Homework assignment#3 (Chap5)

106971001 林上人

(A) Pseudo codes documentation

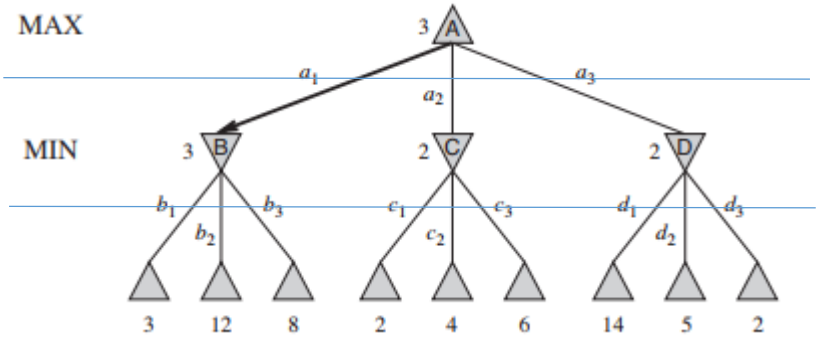
Pages: 19

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

function MAX-VALUE(state) 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)))
  return v

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

① ② ③

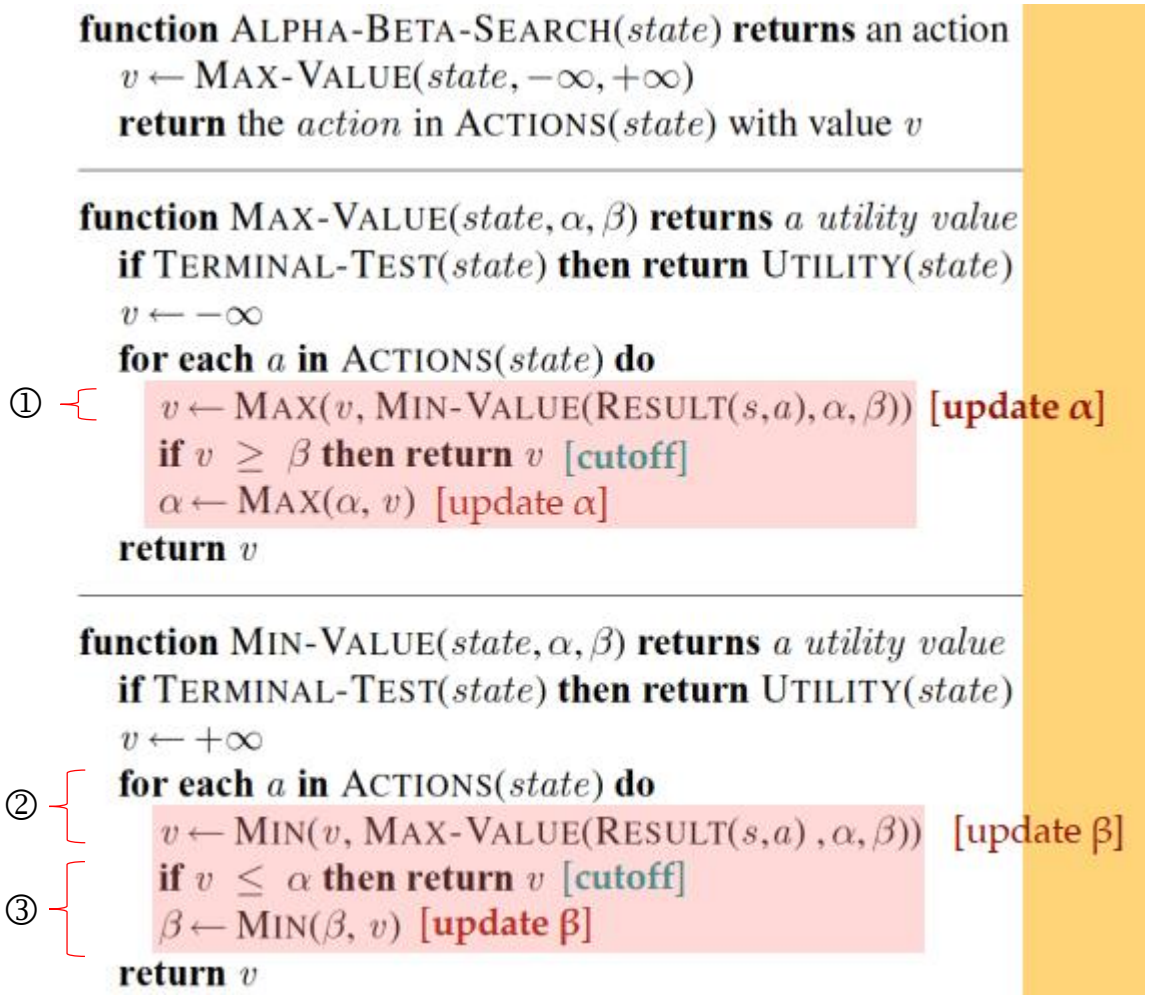
Minimax 演算法 – 在最壞情況中謀求最大利益

此演算法假設雙人對局的情形，且對方一定會選擇讓我方情況最差的動作，因此通常情形會如下圖所示，一層為 MAX 一層為 MIN 交替產生，MAX 表



示我們要從後面取得最大值，MIN 表示對方要從後面取得最小值的情形，①的部分為 root 表示我們要得到能夠從 MIN-VALUE 函數產生 max 值的動作參數，後續②、③先檢查狀態是否結束，例如遊戲以分出勝負或是其他狀況，然後根據所在的層，②MAX-VALUE 函數就從後續的 MIN-VALUE 回傳的值回傳最大值，③MIN-VALUE 函數就從後續的 MAX-VALUE 回傳的值回傳最小值。

Pages: 33



ALPHA-BETA-SEARCH

此處是改善原本的 MINMAX 演算法，從而不需將每一個節點都拓展，以達到優化的目的，主要是在 MINIMAX 中加入 α 、 β 做紀錄，作為是否 pruning 的標準，其中 α 代表到目前為止極大值的最佳解， β 代表到目前為止極小值的最佳解，以 MAX 層來看①因為在 MAX 層時會從下一層的 MIN-VALUE 回傳的值中取最大值，所以②下一層的 MIN-VALUE 在計算每一個 MAX-VALUE(RESULT(s,a))時，③若有出現比 α 值低的就表示在這一層(MIN 層)的這一個分支回傳值會小於 α ，而這個分支回到 MAX 層時就不會被選中，因此可以不用繼續檢查其他 MAX-VALUE(RESULT(s,a))直接回傳 υ ，以 MIN 層來看道理相同，只是改成檢查 β 值，並且在 MAX 節點更新 α 值，在 MIN 節點更新 β 值。

