

Adversarial Search

YAO ZHAO

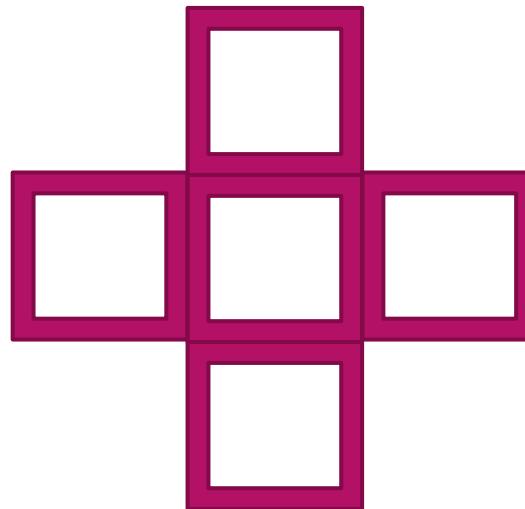
Minimax Algorithm

- ▶ **Minimax Algorithm** is a basic method to solve game-tree problems.

```
15 ▼ def minimax_decision(state, game):
16 ▼     """Given a state in a game, calculate the best move by searching
17         forward all the way to the terminal states. [Figure 5.3]"""
18
19     player = game.to_move(state)
20
21 ▼     def max_value(state):
22 ▼         if game.terminal_test(state):
23             return game.utility(state, player)
24         v = -infinity
25 ▼         for a in game.actions(state):
26             v = max(v, min_value(game.result(state, a)))
27         return v
28
29 ▼     def min_value(state):
30 ▼         if game.terminal_test(state):
31             return game.utility(state, player)
32         v = infinity
33 ▼         for a in game.actions(state):
34             v = min(v, max_value(game.result(state, a)))
35         return v
36
37     # Body of minimax_decision:
38     return argmax(game.actions(state),
39                   key=lambda a: min_value(game.result(state, a)))
```

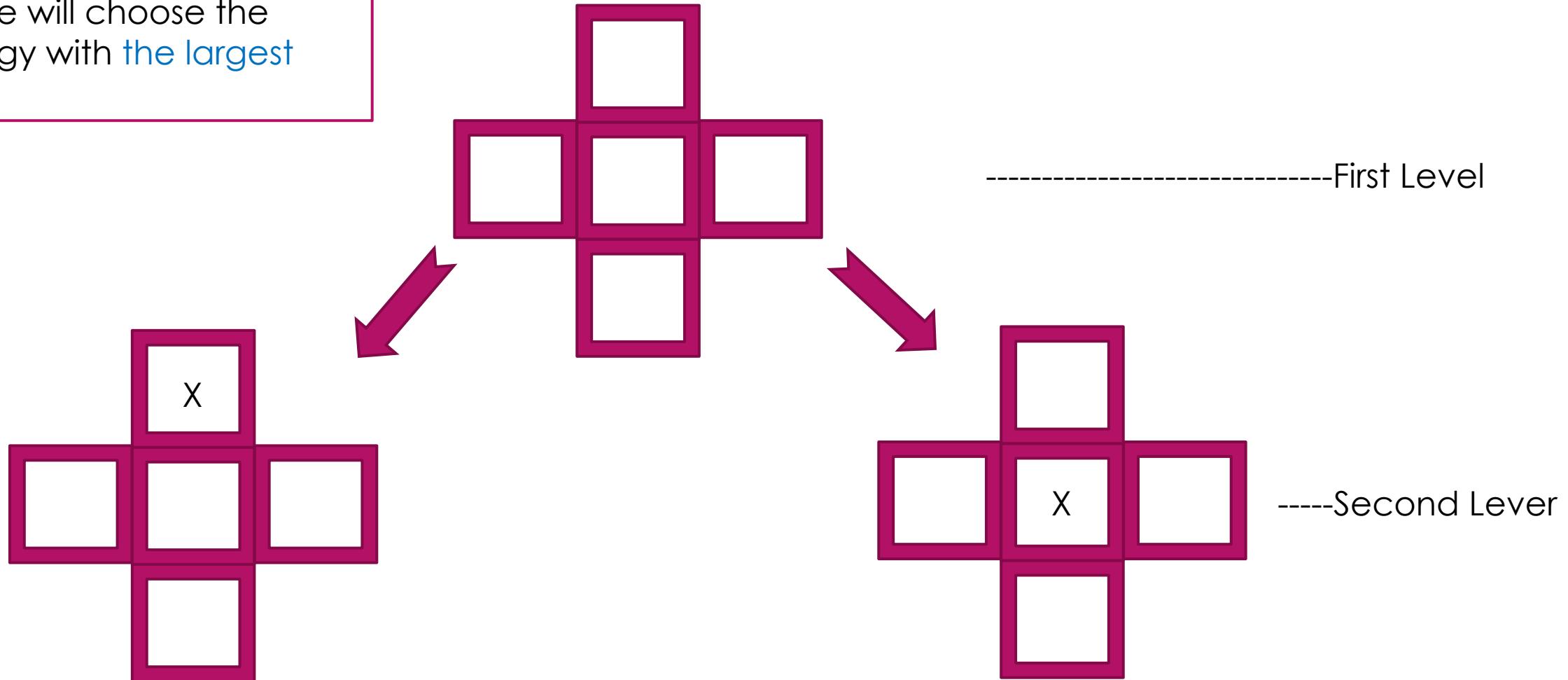
A Basic Example

- ▶ Fill X or O in the box
- ▶ X plays first
- ▶ Termination:
 - If there are two consecutive X or O, then it wins
- ▶ Utility function:
 - If X wins, score=1
 - If O wins, score=-1
 - If tie, score=0



