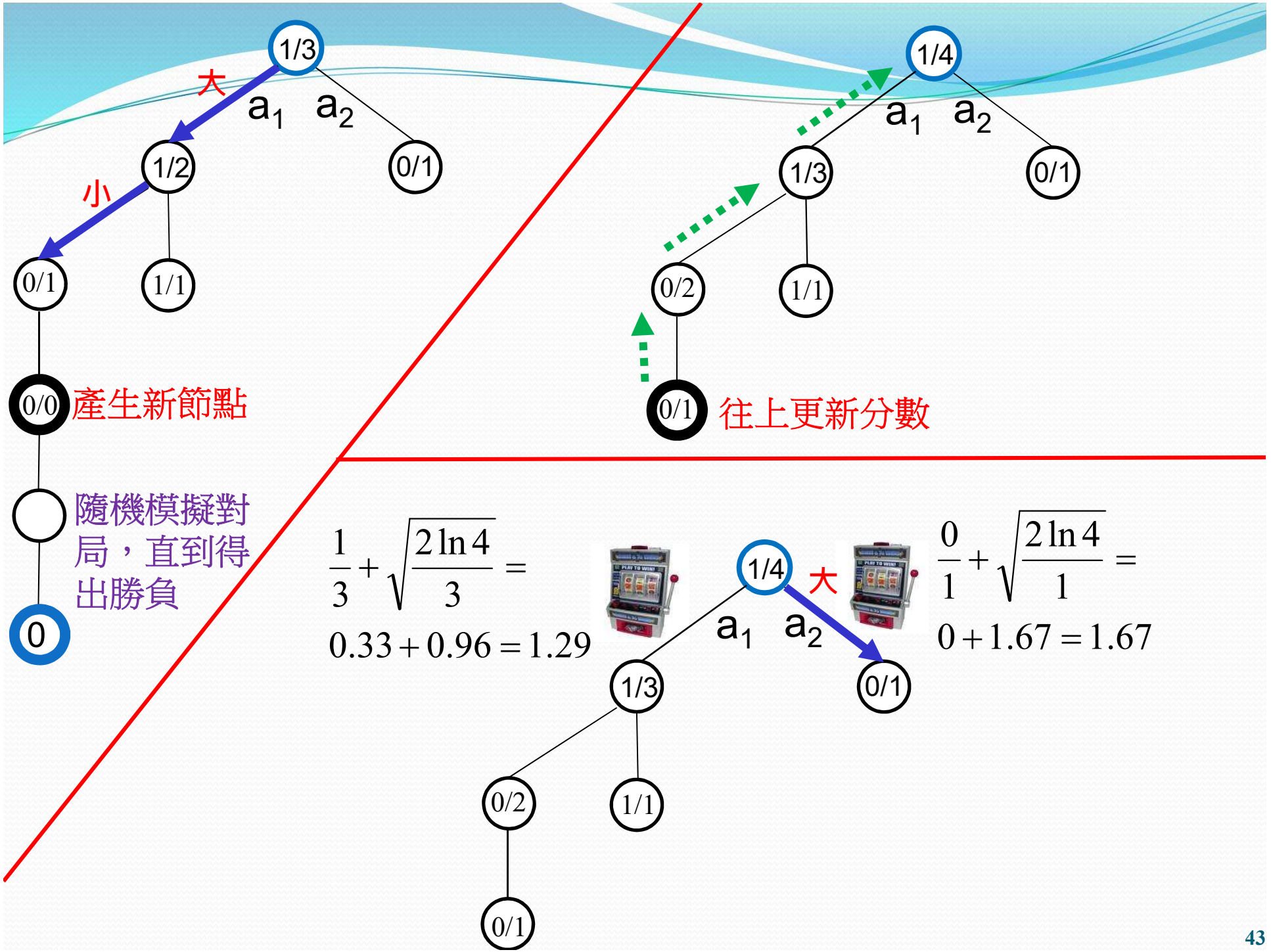
2006年L. Kocsis and C. Szepesvári提出UCT算法。

