Adversarial Search

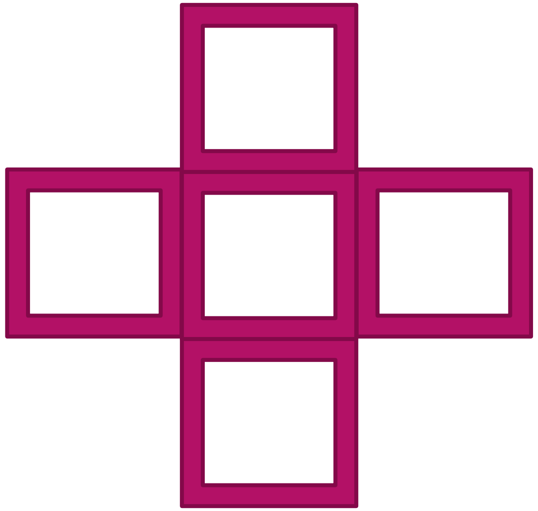
Minimax Algorithm

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

```python
def minimax_decision(state, game):
    """Given a state in a game, calculate the best move by searching
    forward all the way to the terminal states. [Figure 5.3]"""

    player = game.to_move(state)

    def max_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a)))
        return v

    def min_value(state):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a)))
        return v

    # Body of minimax_decision:
    return argmax(game.actions(state),
                  key=lambda a: min_value(game.result(state, a)))
```