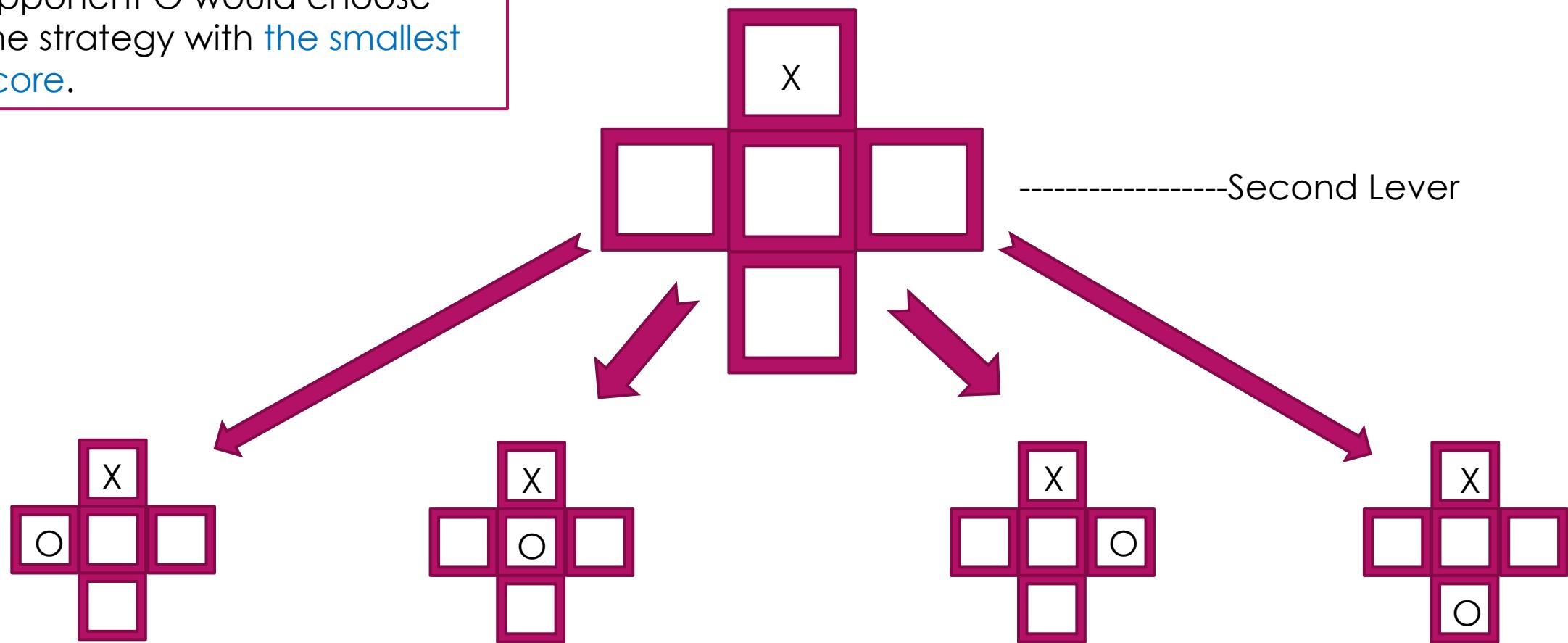
A Basic Example: The First Level with Max

MAX(): If X wants to win, he/she will choose the strategy with **the largest score**

