Adversarial Search and Game Playing

is a field in artificial intelligence (AI) and computer science that focuses on developing strategies and algorithms for playing games where two or more opponents or players have conflicting or adversarial goals. In this context, "adversarial" refers to the competitive nature of these games, where one player's gain is another player's loss. This field has applications in various board games, card games, video games, and even real-world strategic decision-making scenarios.

البحث التنافسي ولعب الألعاب هو مجال في الذكاء الاصطناعي AI وعلوم الكمبيوتر الذي يركز على تطوير الاستراتيجيات والخوارزميات لممارسة الألعاب التي يكون فيها لاثنين أو أكثر من المنافسين أو اللاعبين أهداف متعارضة أو متعارضة. وفي هذا السياق، يشير مصطلح "الخصومة" إلى الطبيعة التنافسية لهذه الألعاب، حيث يكون مكسب أحد اللاعبين بمثابة خسارة للاعب آخر، ولهذا المجال تطبيقات في العديد من ألعاب الطاولة، وألعاب الورق، وألعاب الفيديو، وحتى سيناريوهات اتخاذ القرار الاستراتيجي في العالم الحقيقي.

These techniques are fundamental in the development of AI agents for playing games:

تعتبر هذه التقنيات أساسية في تطوير عملاء الذكاء الاصطناعي لممارسة الألعاب:

1. **Minimax Algorithm:**

- **Minimax** is a decision-making algorithm commonly used in two-player, zero-sum games like chess, checkers, or tic-tac-toe.

خوارزمية الحد الأدنى: هي خوارزمية لاتخاذ القرار تُستخدم بشكل شائع في الألعاب التي يكون مجموع اللاعبين فيها صفر، مثل الشطرنج أو لعبة الداما أو tic-tac-toe.

- The algorithm aims to find the optimal move for a player while considering that the opponent is also trying to minimize the player's chances of winning.

هدف الخوارزمية إلى العثور على الحركة المثالية للاعب مع الأخذ في الاعتبار أن الخصم يحاول أيضًا تقليل فرص فوز اللاعب.

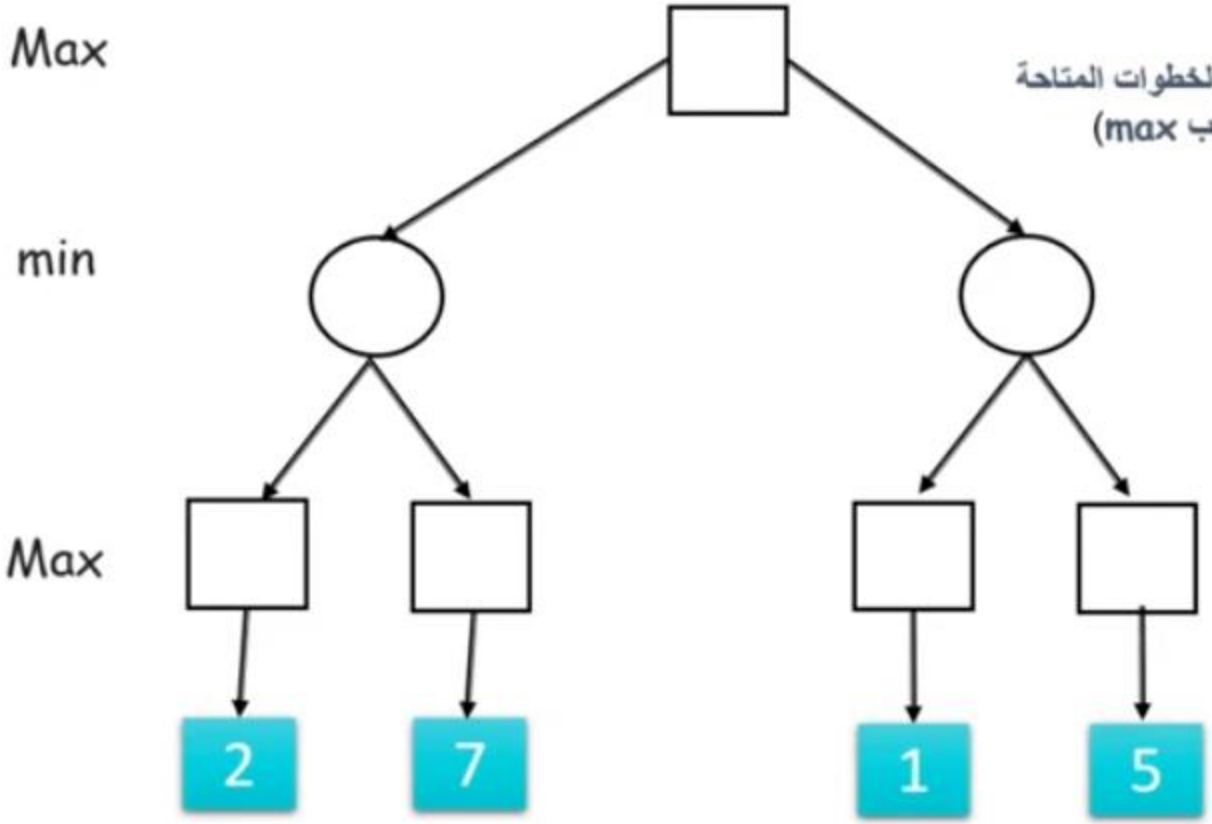
- It works by creating a game tree that explores all possible moves and their outcomes, assigning scores to terminal states (win, lose, draw), and backpropagating these scores to determine the best move for the current player.

- يعمل عن طريق إنشاء شجرة لعبة تستكشف جميع التحركات الممكنة ونتائجها، وتعيين النتائج للحالات النهائية (الفوز، الخسارة، التعادل)، وإعادة نشر هذه النتائج لتحديد أفضل حركة اللاعب الحالي.

- Minimax guarantees an optimal strategy if both players play perfectly, but it can be computationally expensive for deep game trees.

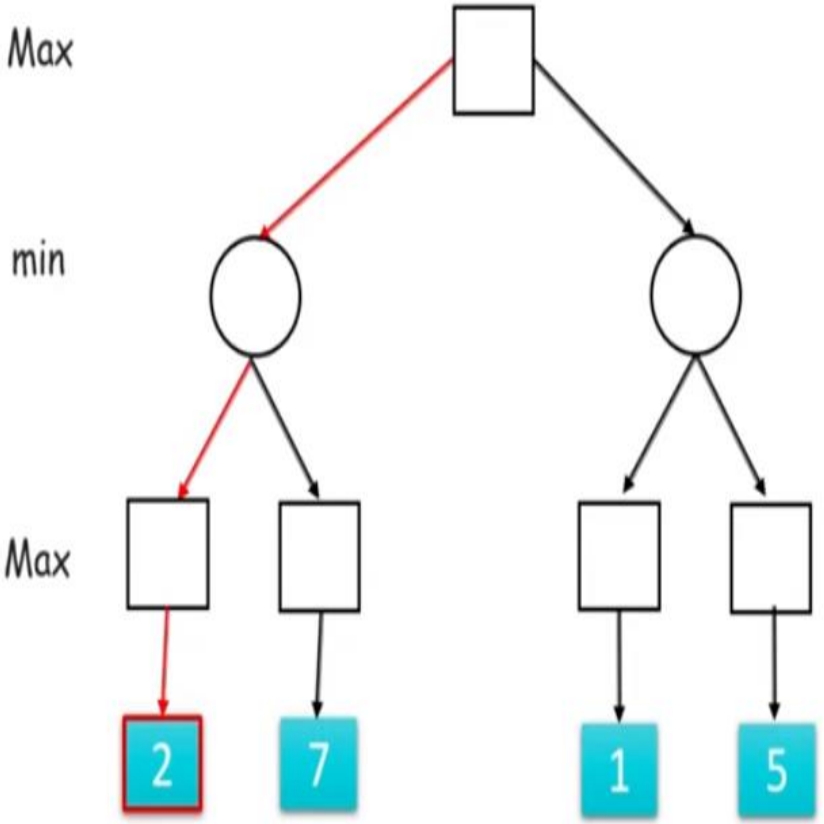
- يضمن Minimax استراتيجية مثالية إذا لعب كلا اللاعبين بشكل مثالي، ولكنه قد يكون مكلفًا من الناحية الحسابية لأشجار اللعبة العميقة.

Min Max algorithm خوارزميات البحث الذكية



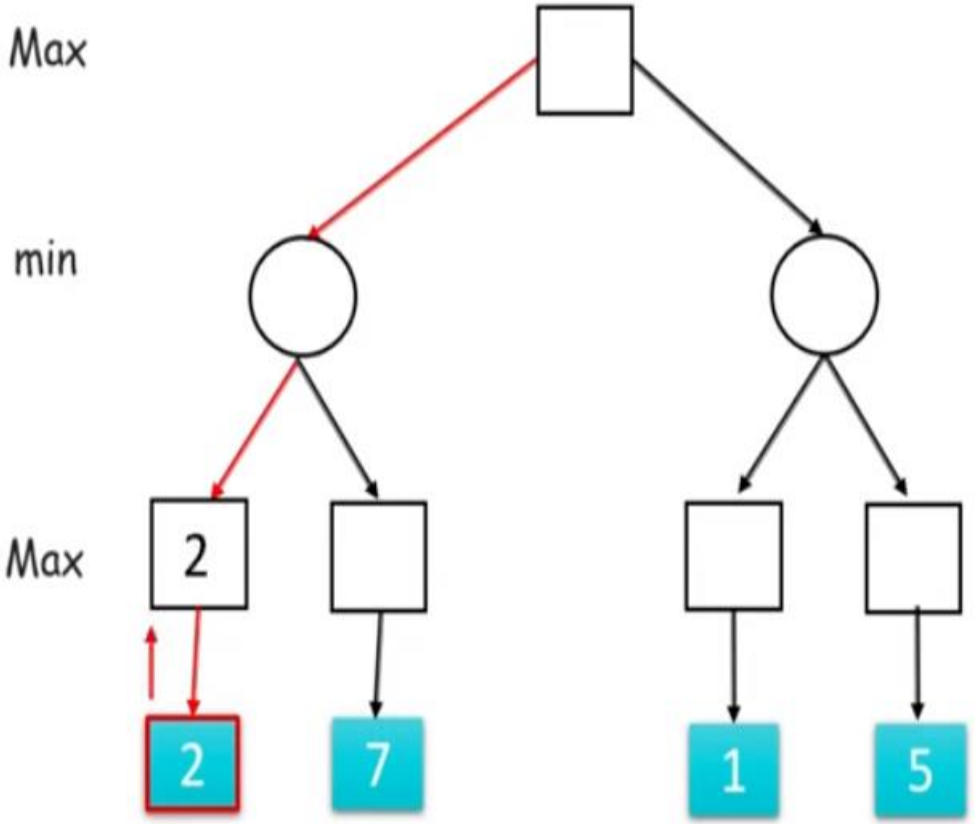
نريد اختيار الخطوة الأفضل من بين الخطوات المتاحة
للوصول إلى فوز اللاعب الأول (اللاعب max)

Use Min Max Algorithm with DFS to find best movements for player Max to win the game.