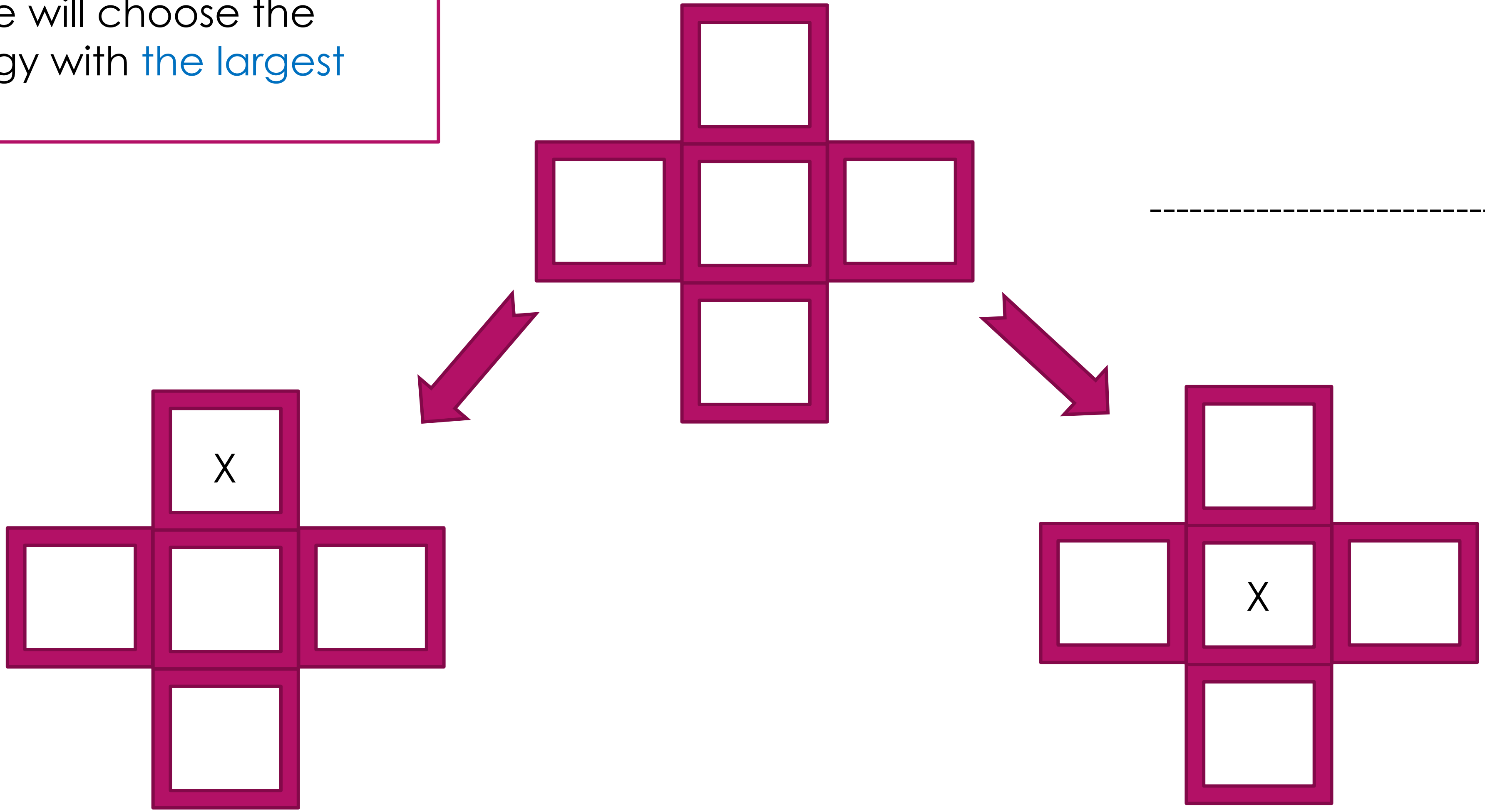
A Basic Example

- 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

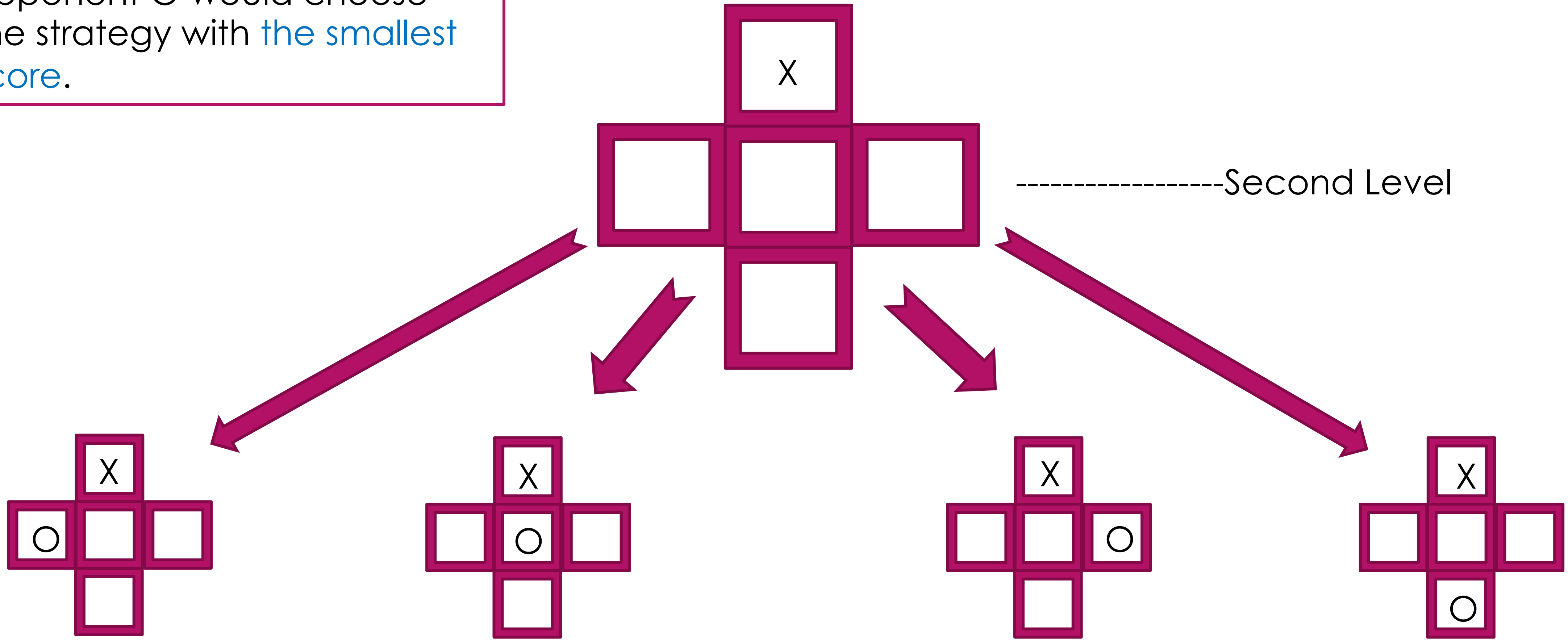