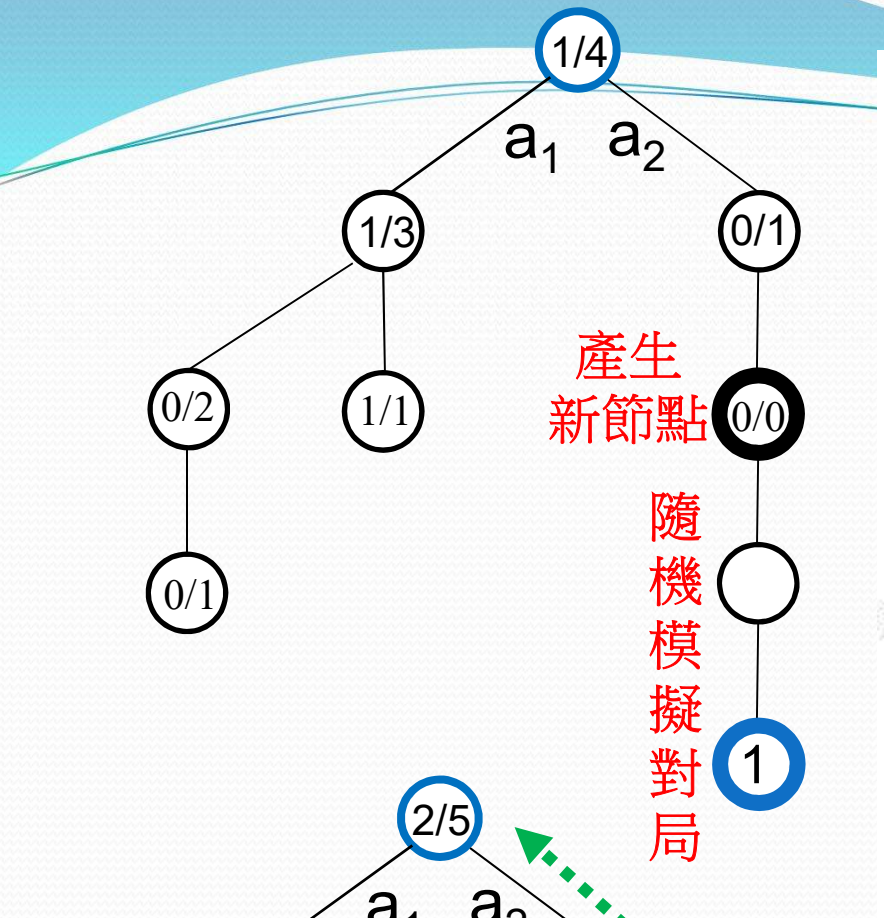
2006年S. Gelly 等人實作了UCT 程式MoGo，榮獲ICGA 2007金牌。