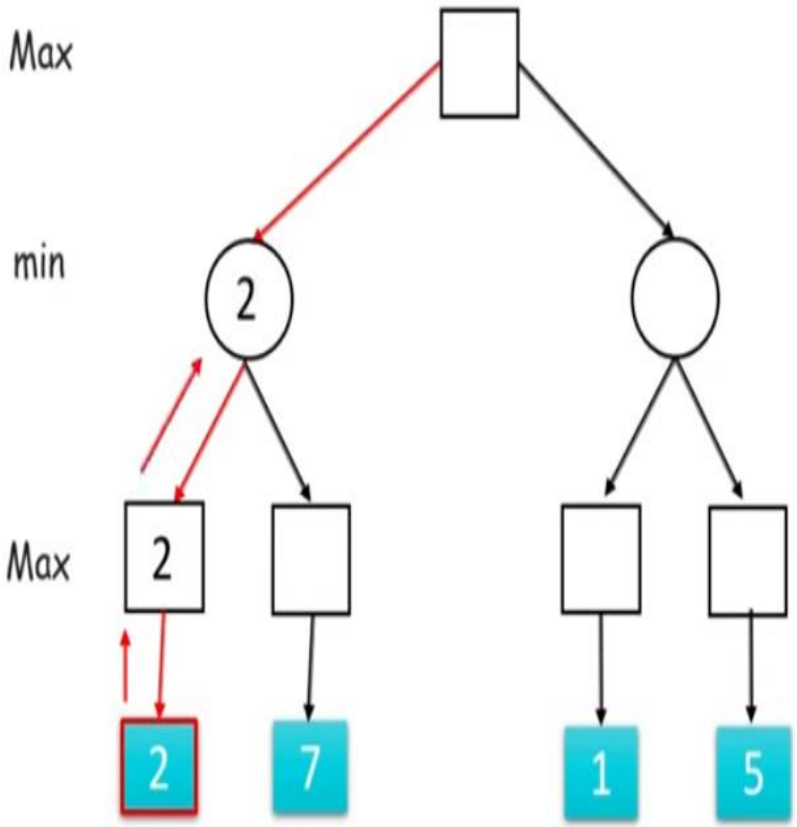


نبدأ من جذر الشجرة root ونتجه الى اقصى اليسار وبالعمق الى ان نصل الى رقم 2.
وهي تمثل قيمة افتراضية تدل على تقييم اللعبة في حال سلكنا هذا الطريق.

Min Max algorithm خوارزميات البحث الذكية

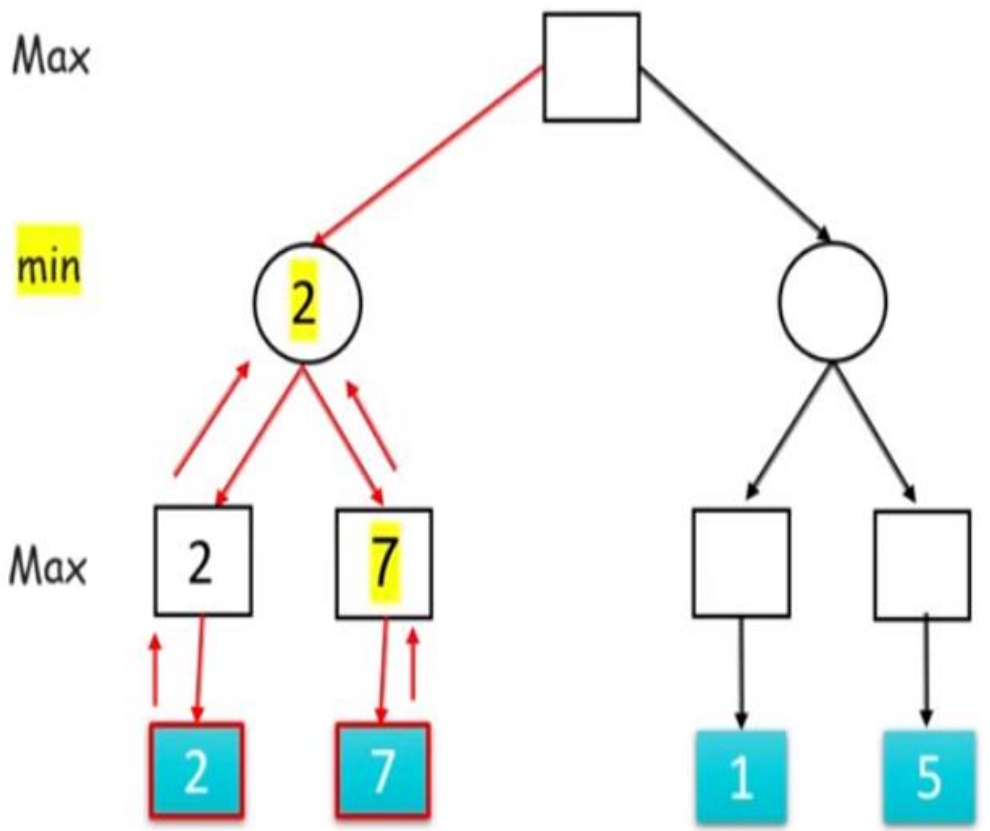


نأخذ 2 ونتجه بها الى الاعلى ونضعها عند اللاعب max ثم نتجه نحو الاعلى مرة اخرى ونضعها عند اللاعب min كما في الشكل

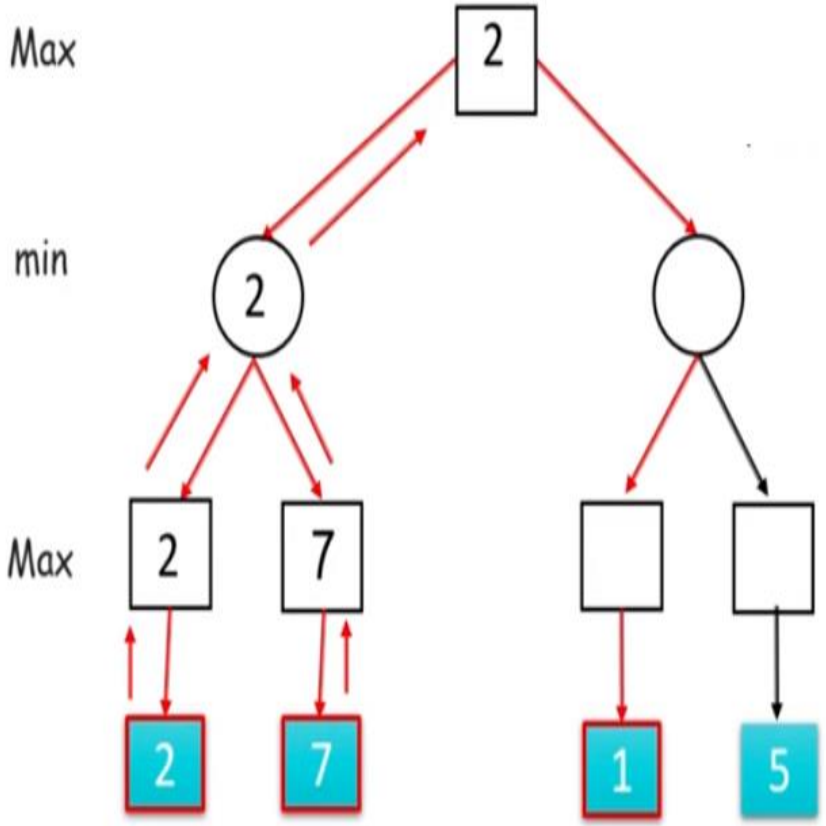


بما انه يوجد child لهذه النود اذن نتجه نحو العمق. لنصل الى الرقم 7. نأخذ الرقم 7 ونتجه به نحو الاعلى. نضعها عند max ثم نحاول نتجه بها الى الاعلى.

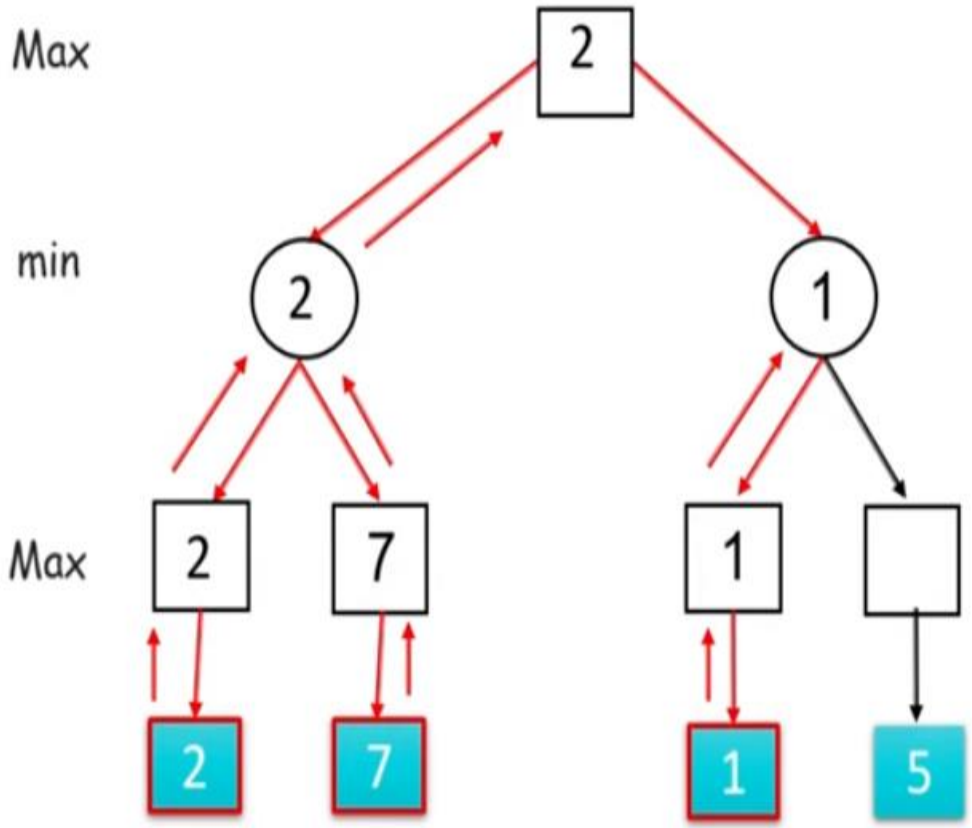
Min Max algorithm خوارزميات البحث الذكية