-----------------------------First Level

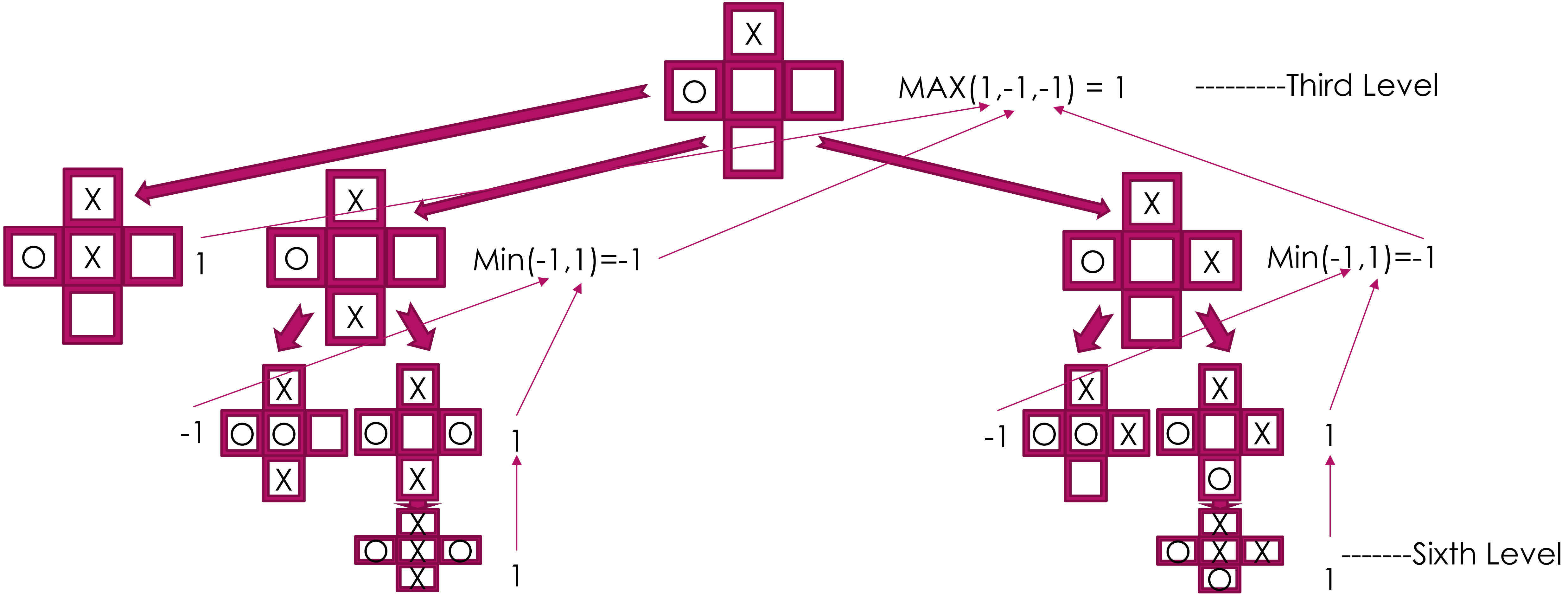
-----Second Level

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.