UCT ＝ MCTS ＋ UCB1

$$UCB1 = \frac{r_i}{n_i} + \sqrt{\frac{2\ln n}{n_i}}$$

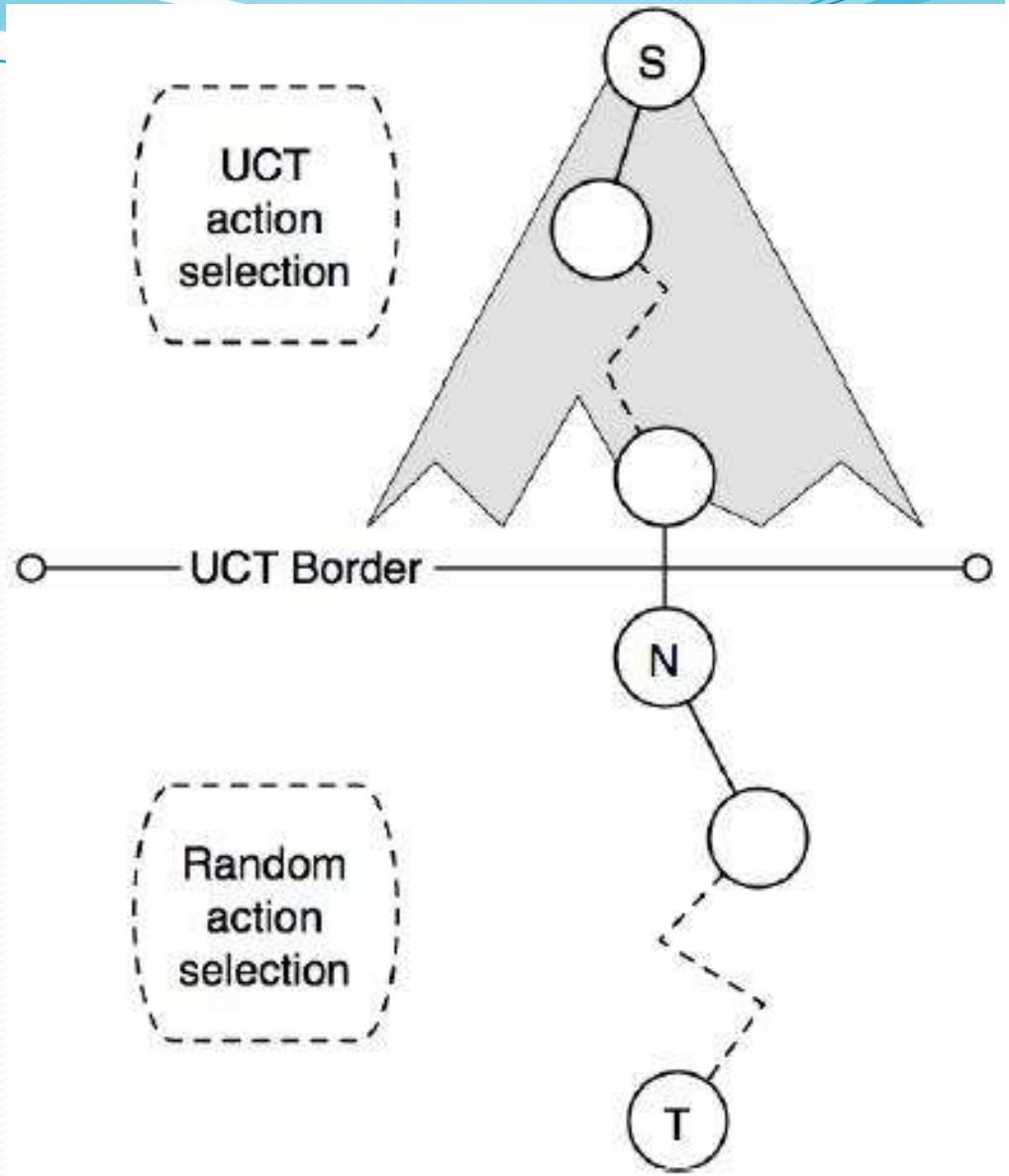



1/3
大
a1
a2
1/2
0/1
小
0/1
1/1
0/0
產生新節點
隨機模擬對局，直到得出勝負
0
1/4
a1
a2
1/3
0/1
0/2
1/1
0/1
往上更新分數
$\frac{1}{3}+\sqrt{\frac{2\ln 4}{3}}=$
$0.33+0.96=1.29$
1/4
大
a1
a2
0/1
1/3
0/2
1/1
0/1
$\frac{0}{1}+\sqrt{\frac{2\ln 4}{1}}=$
$0+1.67=1.67$