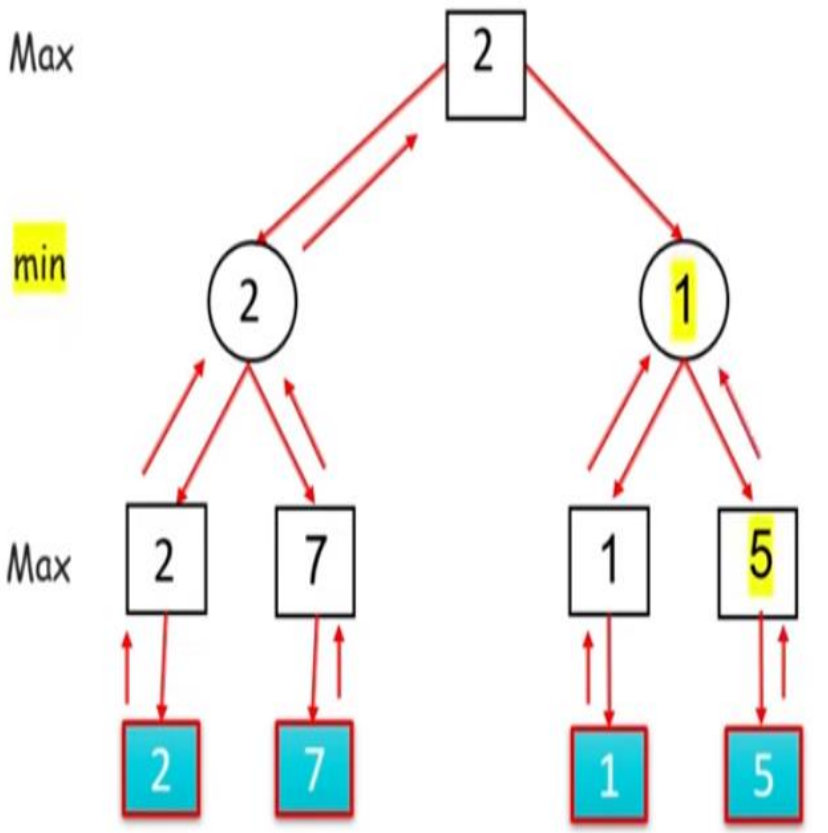


عند اللعب min اصبح عندي قيمتين 2 و 7 . اختار 2 لان دور اللاعب min . اذن اصعد بال 2 الى الاعلى واضعها مع max ثم اتجه الى جهة اليمين نحو العمق.

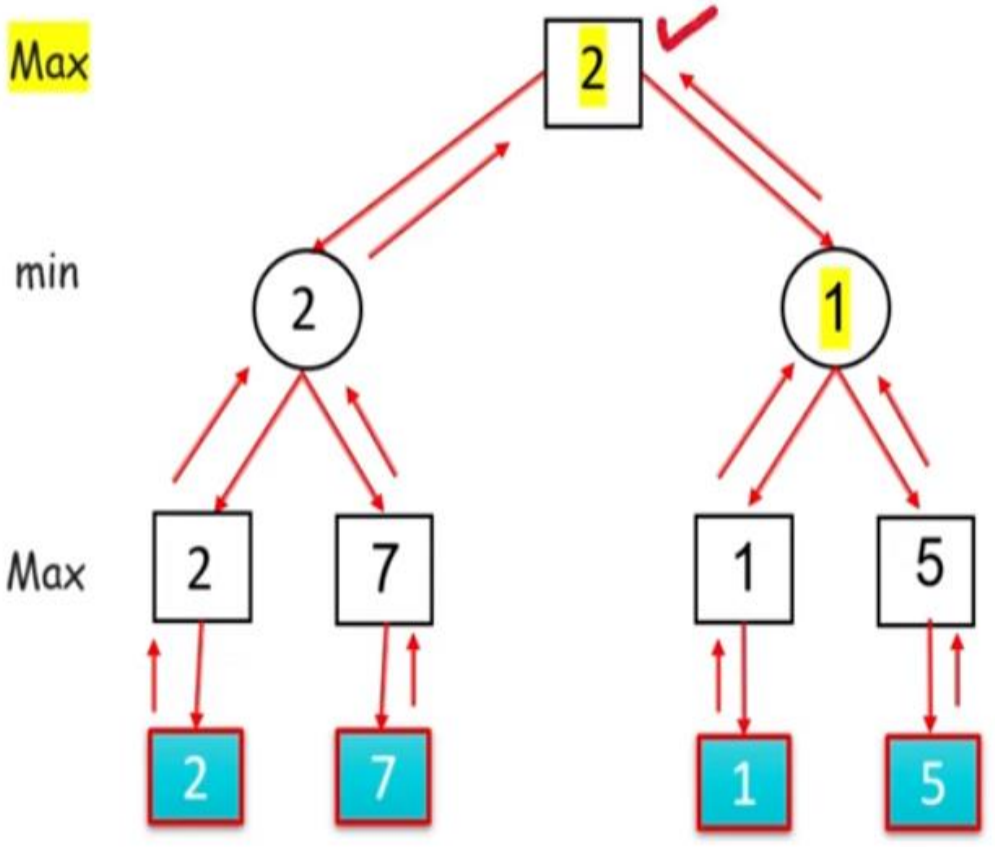


Min Max algorithm خوارزميات البحث الذكية

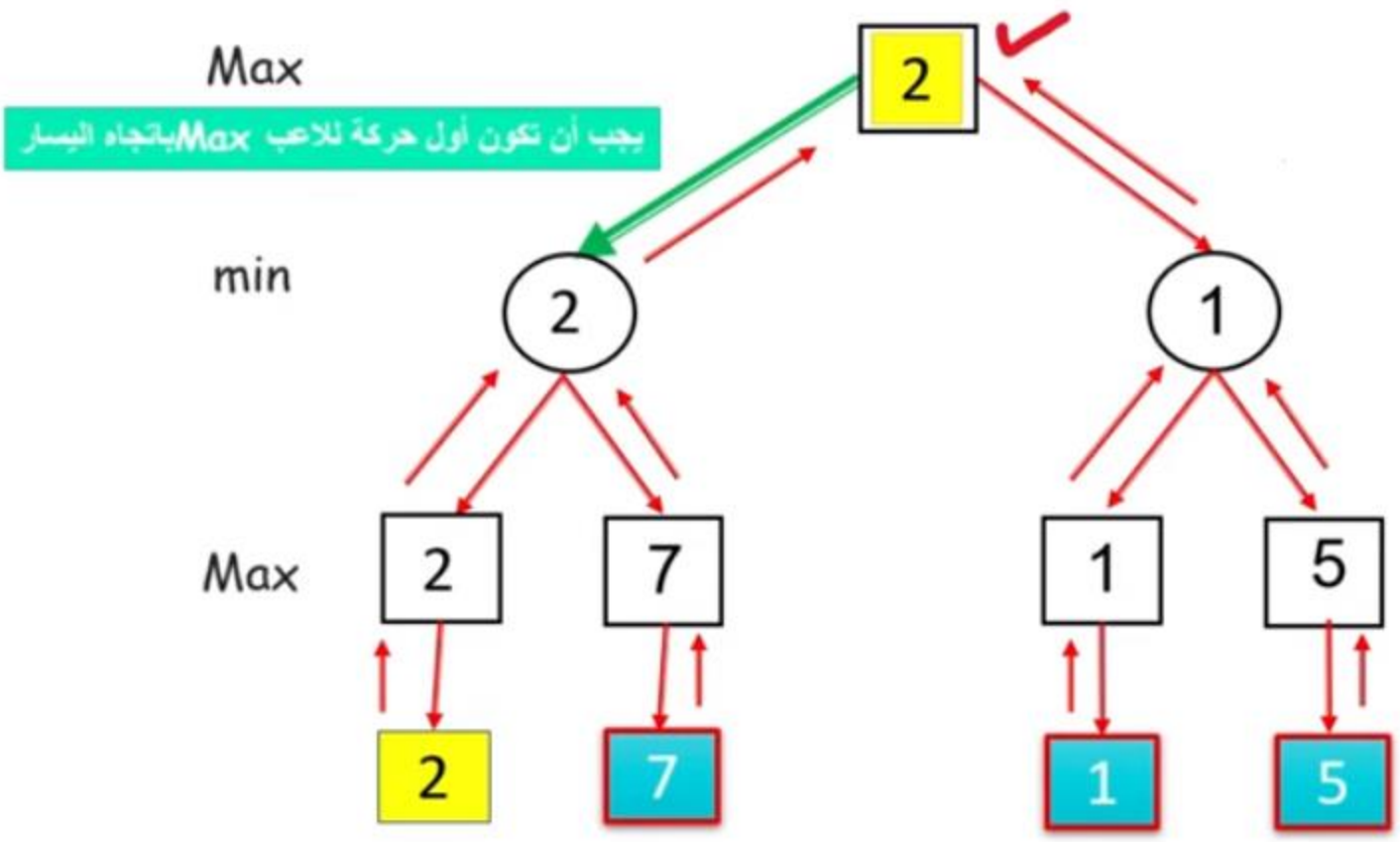




Min Max algorithm خوارزميات البحث الذكية



Min Max algorithm خوارزميات البحث الذكية

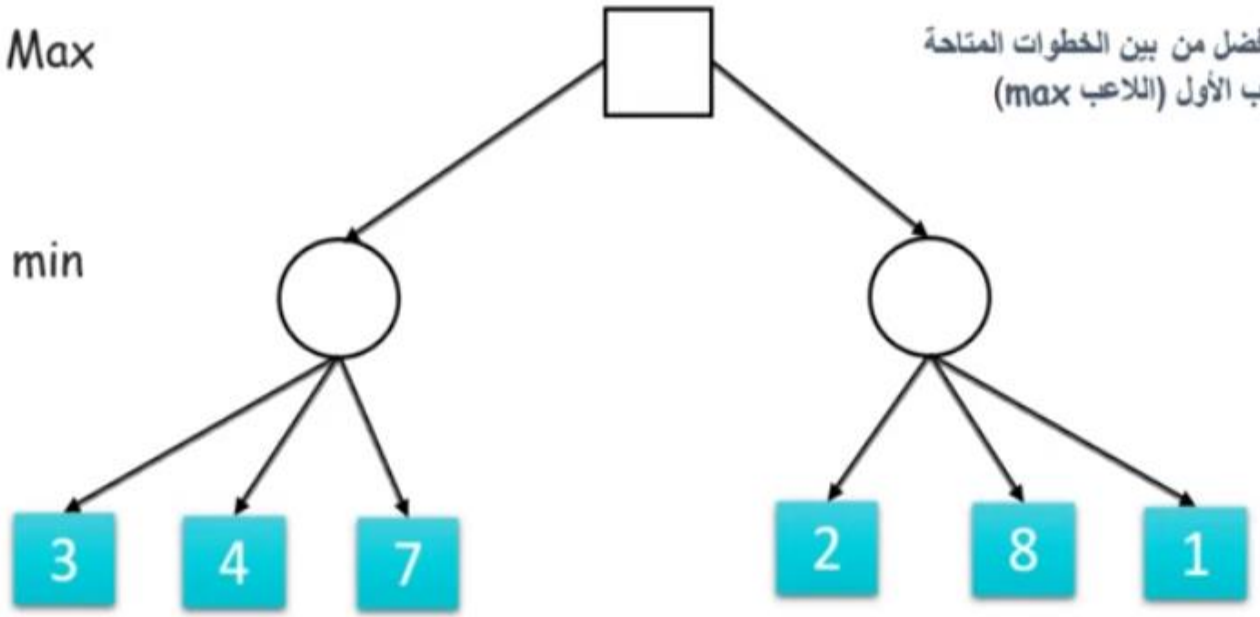


اذن نستنتج ان اول حركة للاعب max يجب ان تكون نحو يمين ليحقق الفوز .