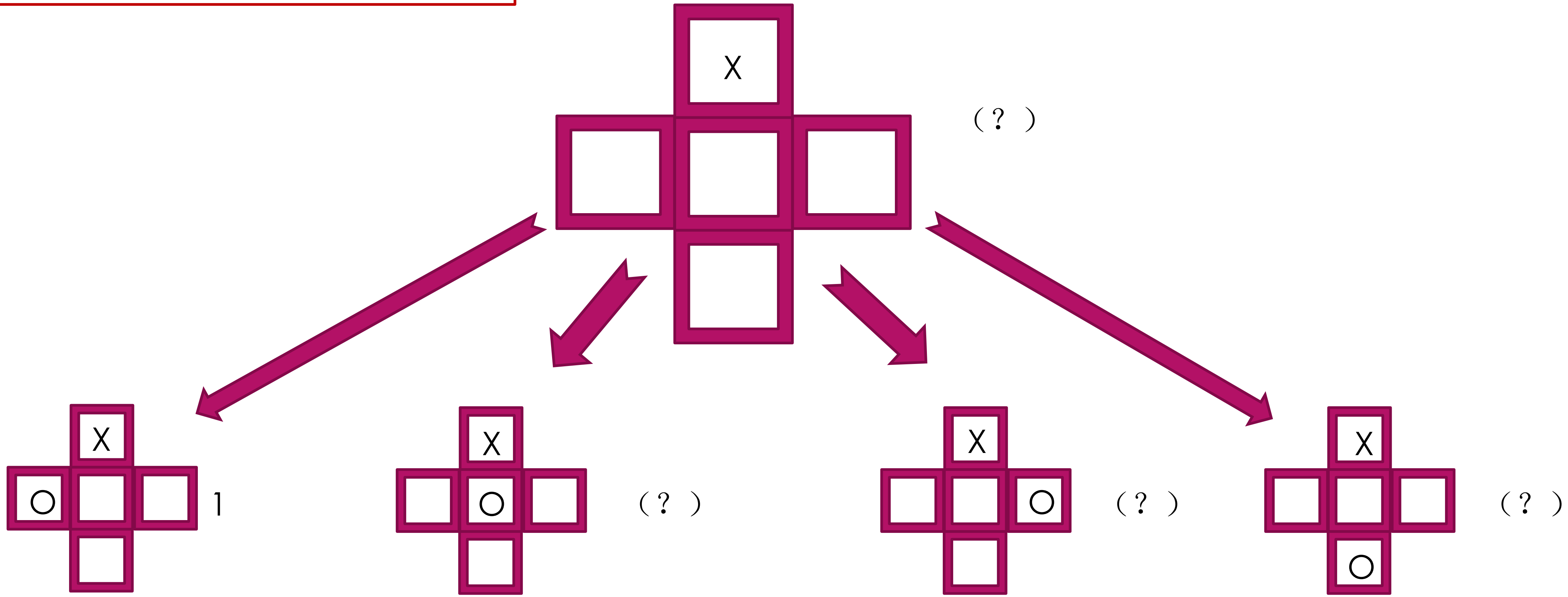


A Basic Example: Third Level to Sixth Level

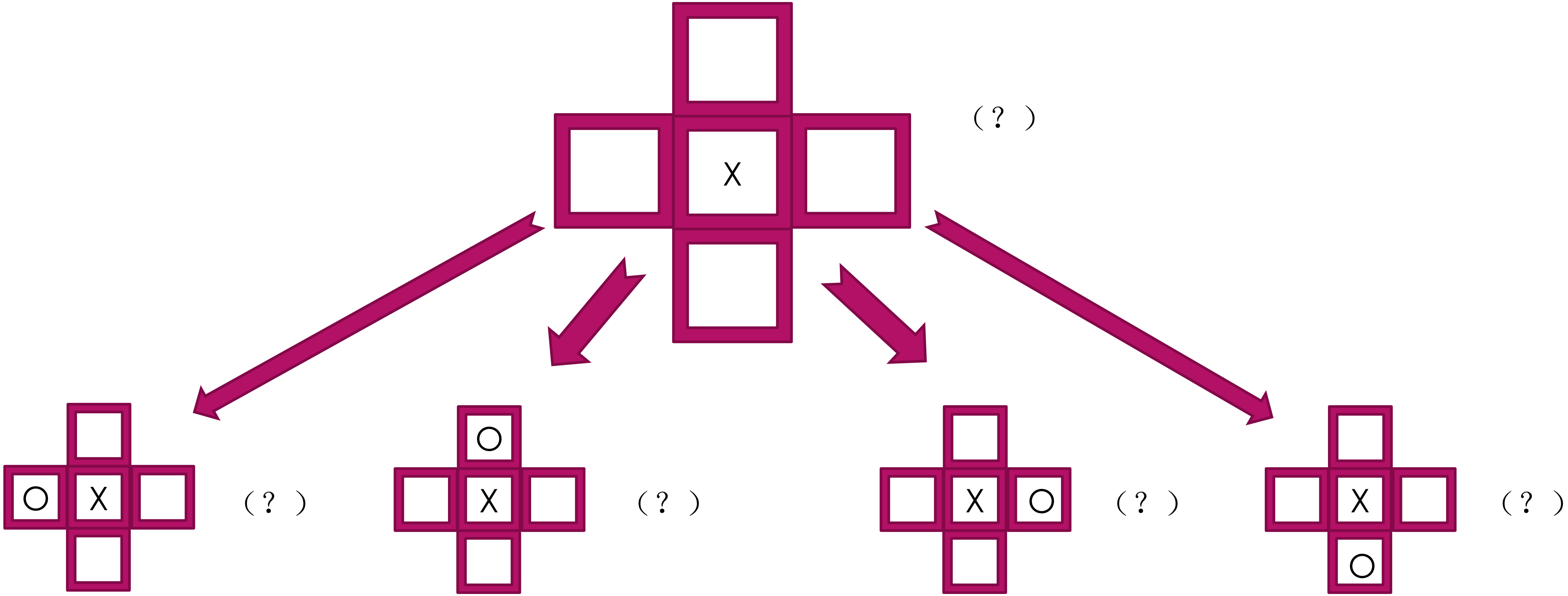


Fill in the Blank

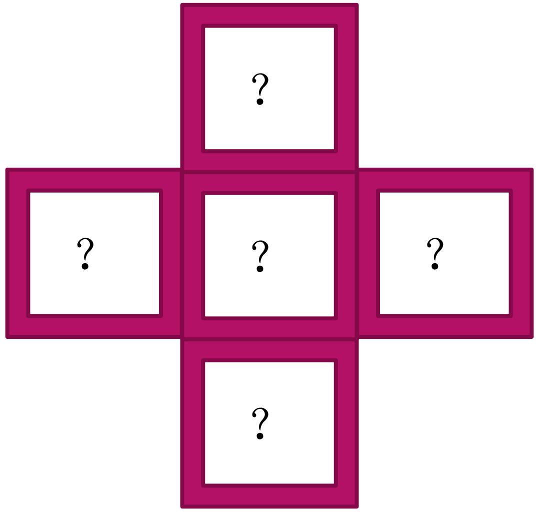
Please complete the scores in the remaining boxes according to the above procedure



Fill in the Blank



Which position would X choose in the first step?