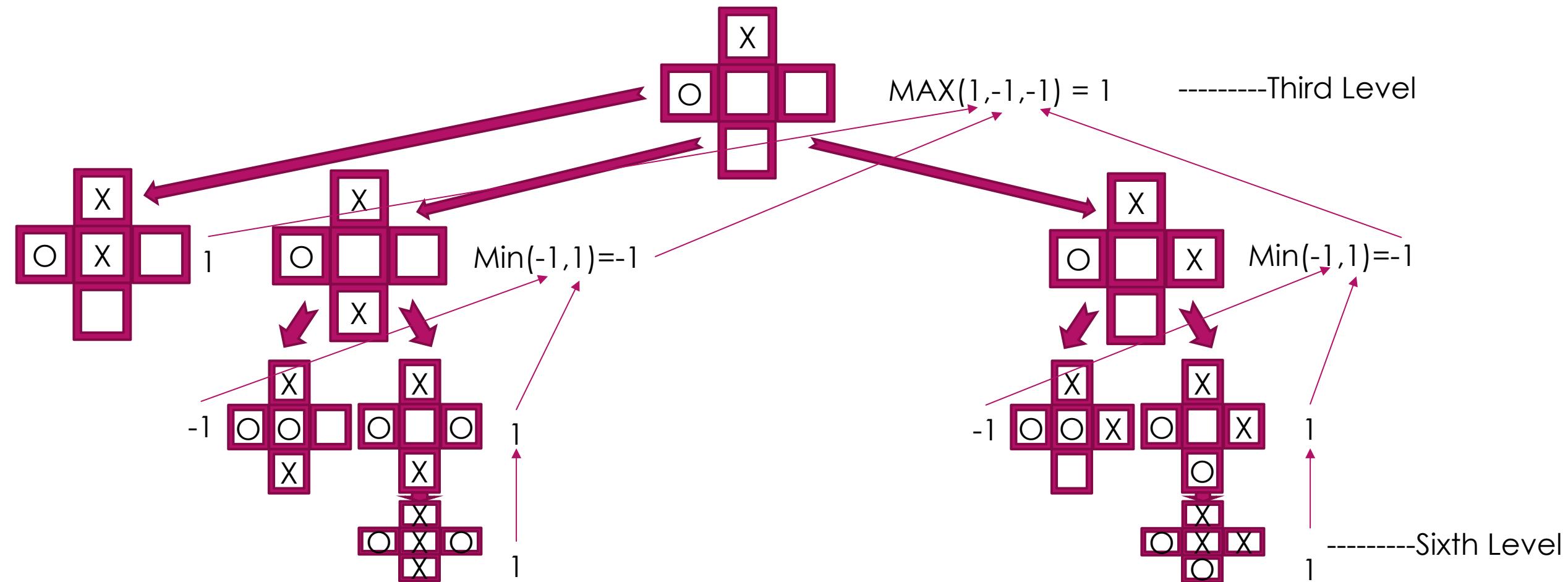


A Basic Example: The Second Level with Min

MIN(): If X has chosen the center-above position, the opponent O would choose the strategy with **the smallest score**.