(B) Exercise

5.8 Consider the two-player game described in Figure 5.17.

a. Draw the complete game tree, using the following conventions:

- Write each state as (s_A, s_B), where s_A and s_B denote the token locations.
- Put each terminal state in a square box and write its game value in a circle.
- Put *loop states* (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.

b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.

c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?

d. This 4-square game can be generalized to n squares for any $n > 2$. Prove that A wins if n is even and loses if n is odd.

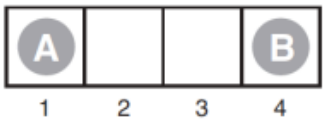
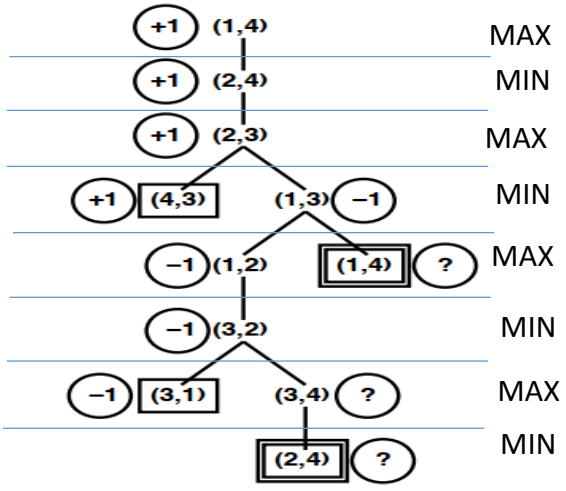


Figure 5.17 The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is $+1$; if player B reaches space 1 first, then the value of the game to A is -1.

(a)



(b)

The "?" values are handled by assuming that an agent with a choice between winning the game and entering a "?" state will always choose the win. That is, min(–1,?) is –1 and max(+1,?) is +1. If all successors are "?", the backed-up value is "?".