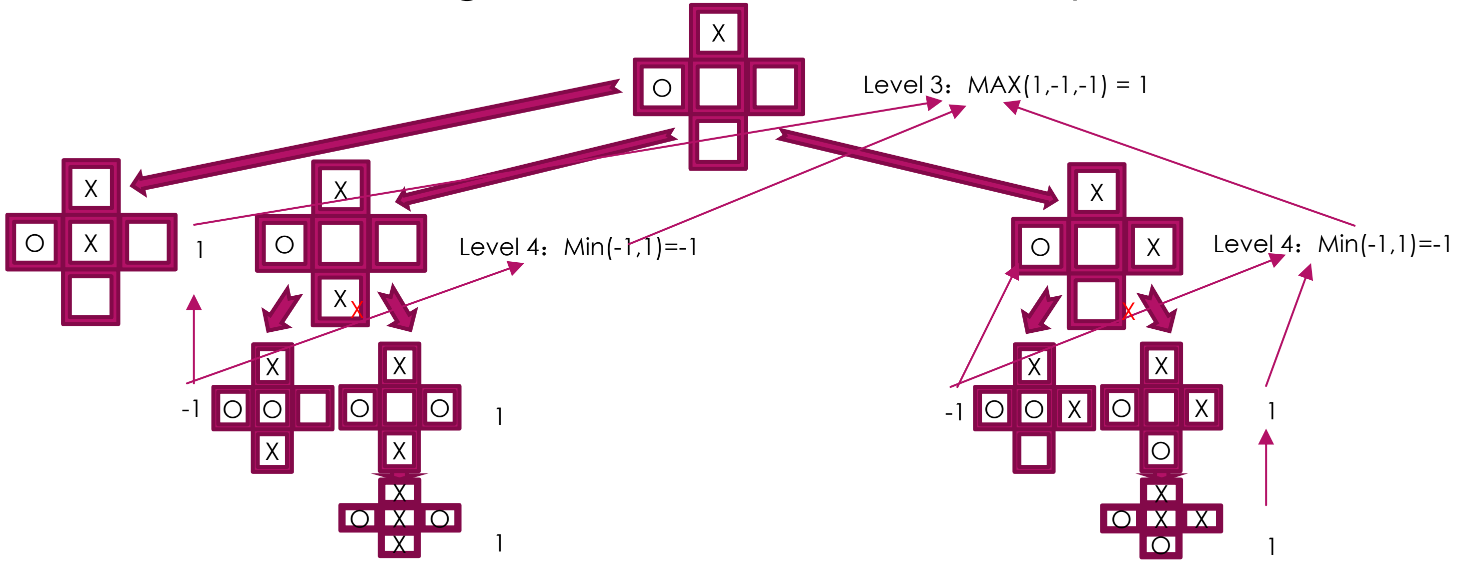


In the implementation of the Minimax algorithm:

- 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?

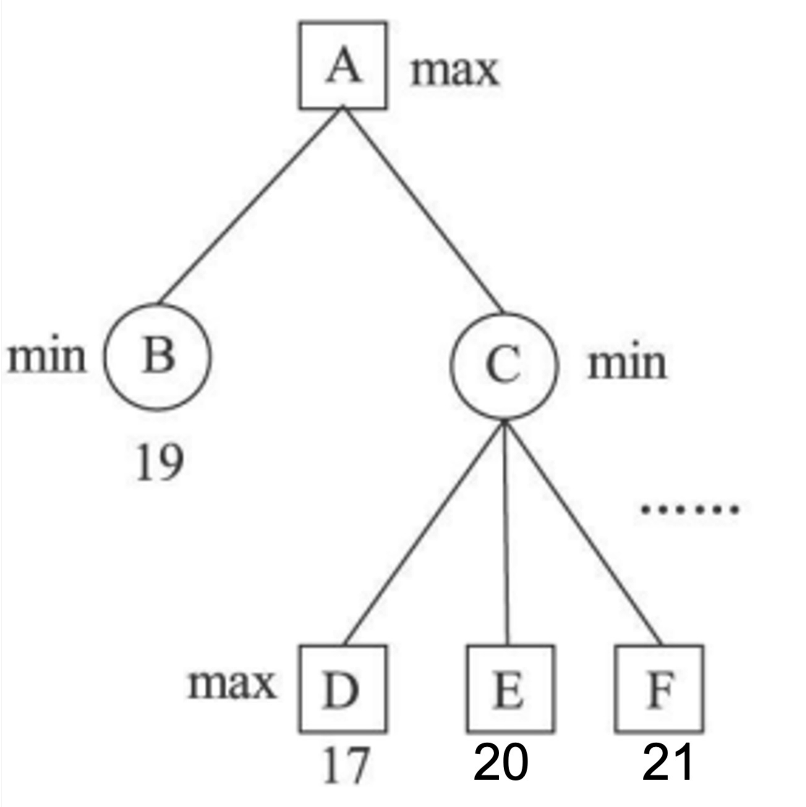
Pruning

- The Level-3 needs to get the maximum value of the Level-4. As the search proceeds, the leftmost node in Level-4 already gets value 1 and returns it to Level-3. Then, Level-3 continues 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 a value larger than -1, it is obvious that Min still gets the value −1 when the search is finished. If the right branch gets a 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.



