第二章 資料結構

第一節 盤面的表示方法

基本的盤面表示方法有許多種，最直覺的是以二維陣列表示，一個基本的棋盤(圖 2-1)，可以棋盤左上角爲(0,0)，棋盤右下角爲(9,8)表示，要得知棋子是在哪一行哪一列直接查縱座標或橫座標即可得知，如表 2-1。

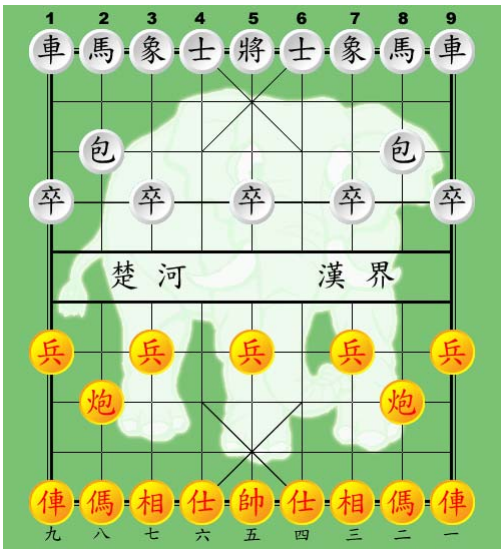


圖 2-1 棋盤圖

表 2-1 二維陣列表示法

(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	(0,8)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)	(1,6)	(1,7)	(1,8)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)	(2,6)	(2,7)	(2,8)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)	(3,6)	(3,7)	(3,8)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)	(4,6)	(4,7)	(4,8)
(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)	(5,6)	(5,7)	(5,8)
(6,0)	(6,1)	(6,2)	(6,3)	(6,4)	(6,5)	(6,6)	(6,7)	(6,8)
(7,0)	(7,1)	(7,2)	(7,3)	(7,4)	(7,5)	(7,6)	(7,7)	(7,8)
(8,0)	(8,1)	(8,2)	(8,3)	(8,4)	(8,5)	(8,6)	(8,7)	(8,8)
(9,0)	(9,1)	(9,2)	(9,3)	(9,4)	(9,5)	(9,6)	(9,7)	(9,8)

除了上述二維陣列的表示方式之外，也可以使用一維陣列來表示棋盤。另外，在產生合法走子步時，爲了檢查棋子是否會超出棋盤，在棋盤的上方及左側加入邊界區域，以方便判斷，如表 2-2 所示，邊界區域以灰底標明。

比較兩種方式，在搜尋及審局函數中，大量存取棋盤資料結構時，採用一維陣列 board[BOARD_SIZE] 的表示方式少了一次 address offset 存取的動作，會比二維陣列表示法來得快。

表 2-2 一維陣列表示法

0	1	2	3	4	5	6	7	8	9
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

當要產生棋子在棋盤上某一位置 N 的走法時，往上移動一格的位置是 N-10，往下移動一格的位置是 N+10，往左一格是 N-1，往右一格則是 N+1。

有了可以表示棋盤上不同位置的一維陣列之後，爲了表示不同的棋子，我們必須對每個棋子加以編號，其中空格和邊界也要給予不同的號碼表示以茲識別，加上 32 個棋子，所以共用了 34 個號碼來表示棋盤上不同的棋子，如表 2-3 所示。

表 2-3 棋子編號表

空格	紅俥		紅傌		紅炮		紅兵	
0	1	2	3	4	5	6	7	8
紅兵			紅相		紅仕		紅帥	
9	10	11	12	13	14	15	16	
邊界	黑車		黑馬		黑包		黑卒	
17	18	19	20	21	22	23	24	25
黑卒			黑象		黑士		黑將	
26	27	28	29	30	31	32	33	