HOMEWORK#2

Min Max algorithm

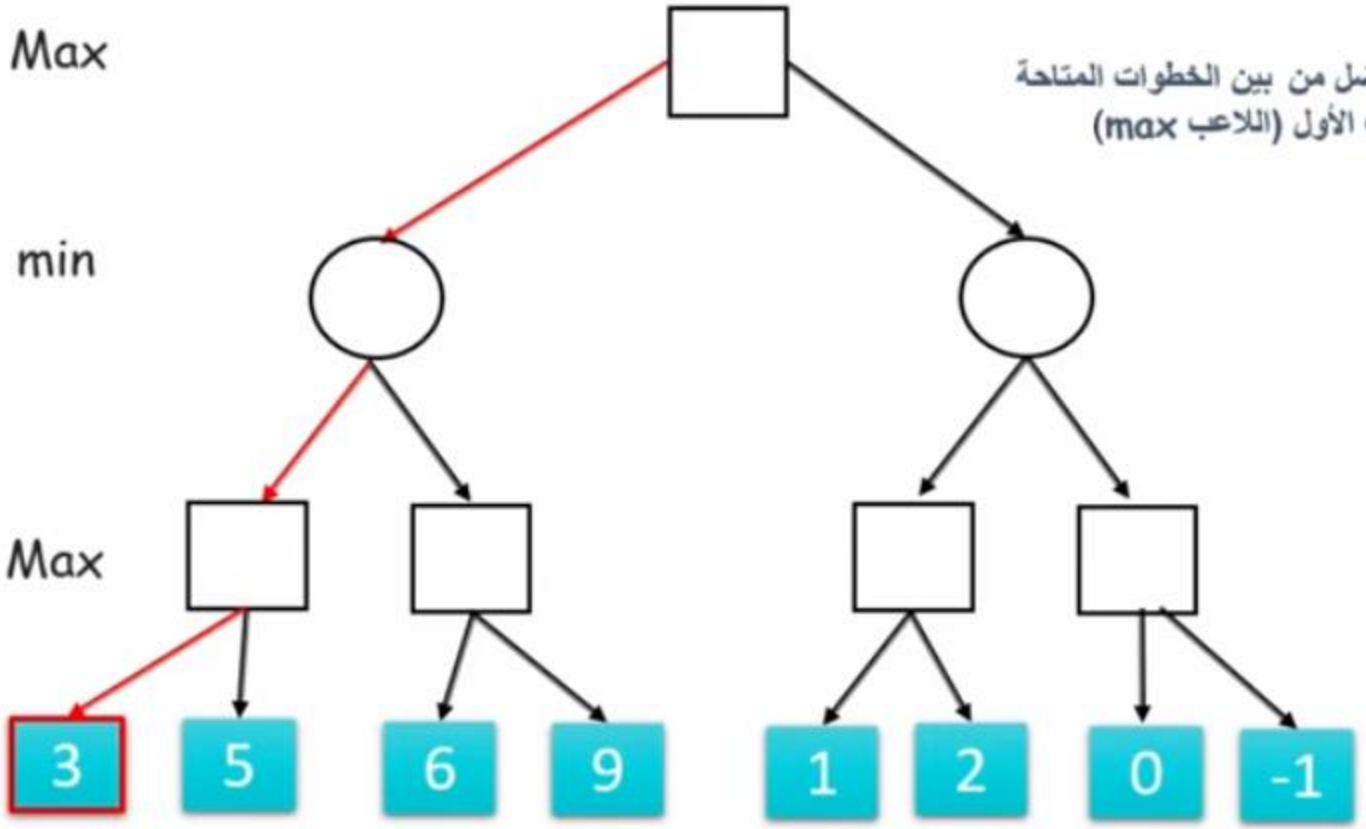
خوارزميات البحث الذكية



نريد اختيار الخطوة الأفضل من بين الخطوات المتاحة
للوصول إلى فوز اللاعب الأول (اللاعب max)

HOME WORK#3

Min Max algorithm خوارزميات البحث الذكية



نريد اختيار الخطوة الأفضل من بين الخطوات المتاحة
للوصول إلى فوز اللاعب الأول (اللاعب max)

2. Alpha-Beta Pruning:

- Alpha-Beta Pruning is an optimization technique applied to the Minimax algorithm to reduce the number of nodes explored in the game tree.

Alpha-Beta Pruningهي تقنية تحسين مطبقة على خوارزمية Minimaxلتقليل عدد العقد المستكشفة في شجرة اللعبة.

- By maintaining two values, alpha and beta, for each node, it can prune branches of the tree that won't affect the final decision.

من خلال الحفاظ على قيمتين، ألفا وبيتا، لكل عقدة، يمكن تقليم فروع الشجرة التي لن تؤثر على القرار النهائي.

- If, during the tree traversal, it is found that a branch is less promising than a previously explored branch, it can be safely ignored, saving computation time.

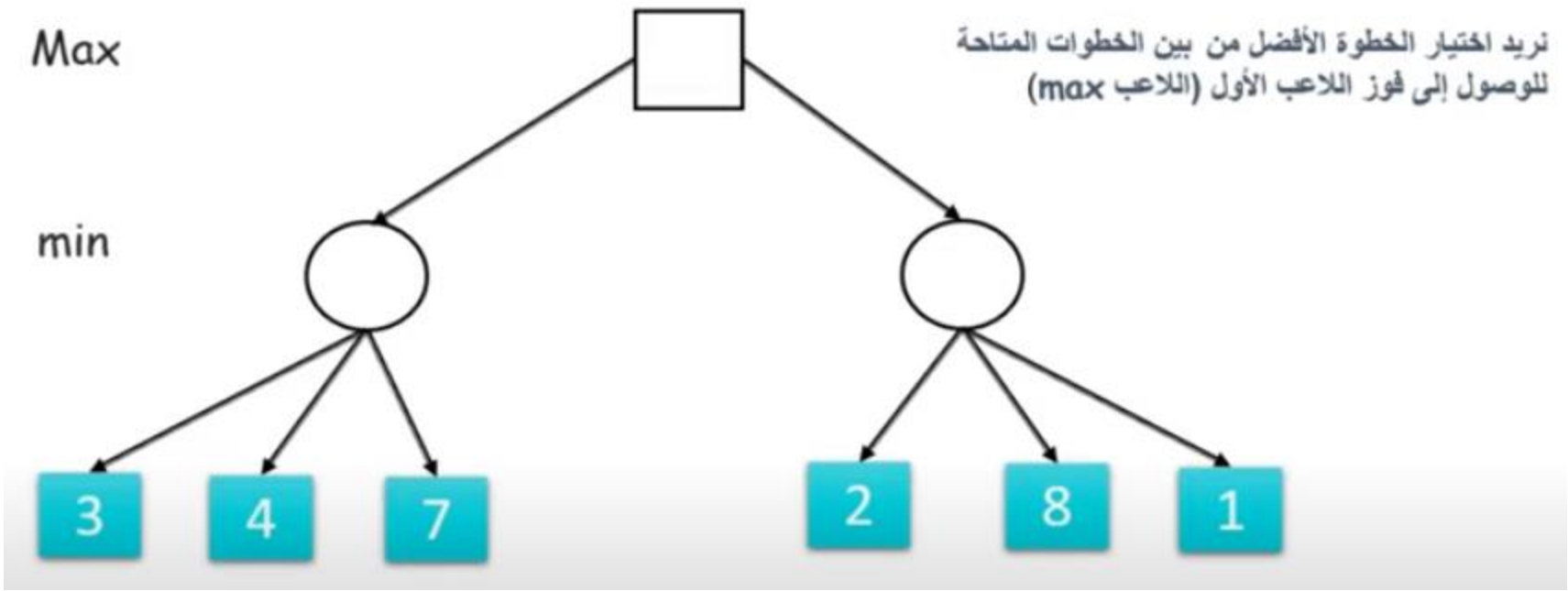
إذا وجد، أثناء اجتياز الشجرة، أن الفرع أقل واعدة من الفرع الذي تم استكشافه مسبقًا، فيمكن تجاهله بأمان، مما يوفر وقت الحساب.

- Alpha-Beta Pruning can significantly reduce the search space, making it more efficient for deeper game trees.

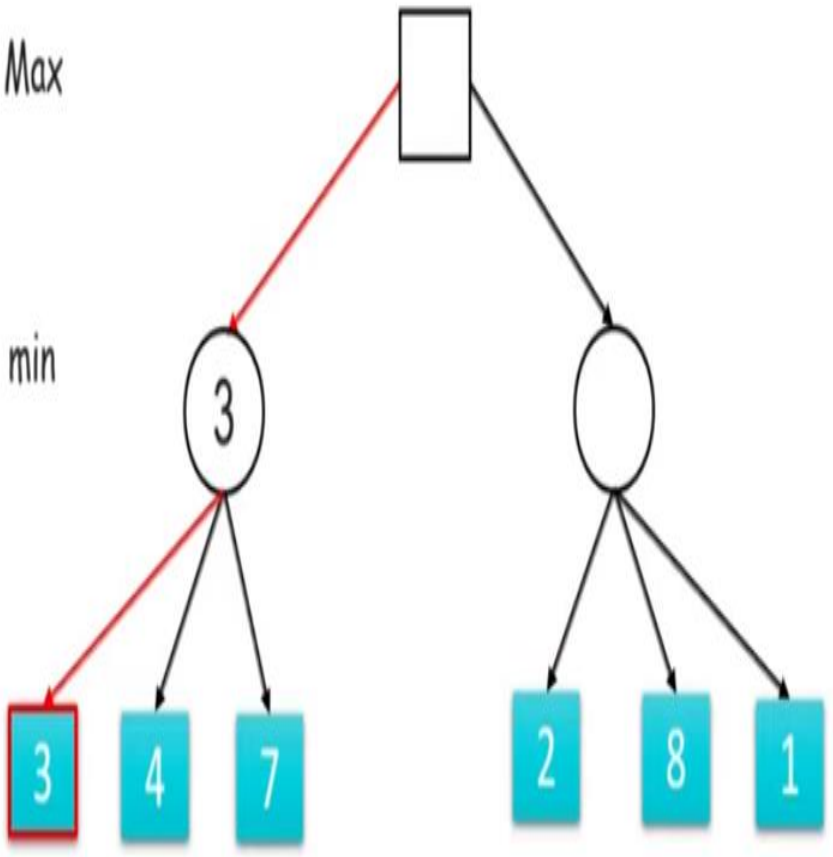
يمكن أن يؤدي تقليم Alpha-Betaإلى تقليل مساحة البحث بشكل كبير، مما يجعله أكثر كفاءة لأشجار الألعاب الأعمق.