(c)

Standard minimax is depth-first and would go into an infinite loop. It can be fixed by comparing the current state against the stack; and if the state is repeated, then return a "?" value.

In this example, both (1,4) and (2,4) repeat in the tree but they are won positions. Although it works in this case, it does not *always* work because it is not clear how to compare "?" with a drawn position; nor is it clear how to handle the comparison when there are wins of different degrees.

(d)

The base case n=3 is a loss for A and the base case n=4 is a win for A.

For any n > 4, the initial moves are the same: A and B both move one step towards each other.

Now, we can see that they are engaged in a subgame of size n − 2 on the squares [2, . . . , n − 1], except that there is an extra choice of moves on squares 2 and n − 1. Ignoring this for amoment, it is clear that if the "n − 2" is won for A, then A gets to the square n − 1 before B gets to square 2 (by the definition of winning) and therefore gets to n before B gets to 1, hence the "n" game is won for A.

Now, the presence of the extra moves complicates the issue, but not too much. First, the player who is slated to win the subgame [2, . . . , n − 1] never moves back to his home square. If the player slated to lose the subgame does so, then it is easy to show that he is bound to lose the game itself—the other player simply moves forward and a subgame of size n − 2k is played one step closer to the loser's home square.

5.10 Consider the family of generalized tic-tac-toe games, defined as follows. Each particular game is specified by a set $\mathcal{S}$ of *squares* and a collection $\mathcal{W}$ of *winning positions*. Each winning position is a subset of $\mathcal{S}$. For example, in standard tic-tac-toe, $\mathcal{S}$ is a set of 9 squares and $\mathcal{W}$ is a collection of 8 subsets of $\mathcal{W}$: the three rows, the three columns, and the two diagonals. In other respects, the game is identical to standard tic-tac-toe. Starting from an empty board, players alternate placing their marks on an empty square. A player who marks every square in a winning position wins the game. It is a tie if all squares are marked and neither player has won.

a. Let $N = |\mathcal{S}|$, the number of squares. Give an upper bound on the number of nodes in the complete game tree for generalized tic-tac-toe as a function of N.

b. Give a lower bound on the size of the game tree for the worst case, where $\mathcal{W} = \{\}$.

c. Propose a plausible evaluation function that can be used for any instance of generalized tic-tac-toe. The function may depend on $\mathcal{S}$ and $\mathcal{W}$.

d. Assume that it is possible to generate a new board and check whether it is a winning position in $100N$ machine instructions and assume a 2 gigahertz processor. Ignore memory limitations. Using your estimate in (a), roughly how large a game tree can be completely solved by alpha–beta in a second of CPU time? a minute? an hour?

(a)

Upper bound : N!

One for each ordering of squares, so an upper bound on the total number of nodes is $\sum_{i=1}^{N} i!$. This is not much bigger than N! itself as the factorial function grows super-exponentiallly. This is an overestimate because some games will end early when a winning position is filled.

(b)

In this case no games terminate early, and there are N! different games ending in a draw.

So ignoring repeated states, we have $\sum_{i=1}^{N} i!$ nodes.

At the end of the game the squares are divided between the two players: ⌈N/2⌉ to the first player and ⌊N/2⌋ to the second. Thus, a good lower bound on the number of distinct states is $\binom{N}{\lceil N/2 \rceil}$,the number of distinct terminal states.

(c)

For a state s, let X(s) be the number of winning positions containing no O's and O(s) the number of winning positions containing no X's.

One evaluation function is then Eval(s) = X(s) − O(S). Notice that empty winning positions cancel out in the evaluation function. Alternatively, we might weight potential winning positions by how close they are to completion.

(d)

Using the upper bound of N! from (a), and observing that it takes 100NN! instructions. At 2GHz we have 2 billion instructions per second, so solve for the largest N using at most this many instructions. For one second we get N = 9, for one minute N = 11, and for one hour N = 12.