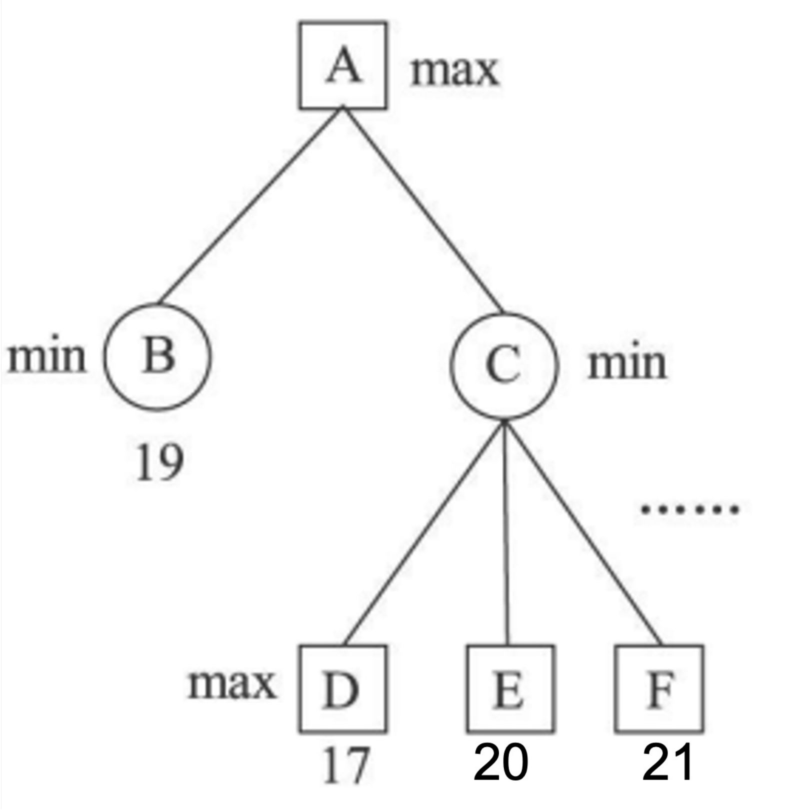
Alpha Pruning

- 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 meaningless of the search for 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**?



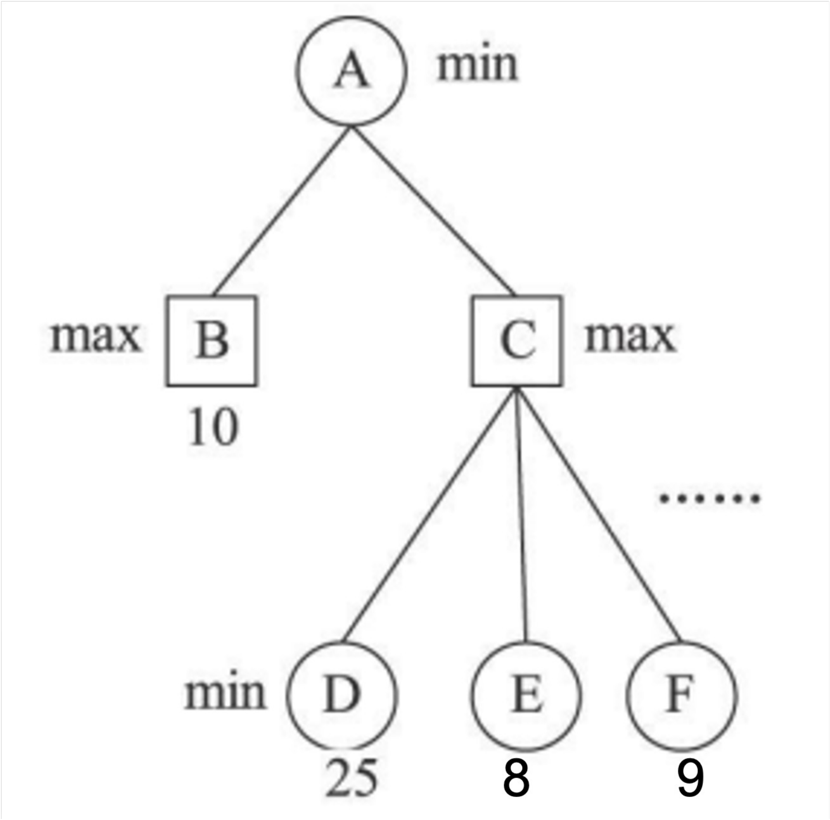
Alpha Pruning: Explanation

- **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 **the 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.