ملاحظة ترتبط قيمة الفا مع اللاعب MAX وترتبط قيمة بيتا مع اللعب MIN.

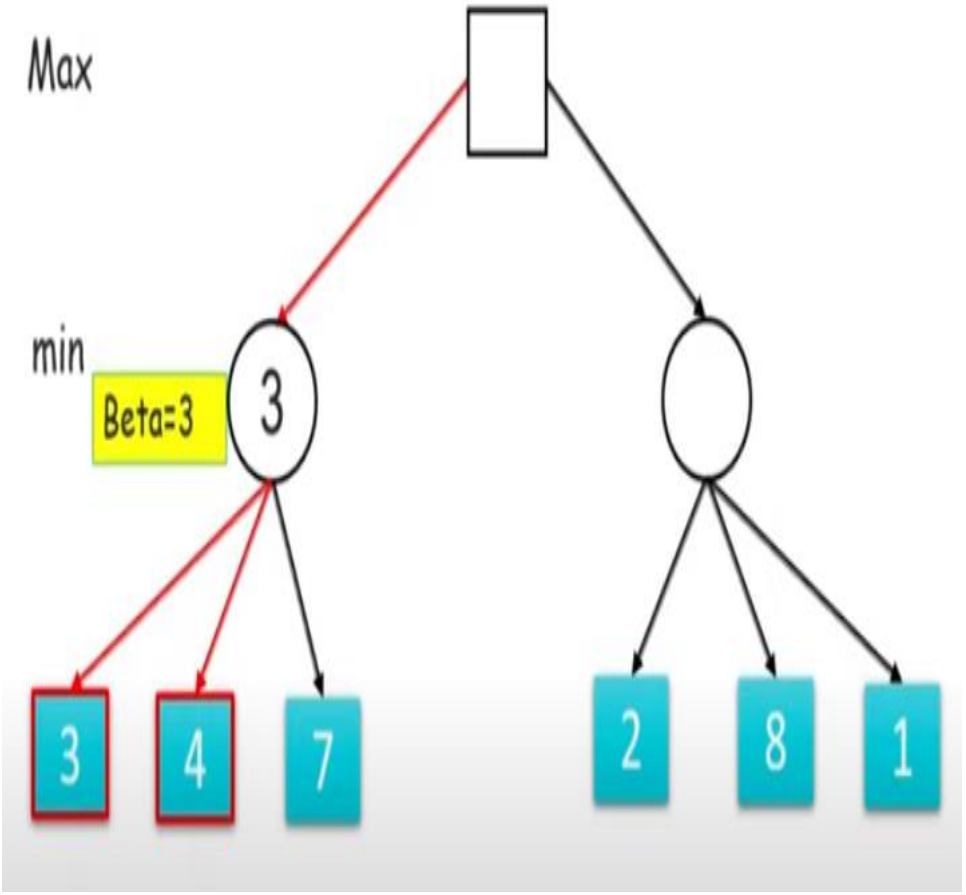
Alpha Beta algorithm خوارزميات البحث الذكية



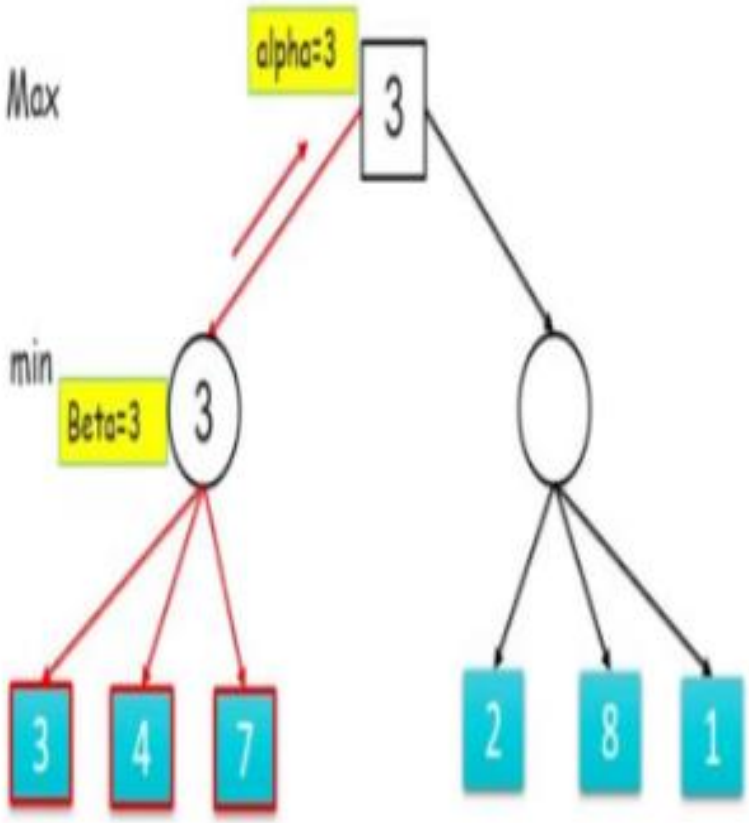
1- نتجول في الشجرة باستخدام خوارزمية DFS.
2- نعمل قطع للشجرة عندما يتحقق الشرط الفا>= بيتا



Alpha Beta algorithm خوارزميات البحث الذكية

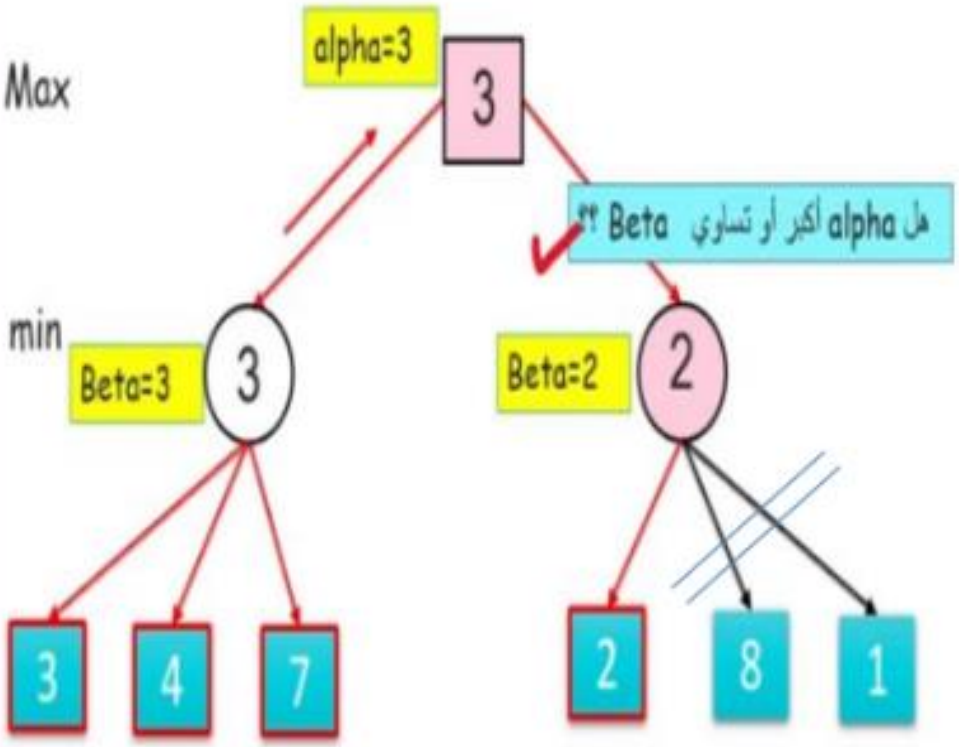


اذن قيمة بيتا = 3 لانني اقف عند اللاعب MIN .
انظر الى الابن الثاني لهذه النود وهو 4. اقارن بين 4 و 3 واختار الاصغر لانني اقف على MIN اذن اختار 3. اذهب الى الابن الثالث لهذه النود وهو 7 واختار 3.



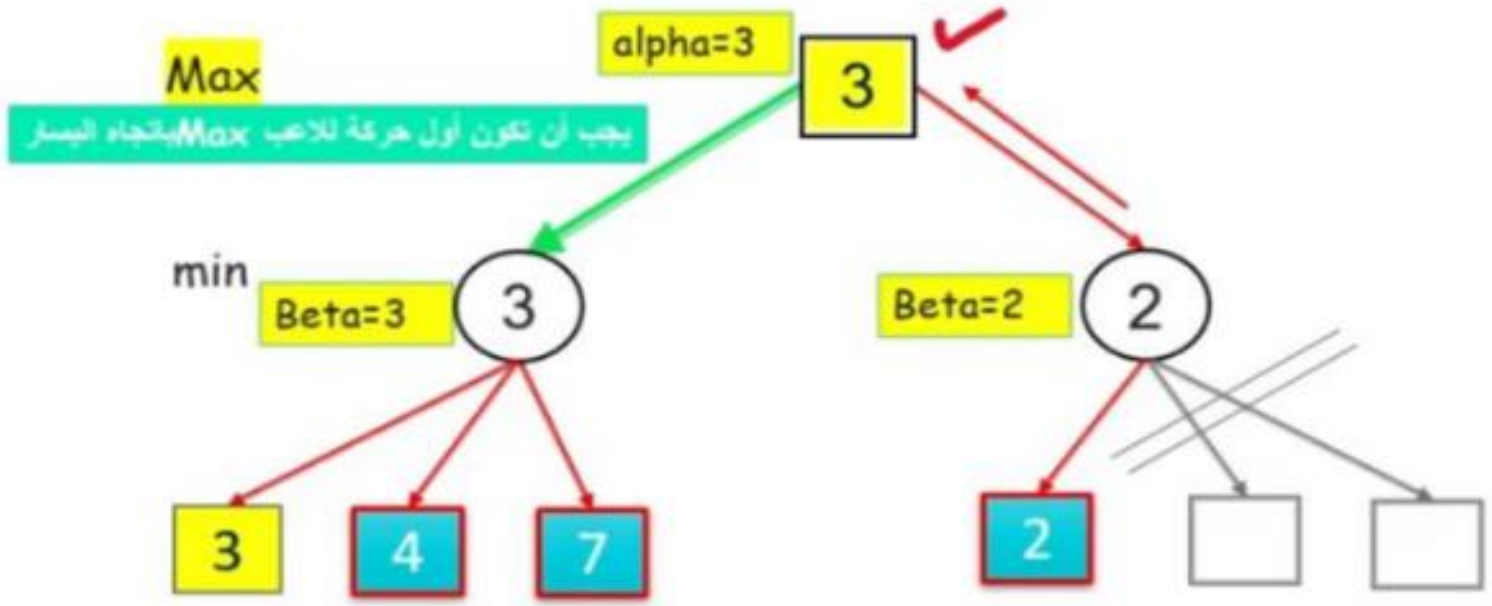
احمل ال 3 واتجه بها الى الاعلى. اصبحت قيمة الفا =3 لانني اقف عند ال MAX. ثم اتجه الى الفرع الثاني من الشجرة واتجه الى اقصى العمق اقصى اليسار.