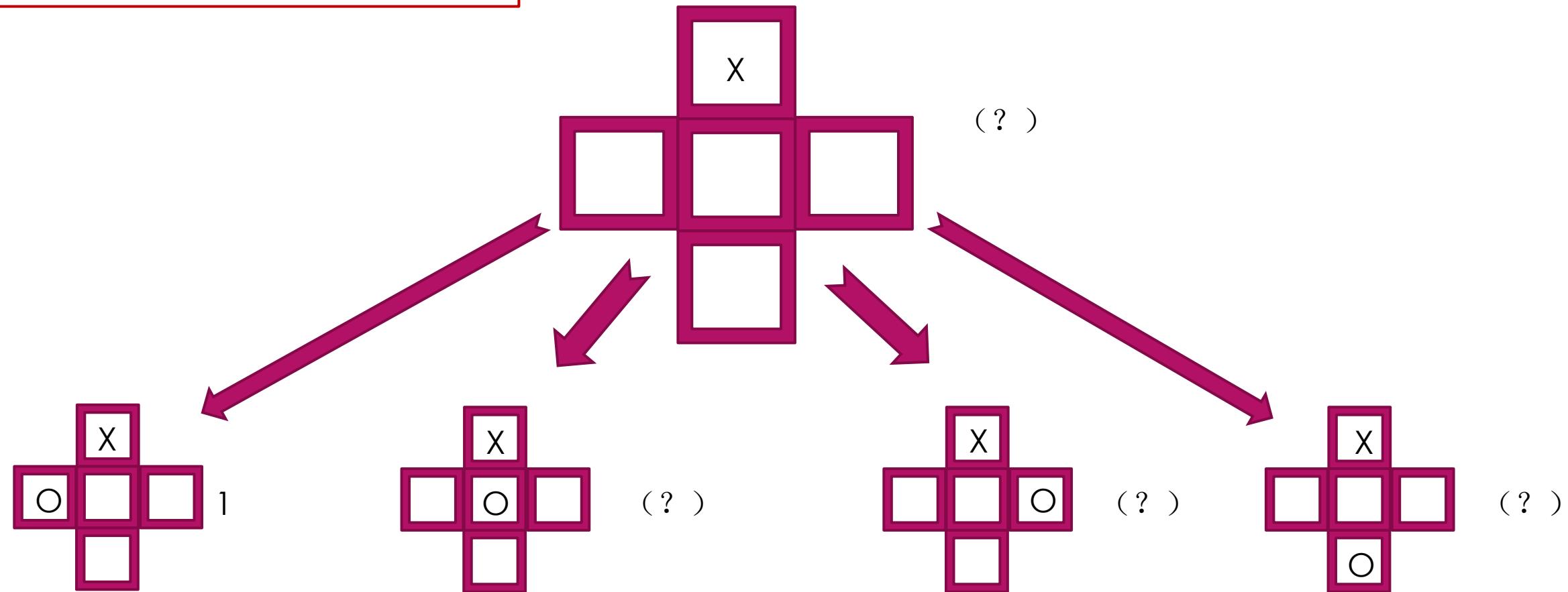


A Basic Example: Third Level to Sixth Level

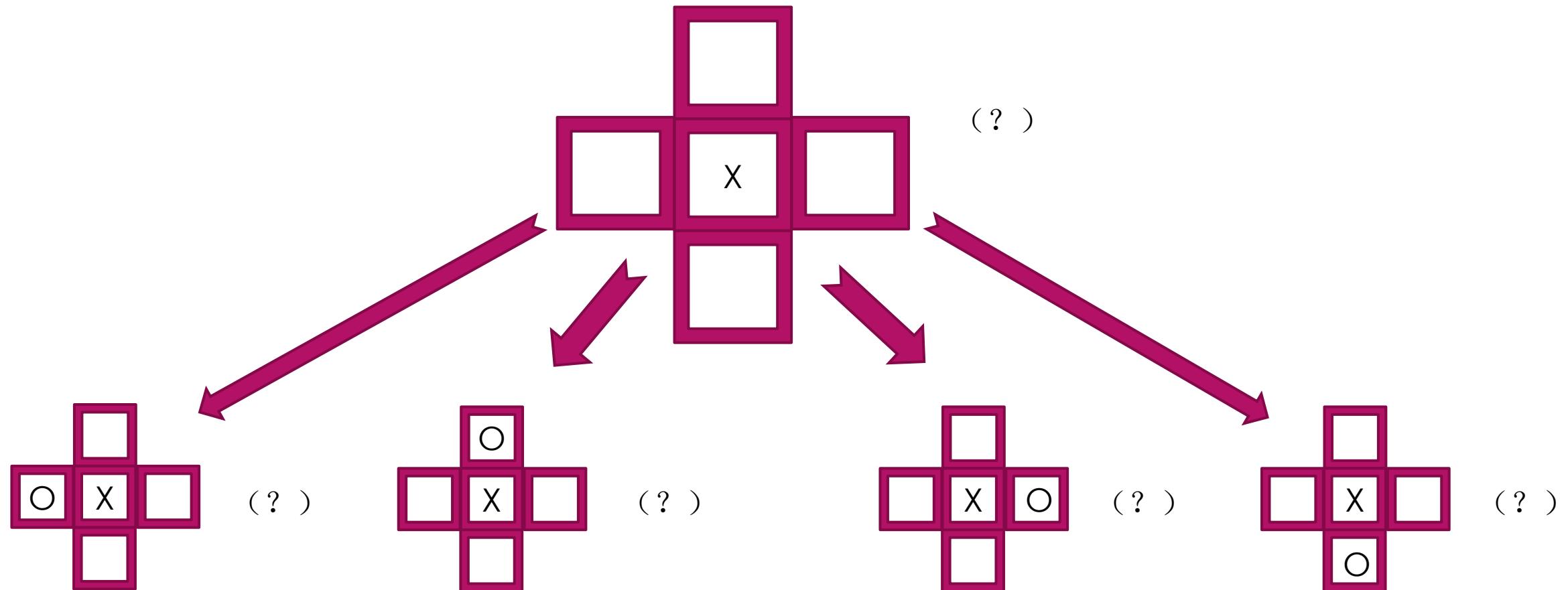


Fill in the Blank

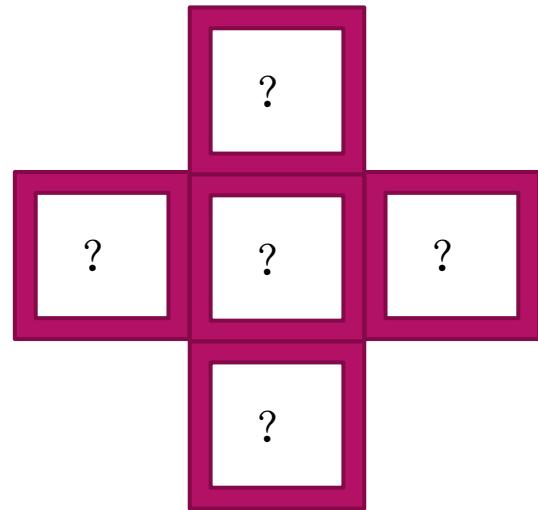
Please complete the scores in the remaining boxes according to the above procedure



Fill in the Blank



Which position would X choose in the first step?