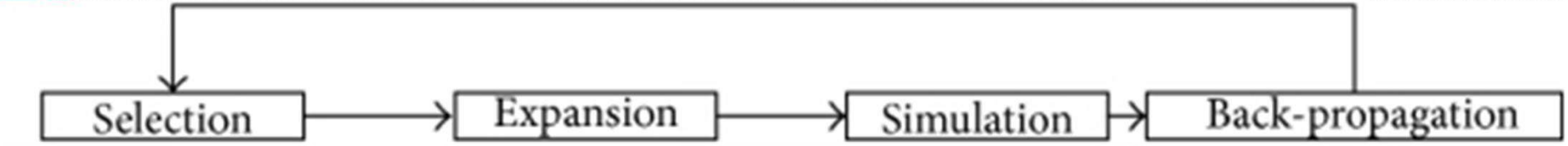
產生新節點

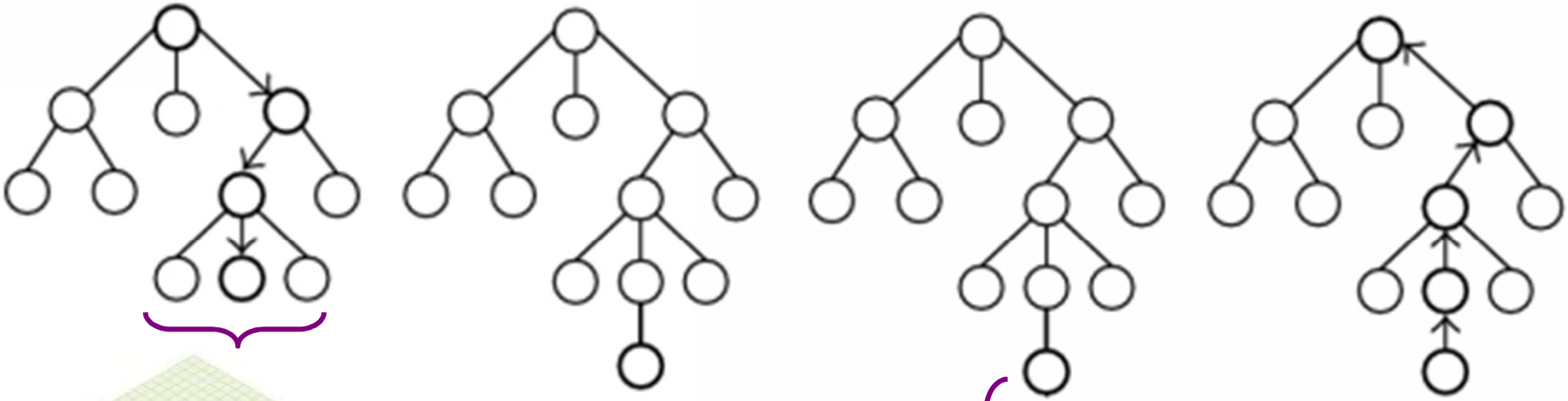
隨機模擬對局

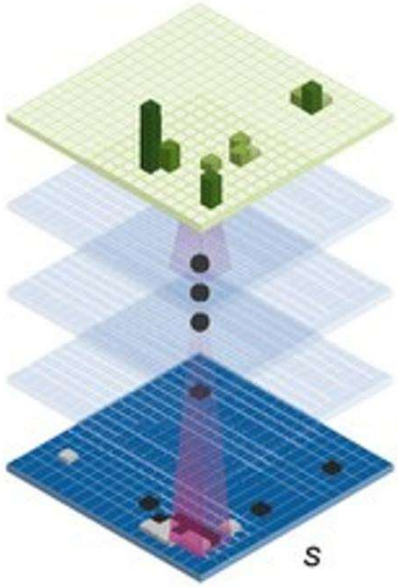


UCT算法最後會選贏率最高的走步**(a_2)**下。

MCTS+深度學習(Deep Learning)