Alpha Beta algorithm خوارزميات البحث الذكية

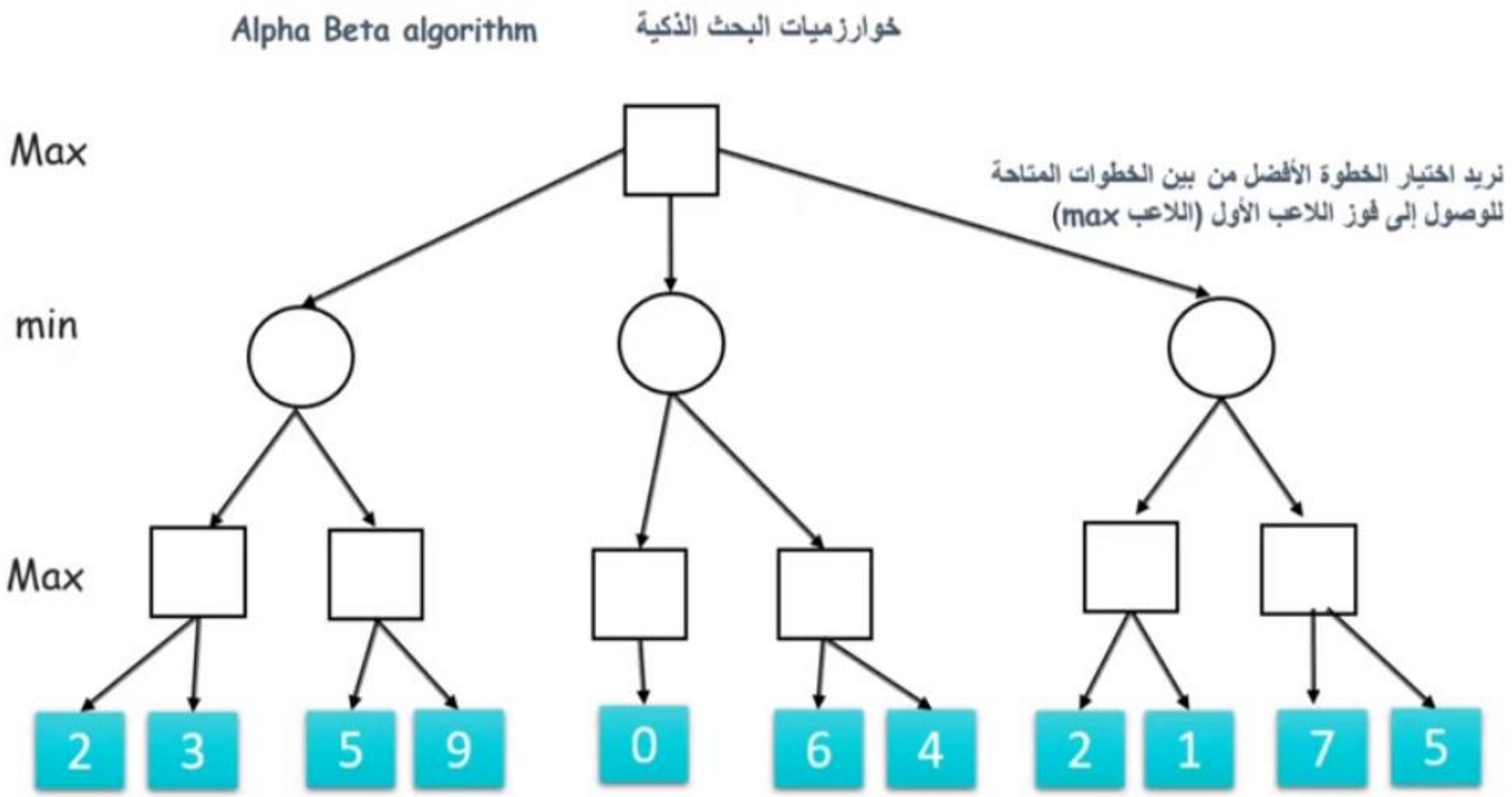


احمل ال 2 واصعد بها الى الاعلى. اصبحت قيمة بيتا =2 . الان افكر هل تحقق الشرط الفا اكبر او تساوي بيتا ؟ نعم .
اذن اعمل قطع للشجرة ولا داعي لمعرفة قيم الابن الثاني والثالث

Alpha Beta algorithm خوارزميات البحث الذكية

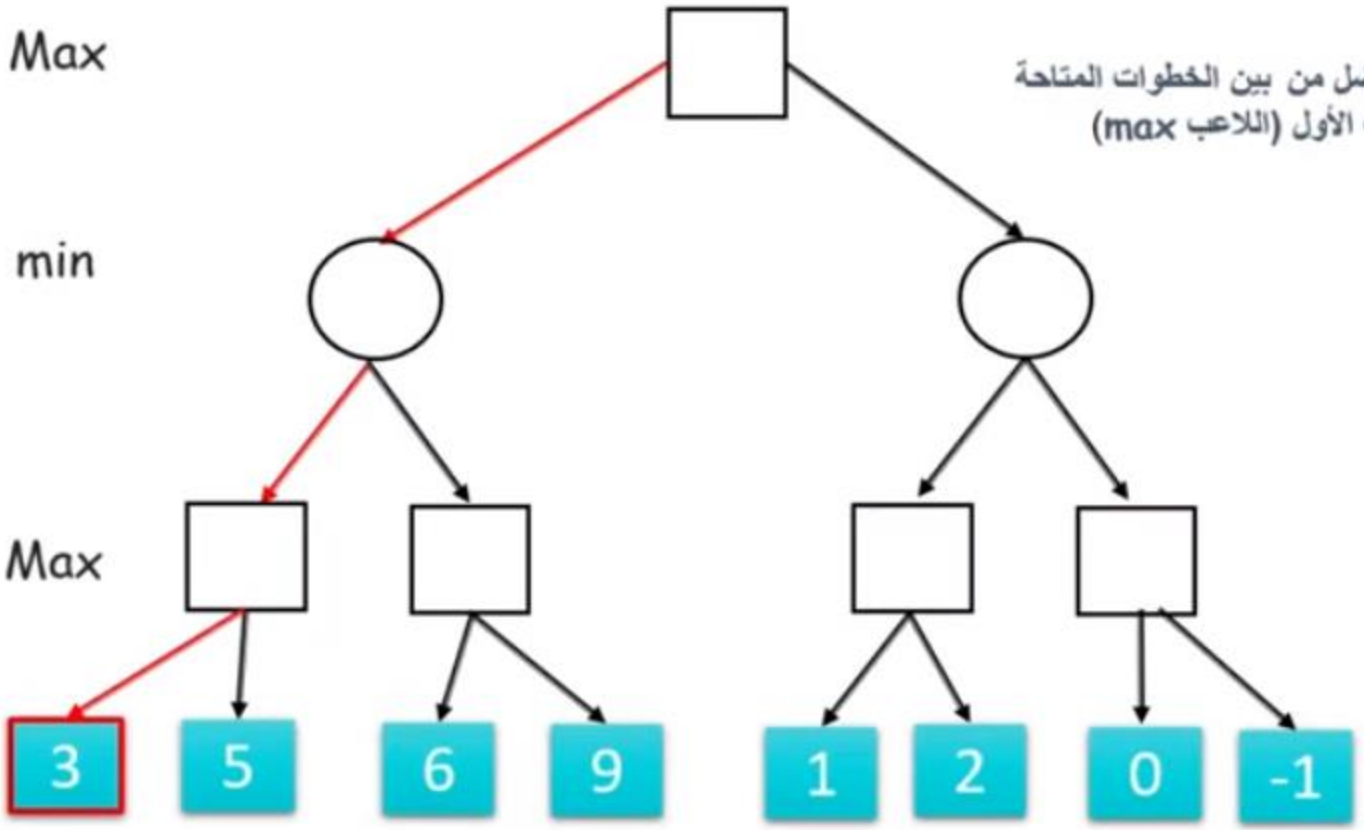


HOME WORK#4


Alpha Beta algorithm
خوارزميات البحث الذكية
Max
min
Max
نريد اختيار الخطوة الأفضل من بين الخطوات المتاحة
للوصول إلى فوز اللاعب الأول (اللاعب max)
2
3
5
9
0
6
4
2
1
7
5