利用一維陣列 board[BOARD_SIZE]可以得知棋盤上每一個位置被哪個棋子所佔據，但是當產生合法走子步時，必須掃描棋盤上每個位置一次，找到每個棋子所在的位置，浪費許多時間，所以額外維護一個一維陣列 piece_table[34] ，用來表示每個棋子目前位在棋盤上哪個位置，如果 piece_table[K]爲零，則代表棋子 K 不在棋盤上。例如棋盤上 67 的位置是黑方第一隻車(編號爲 18)，那麼 board[67] 紀錄的就是 18，而 piece_table[18] 紀錄的則是 67。如此一來，當產生合法走子步時，只需查詢 piece_table[] 陣列，針對還在盤面上的棋子產生合法步即可。

第二節　走法生成資料結構

根據上一節所提到的盤面表示方法，就可以產生棋子的走法了。首先以一個 16 位元的整數代表每個不同的走法，其中 8 個位元代表棋子移動的出發點，另外

8 個位元代表棋子移動的目的地，當我們要產生某個走法時，只要利用簡單的邏輯運算 SHIFT、AND 及 OR 就可得知，反之如果要從某個走法得知棋子移動的出發點和目的地，只要作一次反運算即可。圖 2-2 所示是從棋盤上座標 11 移動到 15 的例子。

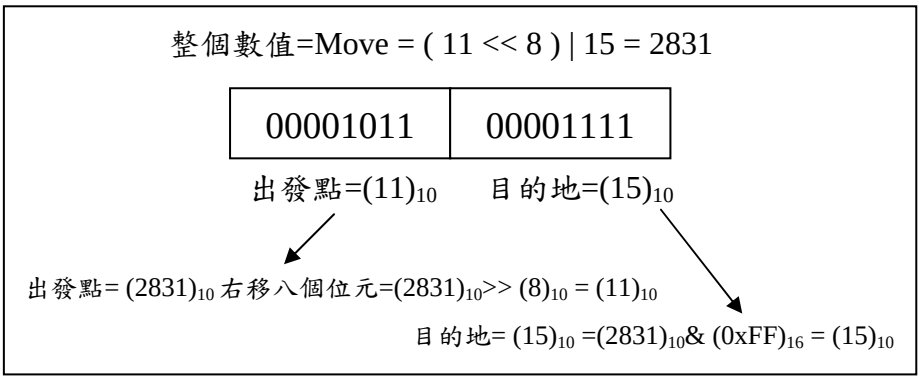


圖 2-2 走法編碼

這樣只能表達一種特定的走法，一般來說，在某個盤面情況下，不管是輪紅方還是黑方走子，總會有超過一種以上的走法，所以需要另一種資料結構，來表達某個盤面下所有可能的走法。既然走法的表示是一個整數，所以另外使用一個一維陣列來紀錄某盤面下所有可能的走法即可。而爲了便於在產生所有走法時，先進行簡單的排序，我們將走法粗略分爲吃子步及非吃子步，所謂的吃子步，即進行此走法後可吃到敵方的棋子，非吃子步則不會吃到敵方的棋子，產生走法時會先產生吃子步，我們將其放在一維陣列的前半部份，而後半部份則存放非吃子步，如此便能產生最簡單的走法排序效果，如圖 2-3 所示。

圖 2-3 走法陣列

爲了考慮走法陣列所需配置的大小，必須要分別考慮最多的吃子步及非吃子步數目。以中國象棋來說，對每種棋子的最多可能走法列表，可得到如表 2-4 所示的結果。

表 2-4 最多走法數目列表

棋子	俥	傌	炮	兵	相	仕	帥	總和
最多走法數量	17 x 2	8 x 2	17 x 2	3 x 5	4+2	4	4	113

理論上最多可能走法有 113 種，但實際上同時考慮己方棋子可能佔據另一個棋子要走的位置，所以實際上最多走法將少於 113 種。例如仕和帥，兩者因爲會互相佔據九宮內彼此走子的位置，所以仕和帥最多可能走法總和只有 6 種，而不是上表所列 8 種。目前已知最多走法的盤面如圖 2-4，紅方共有 110 種走法[25]。