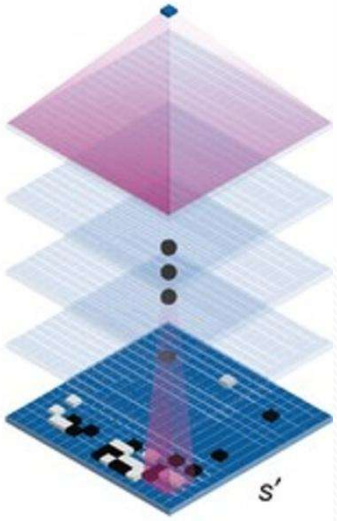




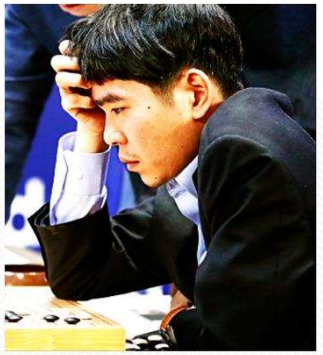


策略網路

評價網路



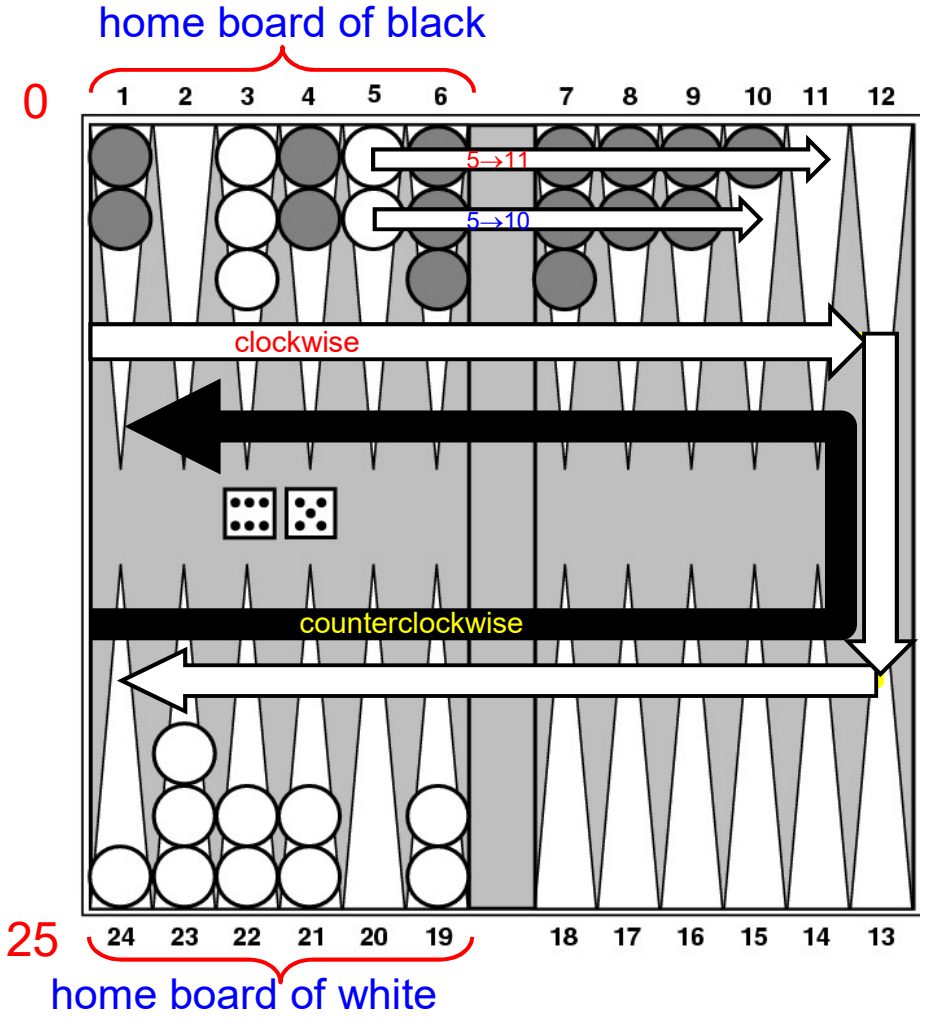
AlphaGo致勝之道！



6.5 Stochastic (Nondeterministic) Games

- Games that combine luck and skill.
  - Dice are rolled at the beginning of a player's turn to determine the legal moves.
  - e.g., Backgammon 西洋雙陸棋

http://www.bkgm.com/motif/go.html



1. **Goal of the game**: move all one's pieces off the board.
2. White moves clockwise toward 25. Black moves counterclockwise toward 0.
3. A piece can move to any position unless there are multiple opponent pieces there.
4. If the position to be move to has only one opponent, the opponent will be captured and restarted over.
5. When one's all pieces are in his home board, the pieces can be moved off the board.