HOME WORK#5

Alpha Beta algorithm خوارزميات البحث الذكية

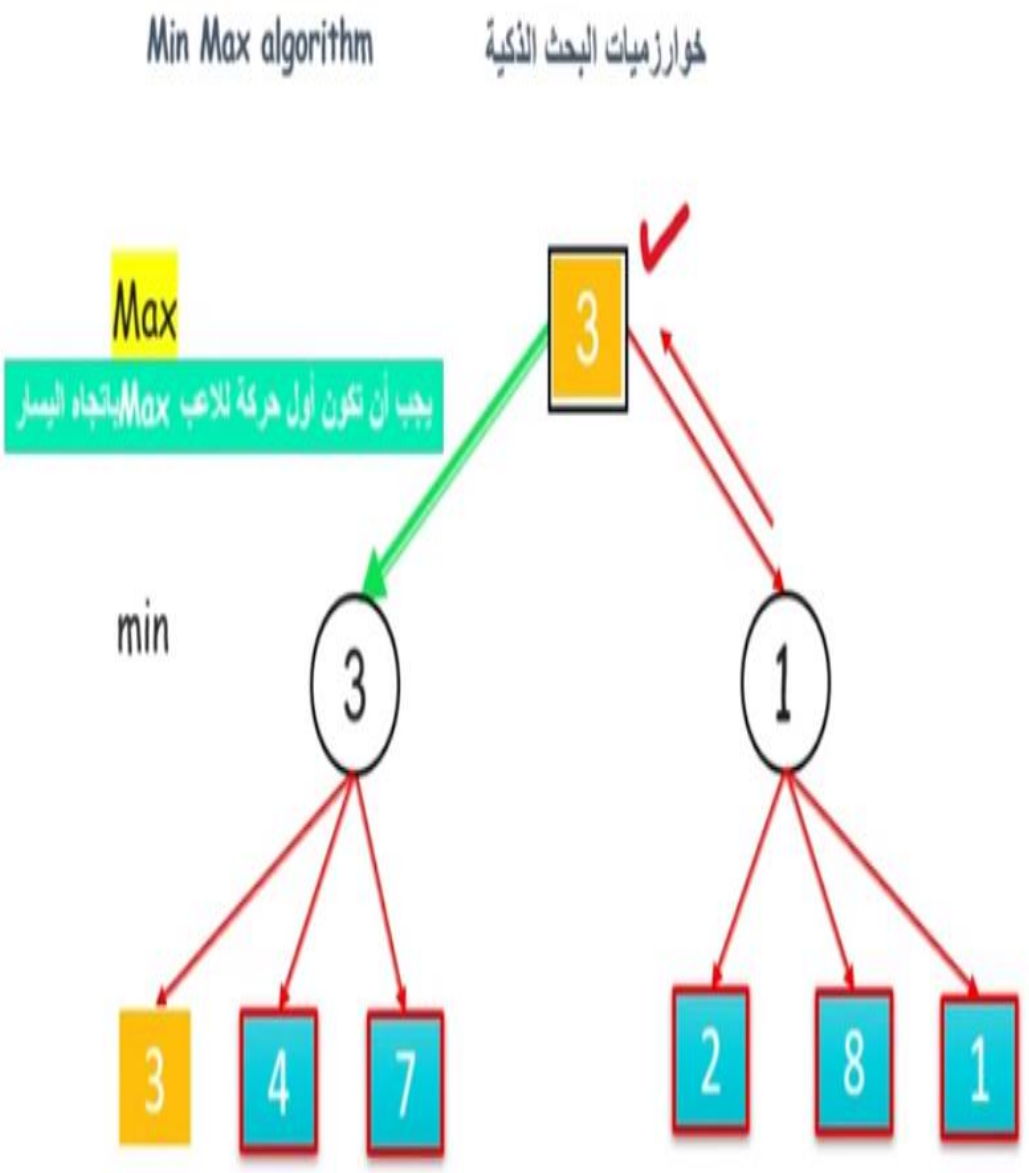


نريد اختيار الخطوة الأفضل من بين الخطوات المتاحة
للوصول إلى فوز اللاعب الأول (اللاعب max)

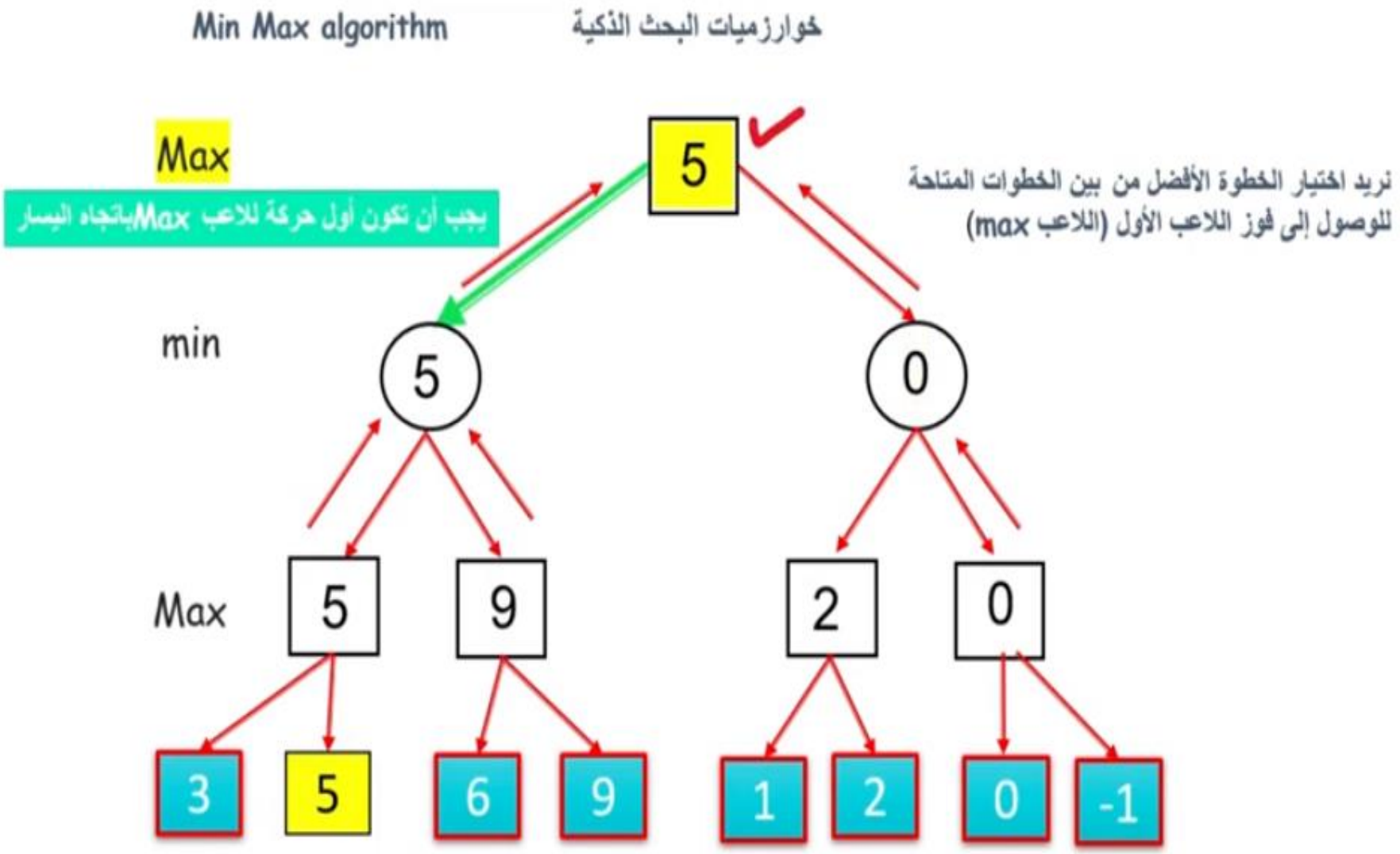
Solution of home work #1.

—	A_0
A_0	$C_4\ B_1 \rightarrow B_1\ C_4$
B_1	$C_4\ C_3\ D_4 \rightarrow C_3\ C_4\ D_4$
C_3	$C_4\ D_4\ E_8$
C_4	$D_4\ E_8\ E_9$
D_4	$E_8\ E_9\ G_8\ F_6 \rightarrow F_6\ E_8\ G_8\ E_9$
F_6	$E_8\ G_8\ E_9\ G_7 \rightarrow G_7\ E_8\ G_8\ E_9$
G_7	Goal