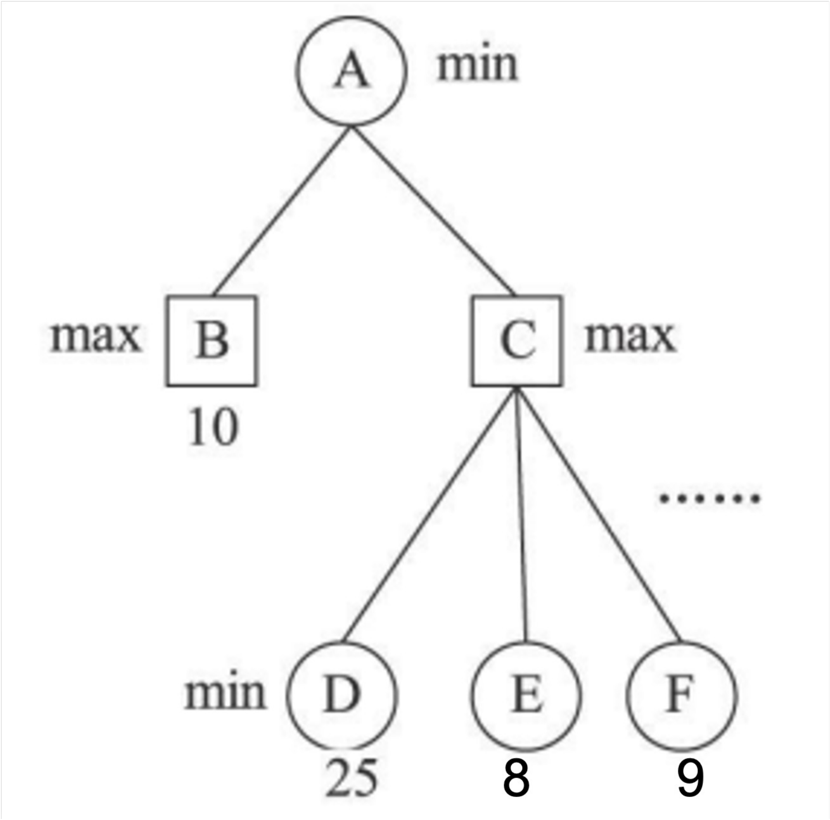
Beta Pruning

- 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.

```
def alphabeta_search(state, game):
    """Search game to determine best action; use alpha-beta pruning.
    As in [Figure 5.7], this version searches all the way to the leaves."""

    player = game.to_move(state)

    # Functions used by alphabeta
    def max_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a), alpha, beta))
            if v >= beta:          Beta Pruning
                return v
            alpha = max(alpha, v)   update alpha value
        return v

    def min_value(state, alpha, beta):
        if game.terminal_test(state):
            return game.utility(state, player)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a), alpha, beta))
            if v <= alpha:          Alpha Pruning
                return v
            beta = min(beta, v)     Update beta value
        return v

    # Body of alphabeta_cutoff_search:
    best_score = -infinity
    beta = infinity
    best_action = None
    for a in game.actions(state):
        v = min_value(game.result(state, a), best_score, beta)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

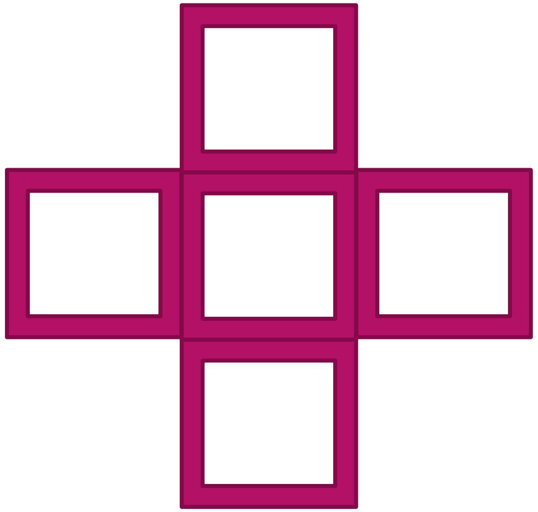
H-Minimax

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

```python
def alphabeta_cutoff_search(state, game, d=4, cutoff_test=None, eval_fn=None):
    """Search game to determine best action; use alpha-beta pruning.
    This version cuts off search and uses an evaluation function."""

    player = game.to_move(state)