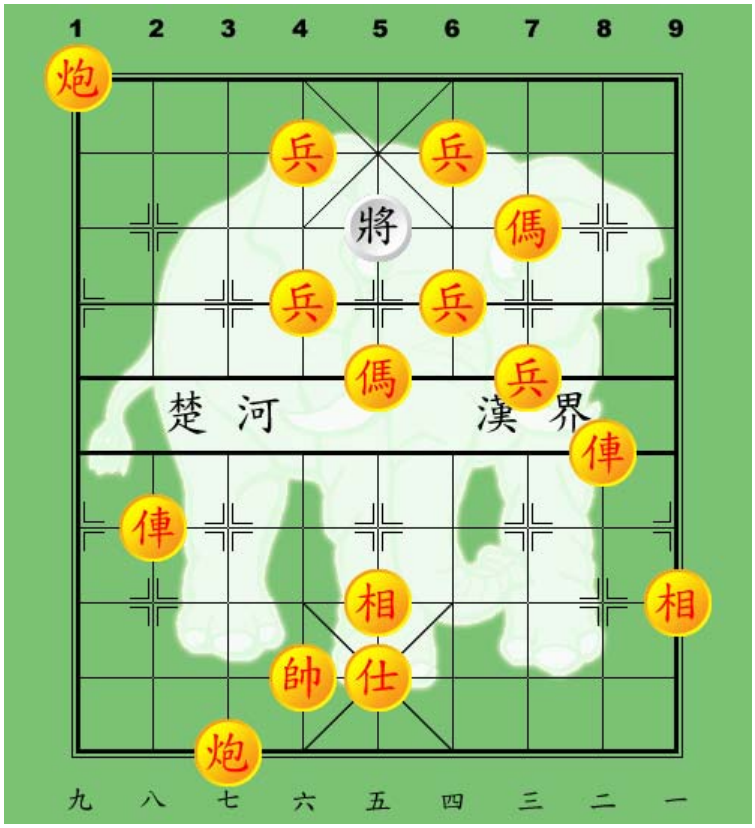


圖 2-4 已知最多走法情況

另外，考慮最多可能的吃子走法，列表如表 2-5。

表 2-5 最多吃子步數目列表

棋子	俥	傌	炮	兵	相	仕	帥	總和
最多吃子走法數量	4 x 2	8 x 2	4 x 2	3 x 5	4+2	4	4	61

由上表可知，理論上的最多吃子走法達 61 種之多，但實際上受限於被吃棋子的數目與棋子所在位置，一般在實戰中出現的最多吃子步數目遠小於 61。目前已知最多吃子步走法數目爲 40 種[25]，如圖 2-5 所示。