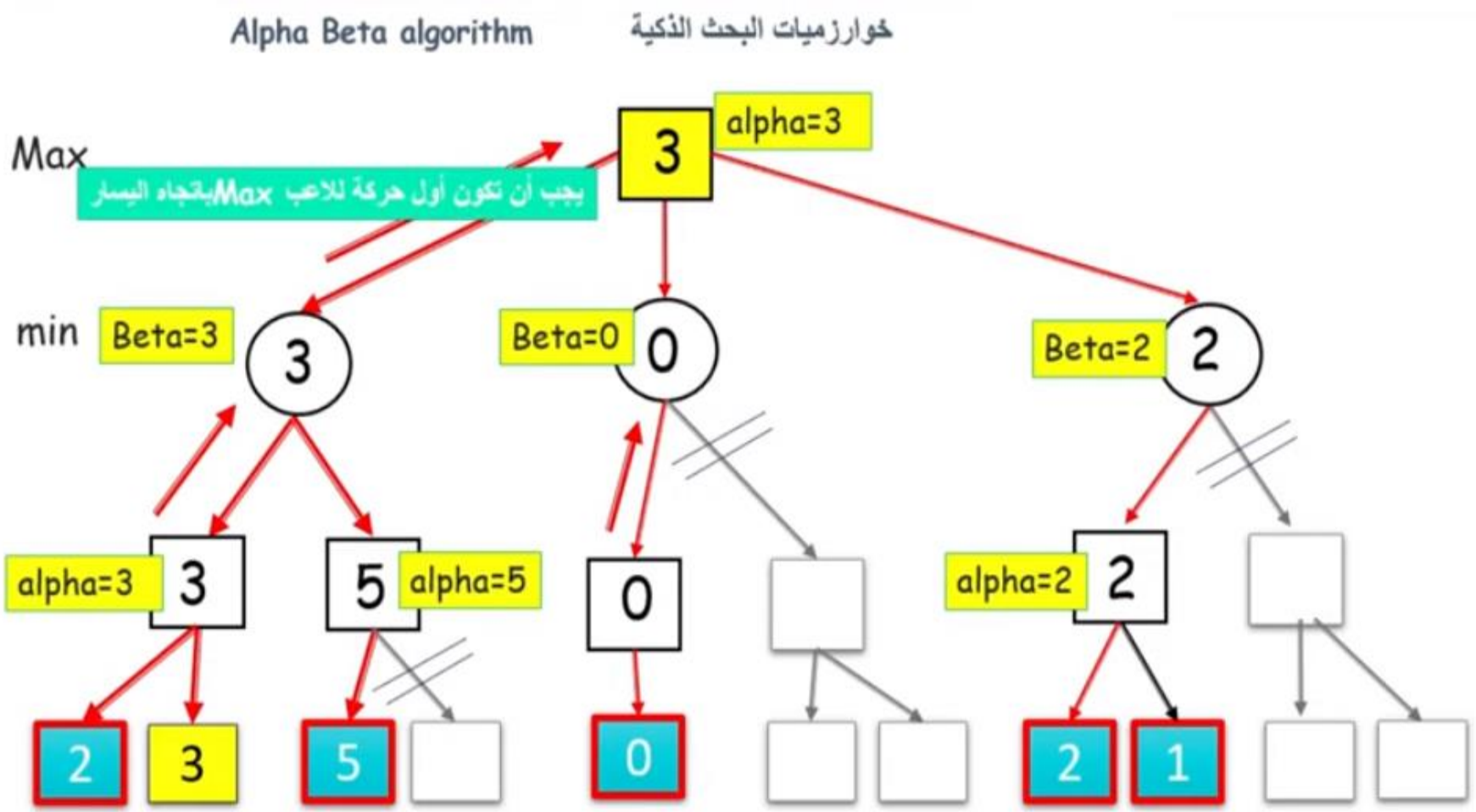
SOLUTION TO HOMEWORK#2



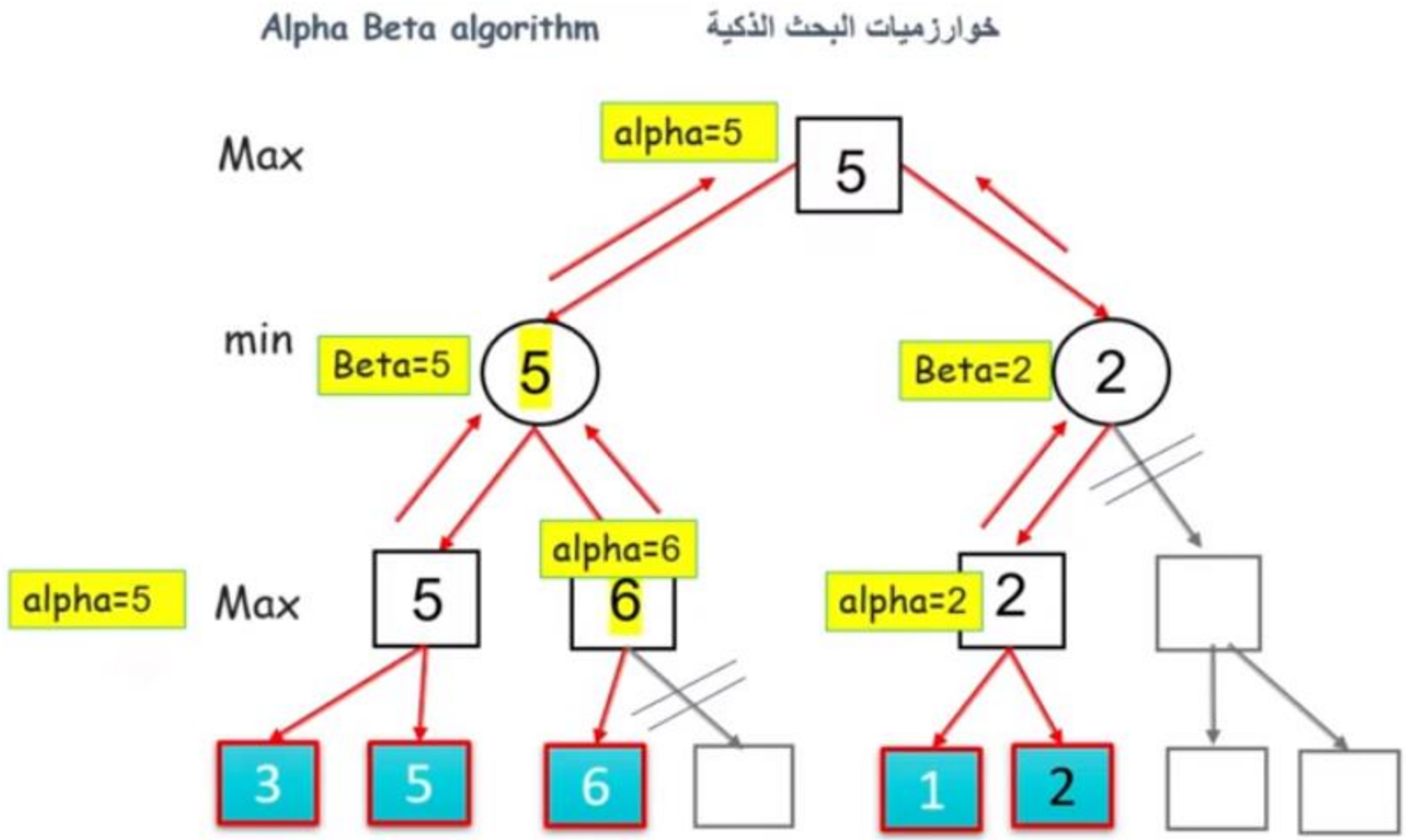
SOLUTION TO HOME WORK#3



SOLUTION TO HOME WORK#4



SOLUTION TO HOME WORK#5



اذن يجب ان تكون اول حركة للاعب MAX نحو اليسار حتى يحقق الفوز.

temp.py × untitled17.py* ×

```python
# Initialize the Tic-Tac-Toe board
board = [" " for _ in range(9)]

# Function to print the Tic-Tac-Toe board
def print_board():
    for i in range(0, 9, 3):
        print(" | ".join(board[i:i + 3]))
        if i < 6:
            print("-" * 9)

# Function to check if the game is over (win or draw)
def is_game_over():
    winning_combinations = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6),
                            (1, 4, 7), (2, 5, 8), (0, 4, 8), (2, 4, 6)]

    for combo in winning_combinations:
        if board[combo[0]] == board[combo[1]] == board[combo[2]] != " ":
            return board[combo[0]]

    if " " not in board:
        return "Draw"

    return None

# Function to take a player's move
def take_turn(player):
    while True:
        try:
            move = int(input(f"Player {player}, enter your move (1-9): ")) - 1
            if 0 <= move < 9 and board[move] == " ":
                board[move] = player
                break
            else:
                print("Invalid move. Try again.")
        except ValueError:
            print("Invalid input. Enter a number between 1 and 9.")

# Main game loop
current_player = "X"

print("Welcome to Tic-Tac-Toe!")
```

```
print_board()

while True:
    take_turn(current_player)
    print_board()

    result = is_game_over()
    if result:
        if result == "Draw":
            print("It's a draw!")
        else:
            print(f"Player {result} wins!")
        break

    current_player = "O" if current_player == "X" else "X"

```

This is the code for tic-tac-toe for 2 players.

Depth Limited Search Algorithm:

```python
def DLS(graph, source, target, limit):
    """
    Depth-limited search algorithm.

    Args:
        graph: A graph represented as a dictionary of adjacency lists.
        source: The starting node.
        target: The goal node.
        limit: The maximum depth to search.

    Returns:
        True if the target node is reachable from the source node within
        the given depth limit, False otherwise.
    """

    visited = set()

    def recursive_search(node, depth):
        if node == target:
            return True
        elif depth <= limit:
            visited.add(node)
            for neighbor in graph[node]:
                if neighbor not in visited:
                    if recursive_search(neighbor, depth + 1):
                        return True
        return False

    return recursive_search(source, 0)

```

```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}

source = 'A'
target = 'G'
limit = 2

if DLS(graph, source, target, limit):
    print('The target node is reachable from the source node within the given depth
else:
    print('The target node is not reachable from the source node within the given de
```

Here is a flowchart for the depth-limited search (DLS) algorithm in Python:

```
Start
Is the current node the target node?
Yes -> Return True
No
Is the current node visited?
Yes -> Go to the next node in the stack
No
Mark the current node as visited
Push all the neighbors of the current node onto the stack
Decrease the depth limit by 1
Is the depth limit reached?
Yes -> Return False
No -> Go to the top of the stack
End
```

Uniform Cost Search:

Here is a step-by-step explanation of how the Python code for the UCS algorithm works:

1. The algorithm starts by creating a priority queue and adding the initial node to it.
2. The algorithm then creates a set to keep track of the visited nodes.
3. The algorithm then iteratively removes the node with the highest priority from the priority queue and explores it.
4. If the node is the goal node, then the algorithm terminates and returns the path to the goal node.
5. Otherwise, the algorithm gets all of the neighboring nodes of the current node.
6. For each neighboring node, the algorithm checks if it has already been visited. If it has not been visited, then the algorithm adds it to the priority queue and marks it as visited.
7. The algorithm then repeats steps 3-6 until either the goal node is reached or the priority queue is empty.
8. If the goal node is not reached, then the algorithm returns `None`.

The UCS algorithm is a simple but effective algorithm for finding the shortest path between two nodes in a graph.

```python
from queue import PriorityQueue

class Node:
    def __init__(self, state, parent=None, cost=0):
        self.state = state
        self.parent = parent
        self.cost = cost

    def __lt__(self, other):
        return self.cost < other.cost

def UCS(graph, start_node, goal_node):
    """
    Uniform cost search algorithm.

    Args:
        graph: A graph represented as a dictionary of adjacency lists.
        start_node: The starting node.
        goal_node: The goal node.

    Returns:
        A path from the start node to the goal node, or None if no path exists.
    """

    queue = PriorityQueue()
    queue.put(Node(start_node))
    visited = set()

    while not queue.empty():
        node = queue.get()
        visited.add(node.state)

        if node.state == goal_node:
            return node

        for neighbor in graph[node.state]:
            if neighbor not in visited:
                cost = node.cost + 1
                neighbor_node = Node(neighbor, node, cost)
                queue.put(neighbor_node)

    return None
```

```
# Define a graph.
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F', 'G'],
    'D': [],
    'E': [],
    'F': [],
    'G': []
}

# Find the path from node A to node G using the UCS algorithm.
path = UCS(graph, 'A', 'G')

# Print the path.
if path is not None:
    print(path.state)
    while path.parent is not None:
        path = path.parent
        print(path.state)
else:
    print('No path found.')

```

Knowledge Representation and Reasoning
تمثيل المعرفة والتفكير

Knowledge representation is a fundamental aspect of artificial intelligence (AI) and plays a crucial role in various AI applications. It involves encoding information and knowledge in a form that can be understood and processed by computers. Logical representations are a common approach to knowledge representation, and they enable computers to reason and make inferences based on the provided knowledge.

عد تمثيل المعرفة جانبًا أساسيًا من الذكاء الاصطناعي (AI)ويلعب دورًا حاسمًا في تطبيقات الذكاء الاصطناعي المختلفة. ويتضمن ترميز المعلومات والمعرفة في شكل يمكن فهمه ومعالجته بواسطة أجهزة الكمبيوتر. التمثيلات المنطقية هي أسلوب شائع لتمثيل المعرفة، وهي تمكن أجهزة الكمبيوتر من التفكير والاستنتاجات بناءً على المعرفة المقدمة.

1. Knowledge Representation Using Logic:

- Knowledge in AI is often represented using formal logic, a systematic way to represent information and make inferences. Logic provides a structured and rigorous framework for representing and reasoning about the world.

- Logical knowledge representation typically involves using symbols, operators, and rules to represent facts, relationships, and rules in a formal way.

- It's essential for knowledge to be represented accurately to enable intelligent systems to draw meaningful conclusions.