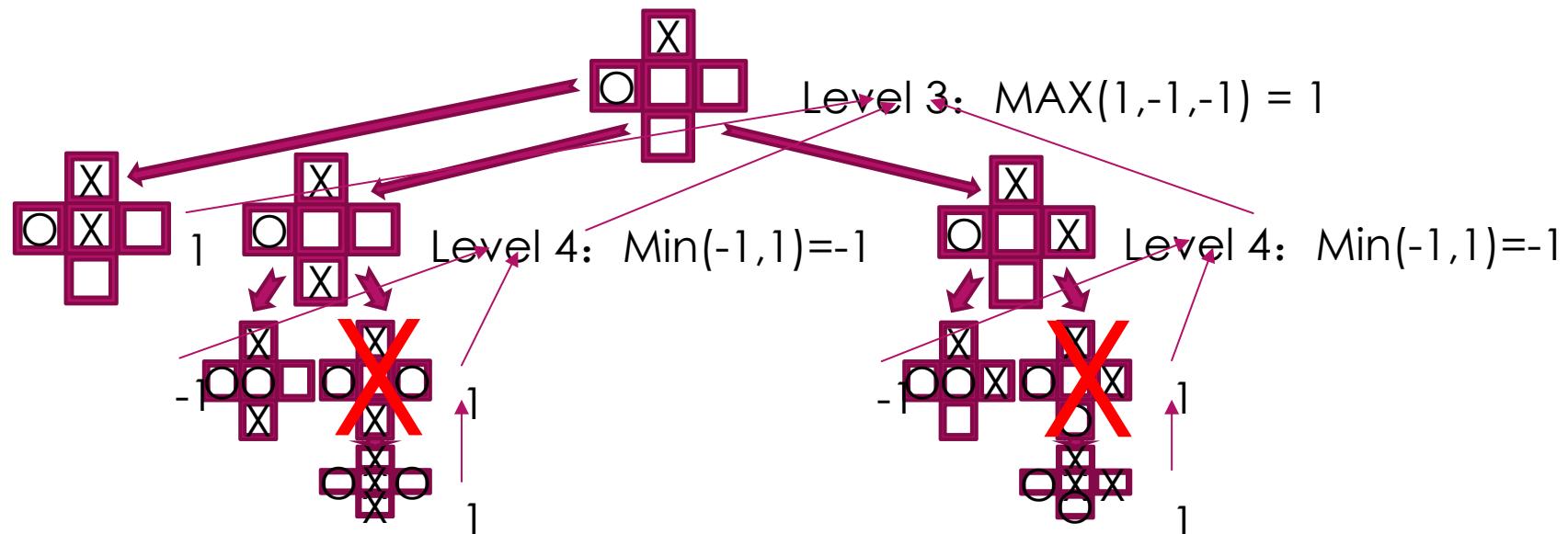


In the implementation of the Minimax algorithm:

- ▶ If we arrive at a symmetric case, is it necessary to search twice?
- ▶ In the process, can you summarize which searches are redundant?
- ▶ If you were to design a suitable evaluation function, how would you design it?

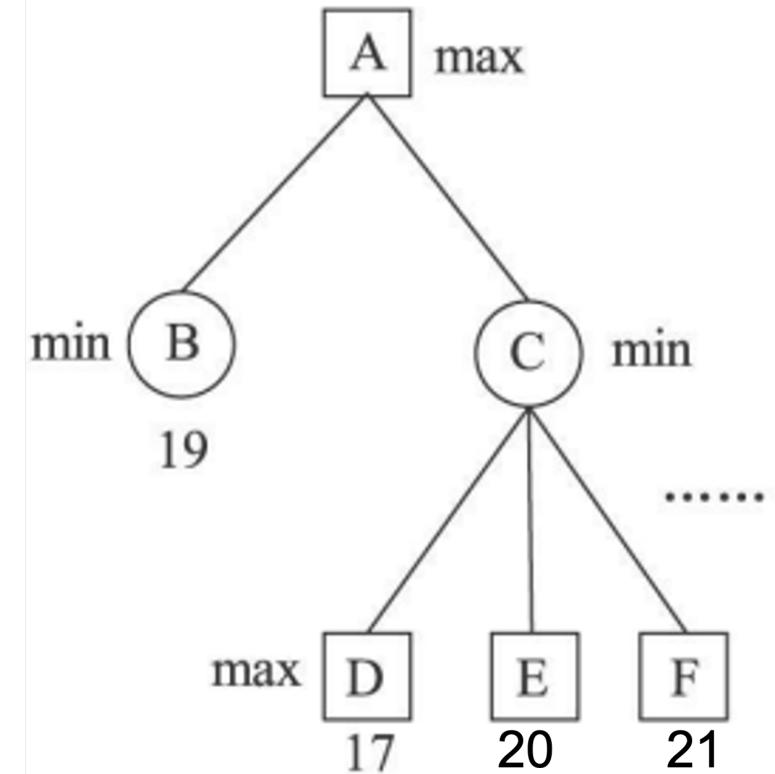
Pruning

- ▶ The Level-3 needs to get the maximum value of the Level-4. As the search proceed, the leftmost node in Level-4 already gets value 1 and return it to Level-3. Then, Level-3 continue to call the second node in Level-4, which already gets value -1. Note $-1 < 1$.
- ▶ Then, does the second node in Level-4 need to continue to search its right branch?
- ▶ If the right branch gets value larger than -1, it is obvious that Min still gets the value -1 when the search is finished. If the right branch gets value smaller than -1, it is clear that Max in Level-3 still gets the value 1. So, the result of Level-3 will remain the same no matter which value the right branch of the second node in Level-4 returns.
- ▶ Conclusion: The second right branch of the fourth layer can be cut off.