5.13 Develop a formal proof of correctness for alpha–beta pruning. To do this, consider the situation shown in Figure 5.18. The question is whether to prune node n_j, which is a max-node and a descendant of node n_1. The basic idea is to prune it if and only if the minimax value of n_1 can be shown to be independent of the value of n_j.

a. Mode n_1 takes on the minimum value among its children: $n_1 = \min(n_2, n_{21}, \ldots, n_{2b_2})$. Find a similar expression for n_2 and hence an expression for n_1 in terms of n_j.

b. Let l_i be the minimum (or maximum) value of the nodes to the *left* of node n_i at depth i, whose minimax value is already known. Similarly, let r_i be the minimum (or maximum) value of the unexplored nodes to the right of n_i at depth i. Rewrite your expression for n_1 in terms of the l_i and r_i values.

c. Now reformulate the expression to show that in order to affect n_1, n_j must not exceed a certain bound derived from the l_i values.

d. Repeat the process for the case where n_j is a min-node.

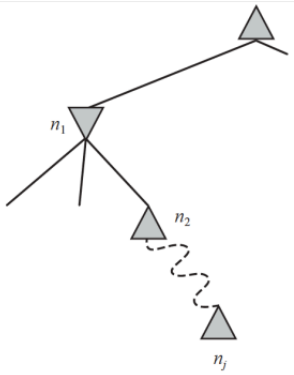


Figure 5.18 Situation when considering whether to prune node n_j.

(a)

n_2 = max(n_3, n_{31}, . . . , n_{3b3}), giving

n_1 = min(max(n_3, n_{31}, . . . , n_{3b3}), n_{21}, . . . , n_{2b2})

Then n_3 can be similarly replaced, until we have an expression containing n_j itself.

(b)

n_1 = min(l_2,max(l_3, n_3, r_3), r_2)

n_3 can be expanded out down to n_j. The most deeply nested term will be $\min(l_j, n_j, r_j)$.

(c)

If n_j is a max node, then the lower bound on its value only increases as its successors are evaluated. Clearly, if it exceeds l_j it will have no further effect on n_1. By extension, if it exceeds $\min(l_2, l_4, \ldots, l_j)$ it will have no effect. Thus, by keeping track of this value we can decide when to prune n_j.

(d)

The corresponding bound form in nodes n_k is $\max(l_3, l_5, \ldots, l_k)$.

5.16 This question considers pruning in games with chance nodes. Figure 5.19 shows the complete game tree for a trivial game. Assume that the leaf nodes are to be evaluated in left-to-right order, and that before a leaf node is evaluated, we know nothing about its value—the range of possible values is $-\infty$ to ∞.

a. Copy the figure, mark the value of all the internal nodes, and indicate the best move at the root with an arrow.

b. Given the values of the first six leaves, do we need to evaluate the seventh and eighth leaves? Given the values of the first seven leaves, do we need to evaluate the eighth leaf? Explain your answers.

c. Suppose the leaf node values are known to lie between –2 and 2 inclusive. After the first two leaves are evaluated, what is the value range for the left-hand chance node?

d. Circle all the leaves that need not be evaluated under the assumption in (c).

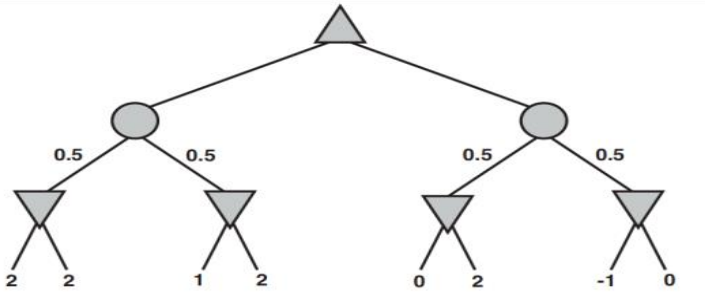
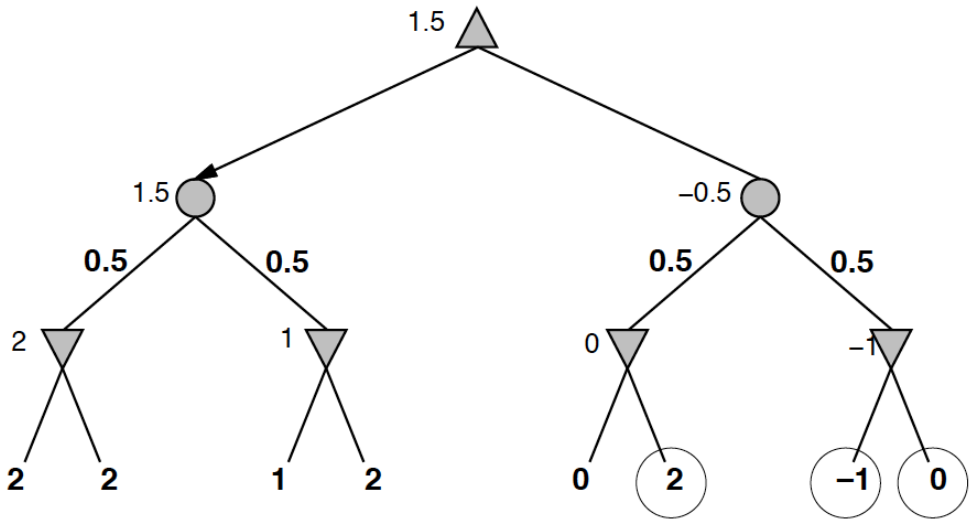