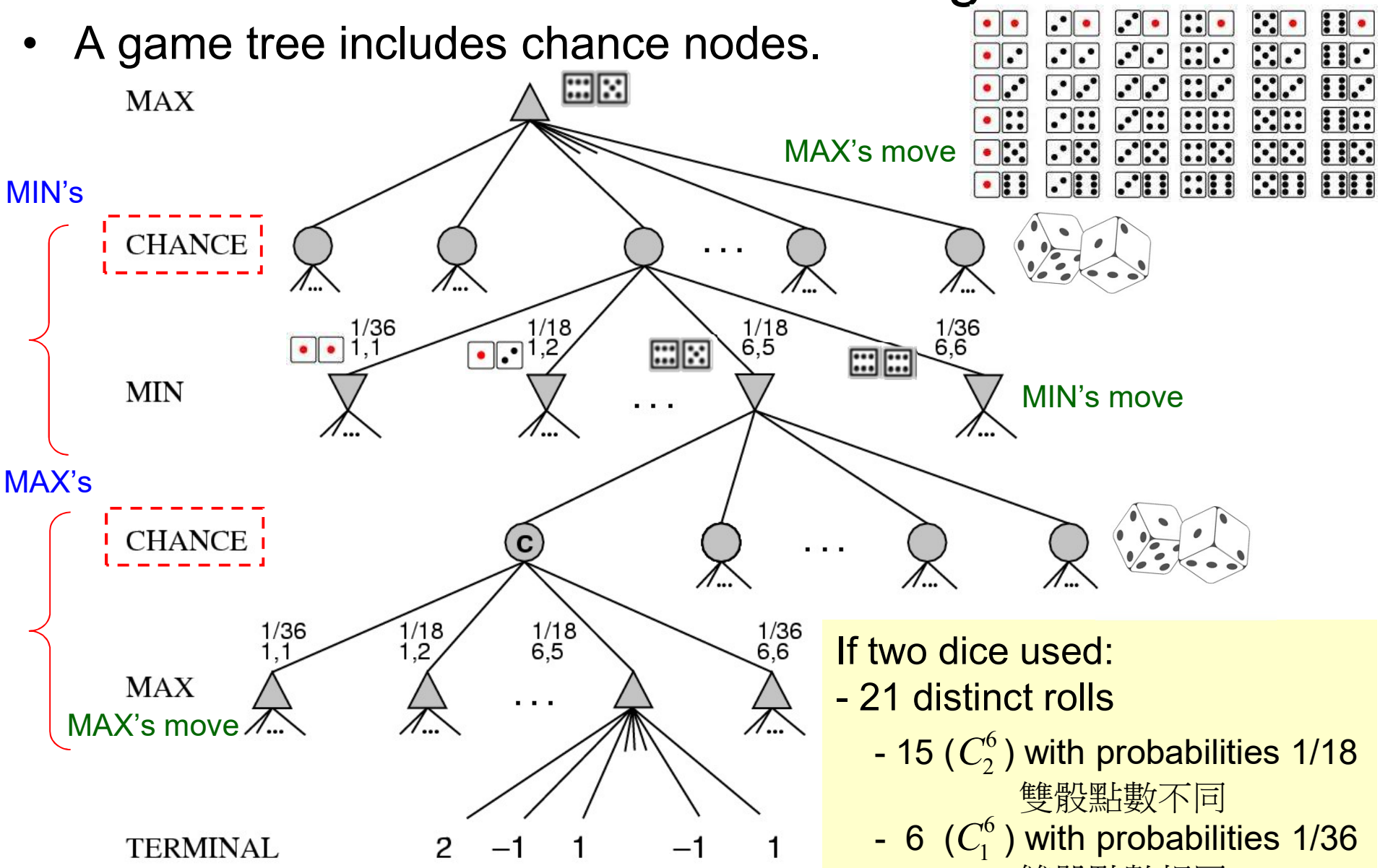
…

When white has rolled 6-5, it must choose among four legal moves:

(5 → 10吃子, 5 → 11) (5 → 11, 19 → 24)
(5 → 10吃子, 10 → 16) (5 → 11, 11 → 16)

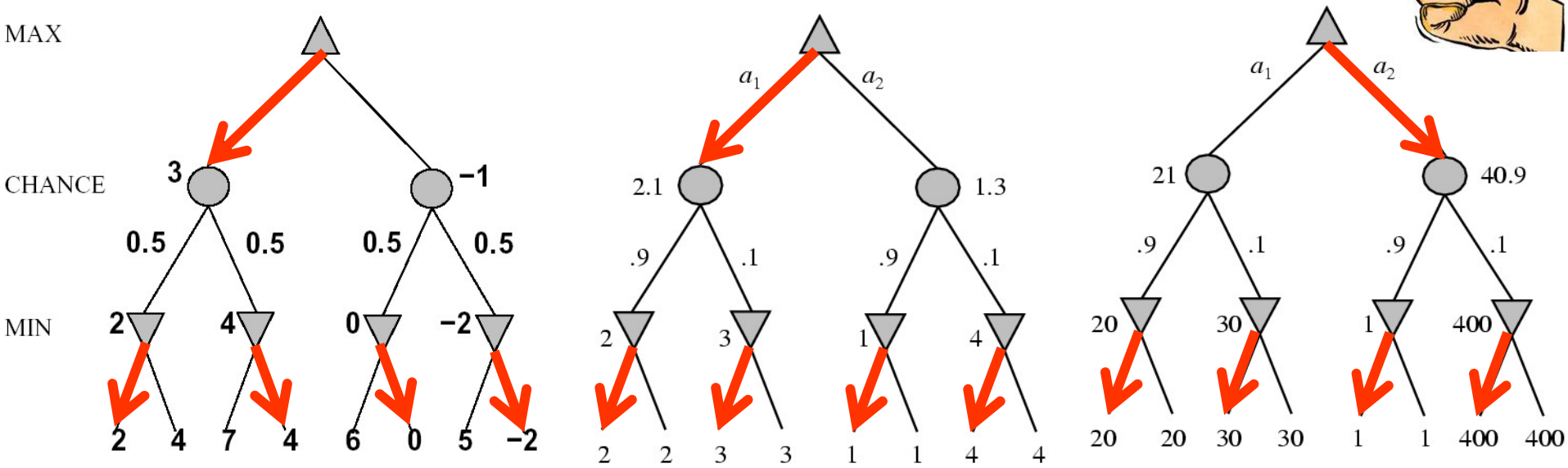
Nondeterministic Games: Backgammon

- A game tree includes chance nodes.



If two dice used:
- 21 distinct rolls
  - 15 (C_2^6) with probabilities 1/18
    雙骰點數不同
  - 6 (C_1^6) with probabilities 1/36
    雙骰點數相同

- Chance introduced by dice, card-shuffling(洗牌)
  - e.g., a simplified example with coin-flipping



Algorithm for Nondeterministic Games: gives perfect play

Just like minimax, except chance nodes must be also handled.

$$\text{expectiminimax}(n) = \begin{cases} \text{Utility}(n) & \text{if } n \text{ is a terminal state} \\ \max_{s \in \text{Successor}(n)} \text{expectiminimax}(s) & \text{if } n \text{ is a MAX node} \\ \min_{s \in \text{Successor}(n)} \text{expectiminimax}(s) & \text{if } n \text{ is a MIN node} \\ \sum_{s \in \text{Successor}(n)} P(s) \cdot \text{expectiminimax}(s) & \text{if } n \text{ is a chance node} \end{cases}$$