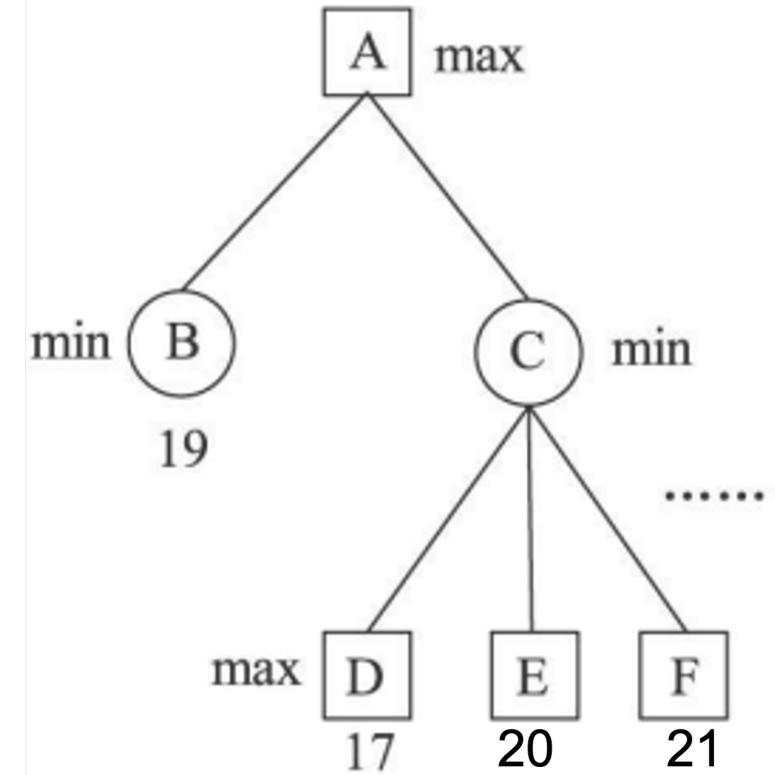
Alpha Pruning

- ▶ The value of **node A** should be the greater of the values of **node B** and **node C**. **Node B** is now known to have a value greater than the value of **node D**. Since the value of **node C** should be the **smallest** of the values of its children, this minimum value must be no larger than the value of **node D**, and therefore must be less than the value of **node B**, indicating the meaninglessness of the search of other children of **node C**, e.g., **node E and F**. Now, we can cut off the subtree rooted at **node C**. This optimization is called **Alpha pruning**.
- ▶ Question: What happens if searching **branches E and F** are in front of **D**?



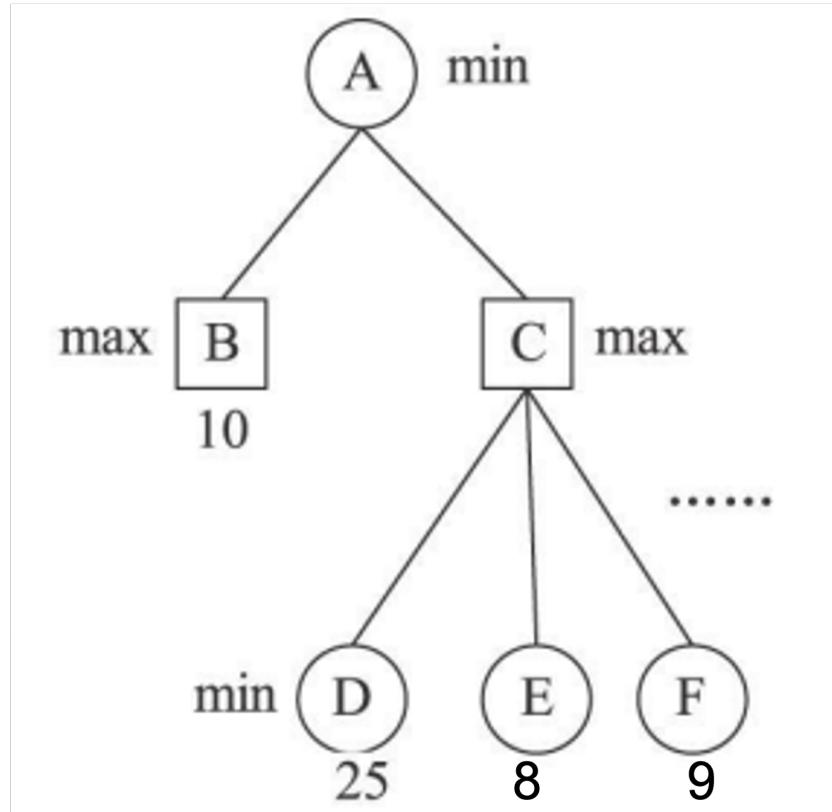
Alpha Pruning: Explanation

- ▶ **MAX Level:** At node A, the maximum value found in the child node is saved in **alpha**. This **alpha** value is passed to the next level along with the function call.
- ▶ The next level is **MIN level**. The minimum value currently found by the node of the **MIN level** is no larger than **alpha** value, so there is no need to continue searching.
- ▶ A sub-node needs to keep updating its own **beta** value. If the node branch does not terminate, and the currently found minimum value < **beta**, we need to update the **beta** value and pass it to the next level of the node.