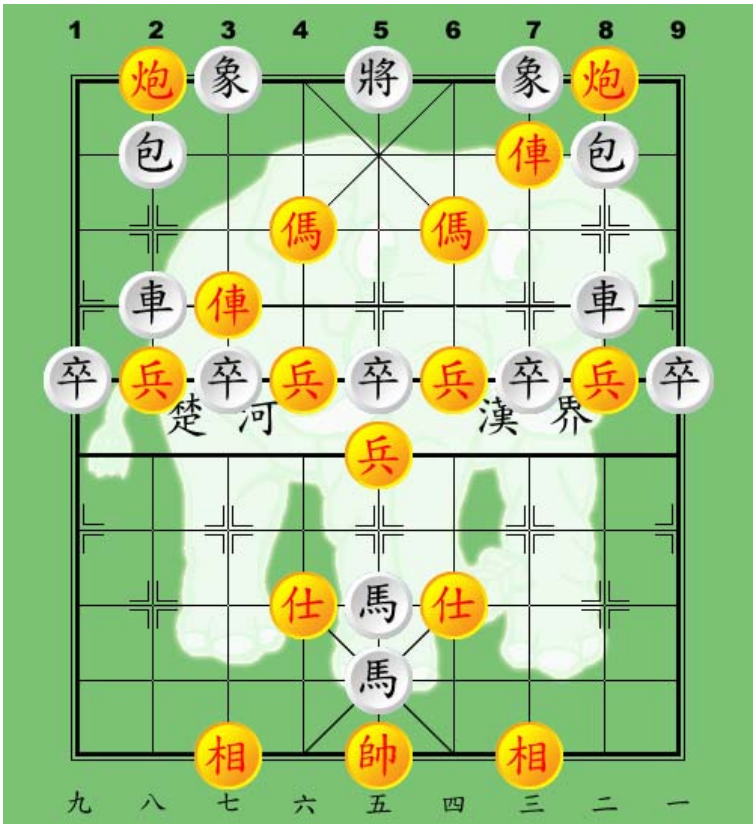
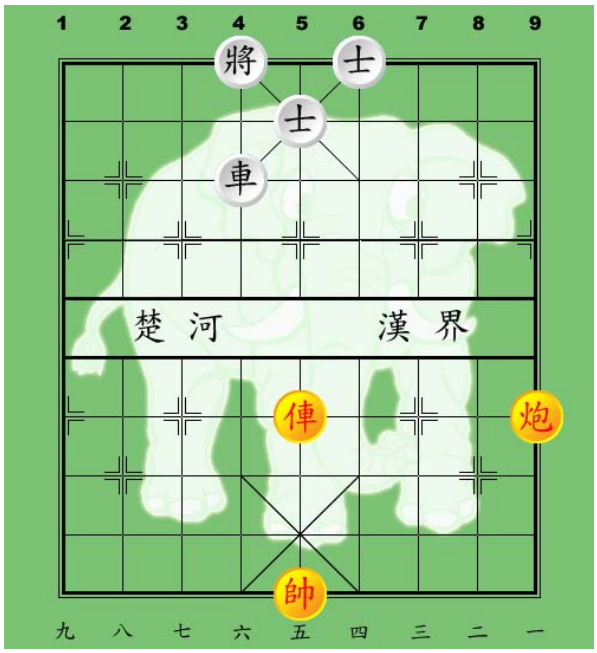


圖 2-5 已知最多吃子步情況

綜合上述，我們將走法陣列的大小定爲 170，其中保留前半部份的 60 個走法供吃子步存放。

第三節　dirnext 資料結構

爲了加快王見王檢查、俥炮在將軍檢測、審局方面的速度，另外在搜尋的過程中維護 dirnext 資料結構。dirnext[K][D]爲二維陣列，代表了每一個棋子上下左右距離最近的是哪些棋子，其中 K 代表棋子編號，D 爲 {0, 1, 2, 3}其中之一，分別代表右、下、左、上四個方向，以圖 2-6 爲例。



紅俥編號為 1
dirnext[1][0] = 79
(紅俥右邊的炮之位置)
dirnext[1][1] = 105
(紅俥下面的帥之位置)
dirnext[1][2] = 70
(紅俥左邊的邊界之位置)
dirnext[1][3] = 25
(紅俥上面的士之位置)

圖 2-6 dirnext 示例圖

表 2-6 是使用 dirnext 資料結構來作王見王檢測，以及一般的檢查方式比較。

表 2-6 使用 dirnext 檢查王見王虛擬碼

一般檢查方式	使用 dirnext

pos = piece_table[RedKing] - 10
while (board[pos] is BLANK)
　　pos = pos – 10
if (board[pos] is BlackKing)
　　return 1
return 0
	pos = dirnext[RedKing][UP]
	if (board[pos] is BlackKing)
	　　return 1
	return 0

表 2-7 是使用和不使用 dirnext 資料結構的平均搜尋耗費時間比較，測試使用的電腦 CPU 爲 AMD Opteron 275，記憶體 2GB。

表 2-7 dirnext 效能比較表

搜尋深度	不使用 dirnext 花費時間	使用 dirnext 花費時間	加快幅度
10	9.06 秒	8.62 秒	4.8%
11	18.26 秒	17.56 秒	3.8%
12	36.2 秒	34.12 秒	5.7%