    # Functions used by alphabeta
    def max_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = -infinity
        for a in game.actions(state):
            v = max(v, min_value(game.result(state, a),
                                 alpha, beta, depth + 1))
            if v >= beta:
                return v
            alpha = max(alpha, v)
        return v

    def min_value(state, alpha, beta, depth):
        if cutoff_test(state, depth):
            return eval_fn(state)
        v = infinity
        for a in game.actions(state):
            v = min(v, max_value(game.result(state, a),
                                 alpha, beta, depth + 1))
            if v <= alpha:
                return v
            beta = min(beta, v)
        return v

    # Body of alphabeta_cutoff_search starts here:
    # The default test cuts off at depth d or at a terminal state
    cutoff_test = (cutoff_test or
                   (lambda state, depth: depth > d or
                    game.terminal_test(state)))
    eval_fn = eval_fn or (lambda state: game.utility(state, player))
    best_score = -infinity
    beta = infinity
    best_action = None
    for a in game.actions(state):
        v = min_value(game.result(state, a), best_score, beta, 1)
        if v > best_score:
            best_score = v
            best_action = a
    return best_action
```

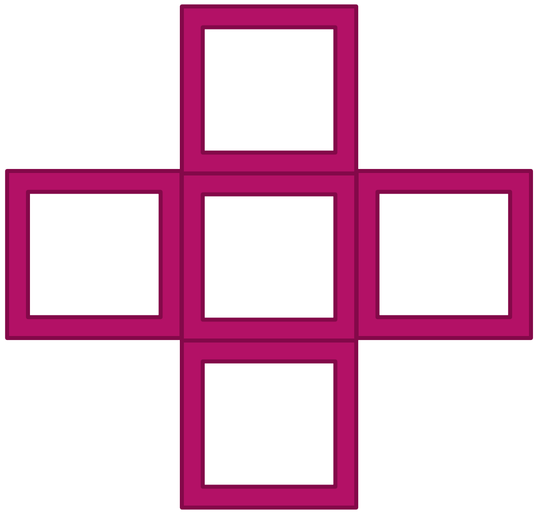
Design of Evaluation Function

- How to design the evaluation function of this game?



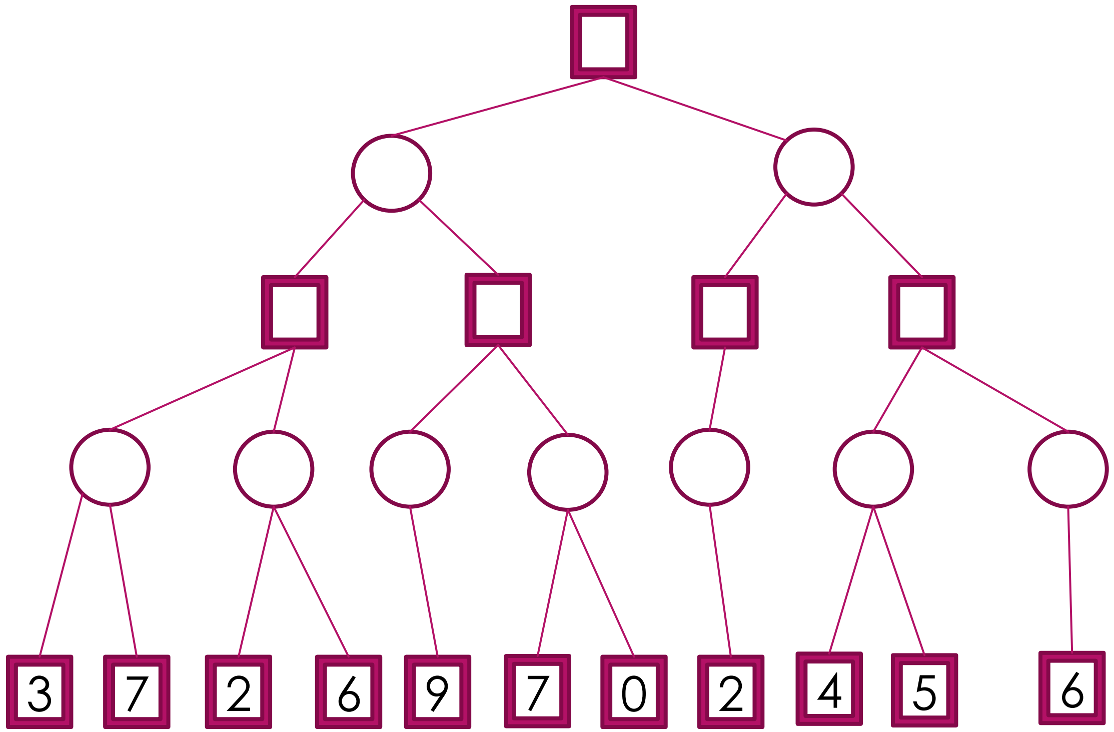
Design of Evaluation Function

- 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/