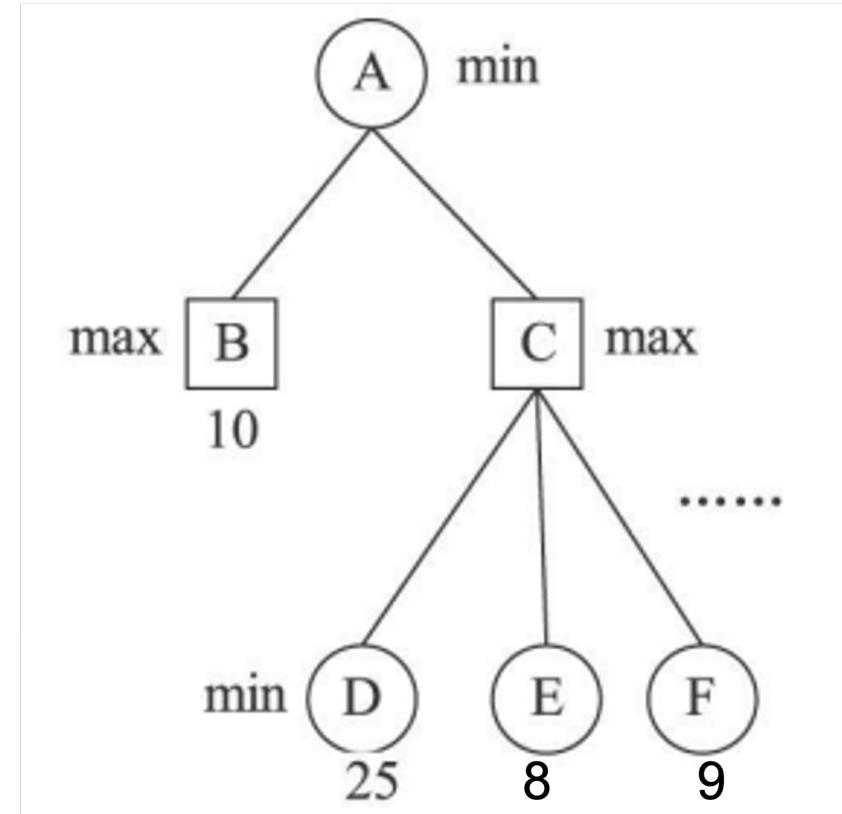
Beta Pruning

- ▶ The value of **node A** should be the lesser of the values of **node B** and **node C**. **Node B** is known to have a value less than the value of **node D**. Since the value of **node C** should be the **largest** of its sub-node values, this maximum value must be no less than the value of **node D**, and therefore greater than the value of **node B**, indicating that continuing to search for other children of **node C** have no meaning, and all subtrees rooted at **node C** can be **cut off**. This optimization is called **Beta pruning**.
- ▶ Question: What would happen if the branches of **E** and **F** are in front of **D**?



Beta Pruning: Explanation

- ▶ **MIN Level:** At **node A**, the **minimum value** found in the child node is saved in **beta**. This **beta** value is passed to the next level along with the function call.
- ▶ The next level is **MAX level**. The maximum value currently found by the node of the **MAX level** is no less than **beta** value, so there is no need to continue searching.
- ▶ A sub-node (max) needs to keep updating its own **alpha** value. If the node branch does not terminate, and the currently found maximum value >**alpha**, we need to **update** the **alpha** value and pass it to the next level of the node.



Alpha-Beta Pruning

- ▶ Applying Alpha-Beta pruning to the Minimax algorithm, we derive the Alpha-Beta search algorithm.
- ▶ Its optimization uses properties of Minimax and does not change the result of Minimax.
- ▶ The optimization depends on the order of nodes.

```
44 ▼ def alphabeta_search(state, game):
45 ▼     """Search game to determine best action; use alpha-beta pruning.
46     As in [Figure 5.7], this version searches all the way to the leaves."""
47
48     player = game.to_move(state)
49
50     # Functions used by alphabeta
51 ▼     def max_value(state, alpha, beta):
52 ▼         if game.terminal_test(state):
53             return game.utility(state, player)
54
55 ▼         v = -infinity
56         for a in game.actions(state):
57             v = max(v, min_value(game.result(state, a), alpha, beta))
58             if v >= beta:
59                 return v
60             alpha = max(alpha, v) update alpha value
61
62     def min_value(state, alpha, beta):
63         if game.terminal_test(state):
64             return game.utility(state, player)
65
66         v = infinity
67         for a in game.actions(state):
68             v = min(v, max_value(game.result(state, a), alpha, beta))
69             if v <= alpha:
70                 return v
71             beta = min(beta, v) Alpha Pruning Update beta value
72
73     # Body of alphabeta_cutoff_search:
74     best_score = -infinity
75     beta = infinity
76     best_action = None
77     for a in game.actions(state):
78         v = min_value(game.result(state, a), best_score, beta)
79         if v > best_score:
80             best_score = v
81             best_action = a
82
83     return best_action
```