Figure 5.19 The complete game tree for a trivial game with chance nodes.

(a) & (d)



(b)

Given nodes 1 – 6, we would need to look at 7 and 8: if they were both $+\infty$ then the values of the min node and chance node above would also be $+\infty$ and the best move would change. Given nodes 1 – 7, we do not need to look at 8. Even if it is $+\infty$, them in node cannot be worth more than -1, so the chance node above cannot be worth more than -0.5, so the best move won't change.

(c)

The worst case is if either of the third and fourth leaves is −2, in which case the chance node above is 0. The best case is where they are both 2, then the chance node has value 2. So it must lie between 0 and 2.

5.20 In the following, a "max" tree consists only of max nodes, whereas an "expectimax" tree consists of a max node at the root with alternating layers of chance and max nodes. At chance nodes, all outcome probabilities are nonzero. The goal is to *find the value of the root* with a bounded-depth search.

a. Assuming that leaf values are finite but unbounded, is pruning (as in alpha–beta) ever possible in a max tree? Give an example, or explain why not.

b. Is pruning ever possible in an expectimax tree under the same conditions? Give an example, or explain why not.

c. If leaf values are constrained to be in the range $[0, 1]$, is pruning ever possible in a max tree? Give an example, or explain why not.

d. If leaf values are constrained to be in the range $[0, 1]$, is pruning ever possible in an expectimax tree? Give an example (qualitatively different from your example in (e), if any), or explain why not.

e. If leaf values are constrained to be nonnegative, is pruning ever possible in a max tree? Give an example, or explain why not.

f. If leaf values are constrained to be nonnegative, is pruning ever possible in an expectimax tree? Give an example, or explain why not.

g. Consider the outcomes of a chance node in an expectimax tree. Which of the following evaluation orders is most likely to yield pruning opportunities: (i) Lowest probability first; (ii) Highest probability first; (iii) Doesn't make any difference?

(a)

No pruning. In a max tree, the value of the root is the value of the best leaf. Any unseen leaf might be the best, so we have to see them all.

(b)

No pruning. An unseen leaf might have a value arbitrarily higher or lower than any other leaf, which (assuming non-zero outcome probabilities) means that there is no bound on the value of any incompletely expanded chance or max node.

(c)

No pruning.same as (a)

(d)

No pruning. Nonnegative values allow lower bounds on the values of chance nodes, but a lower bound does not allow any pruning.

(e)

Yes. If the first successor has value 1, the root has value 1 and all remaining successors can be pruned.

(f)

Yes. Suppose the first action at the root has value 0.6, and the first outcome of the second action has probability 0.5 and value 0; then all other outcomes of the second action can be pruned.

(g)

(ii) Highest probability first. This gives the strongest bound on the value of the node, all other things being equal.

5.21 Which of the following are true and which are false? Give brief explanations.

a. In a fully observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what strategy the second player is using—that is, what move the second player will make, given the first player's move.

b. In a partially observable, turn-taking, zero-sum game between two perfectly rational players, it does not help the first player to know what move the second player will make, given the first player's move.

c. A perfectly rational backgammon agent never loses.

(a)

True. The second player will play optimally, and so is perfectly predictable up to ties. Knowing which of two equally good moves the opponent will make does not change the value of the game to the first player.

(b)

False. By knowing the second player's strategy still can increase the first player's chances of winning.

(c)

False. Backgammon is game of chance, and the opponent may consistently roll much better dice.