Beta Pruning

Alpha Pruning

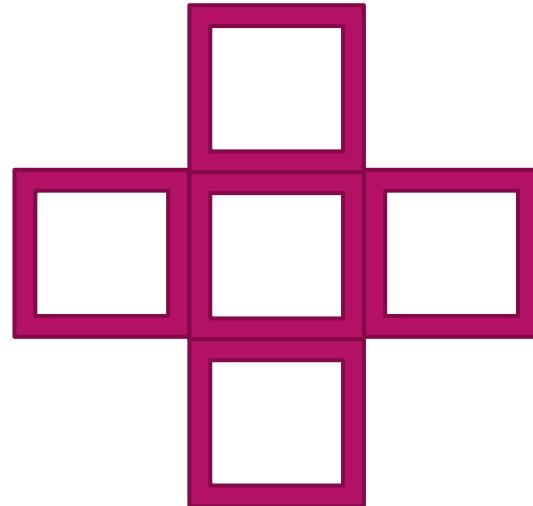
H-Minimax

- ▶ Use the **eval** function instead of the utility function
- ▶ Use **cutoff** test instead of **terminal test**
- ▶ Question: What are the benefits of doing this?

```
85 ▼ def alphabeta_cutoff_search(state, game, d=4, cutoff_test=None, eval_fn=None):
86 ▼     """Search game to determine best action; use alpha-beta pruning.
87     This version cuts off search and uses an evaluation function."""
88
89     player = game.to_move(state)
90
91     # Functions used by alphabeta
92 ▼     def max_value(state, alpha, beta, depth):
93 ▼         if cutoff_test(state, depth):
94             return eval_fn(state)
95         v = -infinity
96         for a in game.actions(state):
97             v = max(v, min_value(game.result(state, a),
98                                 alpha, beta, depth + 1))
99             if v >= beta:
100                 return v
101             alpha = max(alpha, v)
102     return v
103
104 ▼     def min_value(state, alpha, beta, depth):
105 ▼         if cutoff_test(state, depth):
106             return eval_fn(state)
107         v = infinity
108         for a in game.actions(state):
109             v = min(v, max_value(game.result(state, a),
110                     alpha, beta, depth + 1))
111             if v <= alpha:
112                 return v
113             beta = min(beta, v)
114     return v
115
116     # Body of alphabeta_cutoff_search starts here:
117     # The default test cuts off at depth d or at a terminal state
118     cutoff_test = (cutoff_test or
119                   (lambda state, depth: depth > d or
120                    game.terminal_test(state)))
121     eval_fn = eval_fn or (lambda state: game.utility(state, player))
122     best_score = -infinity
123     beta = infinity
124     best_action = None
125     for a in game.actions(state):
126         v = min_value(game.result(state, a), best_score, beta, 1)
127         if v > best_score:
128             best_score = v
129             best_action = a
130     return best_action
```

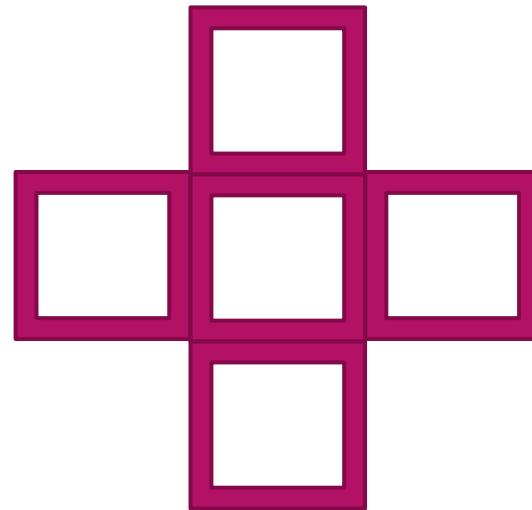
Design of Evaluation Function

- ▶ How to design the evaluation function of this game?



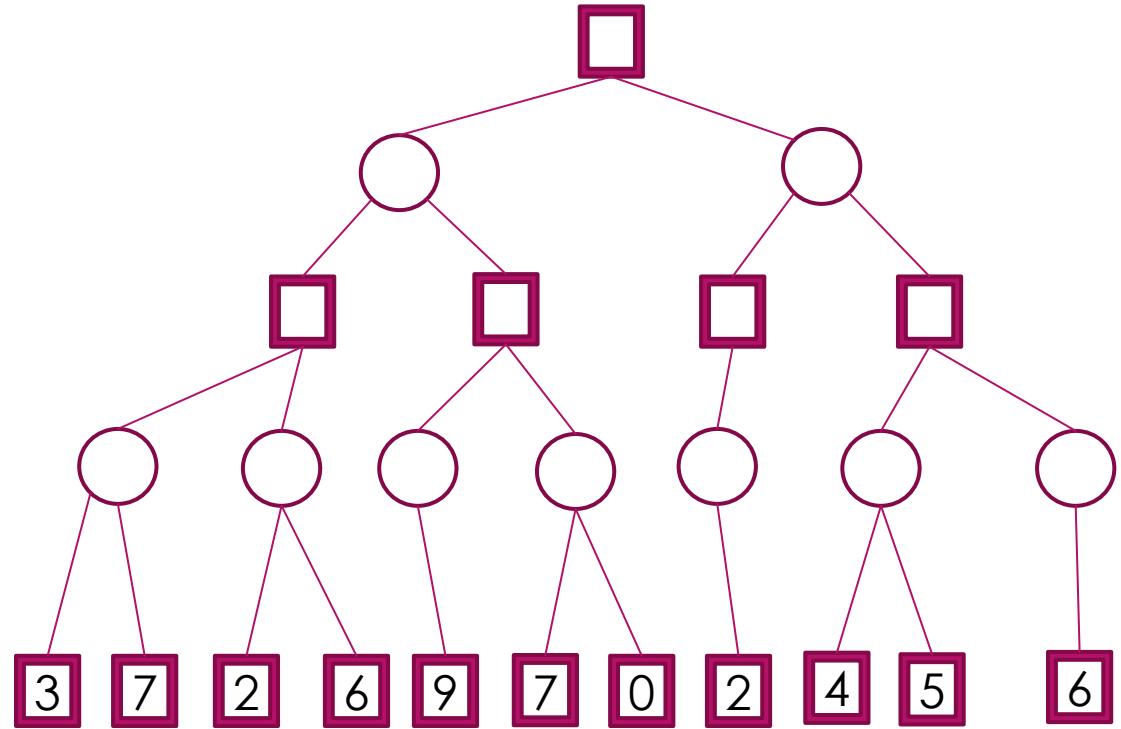
Design of Evaluation Function

- ▶ How to design the evaluation function of this game?
- ▶ Suggestions:
 - If X is in the middle position, the score is 4; if it is in the four sides, the score is 1
 - If O is in the middle position, the score is -4; if it is at the four sides, the score is -1
 - Accumulate all X and O scores as described above



Application of Alpha-Beta Pruning

- ▶ The execution result of a MINIMAX algorithm is shown in the right figure.
- ▶ Using the Alpha-Beta pruning algorithm to prune the right figure.



Tic-Tac-Toe

- ▶ <http://aimacode.github.io/aima-javascript/5-Adversarial-Search/>