CSE 150B Week 3 Discussion

Chia-Han Chen, Raymond Xia, Benjamin Xia

Adversarial Search

Game Tree:

- Each node represents a state of the game
- Edges emanating from a node represent possible moves that a player can make at the state.
- Each player can make a finite number of actions, and the game has finite length.
- Assumes perfect information.

A very simple example: Tic Tac Toe

Let X be the max player

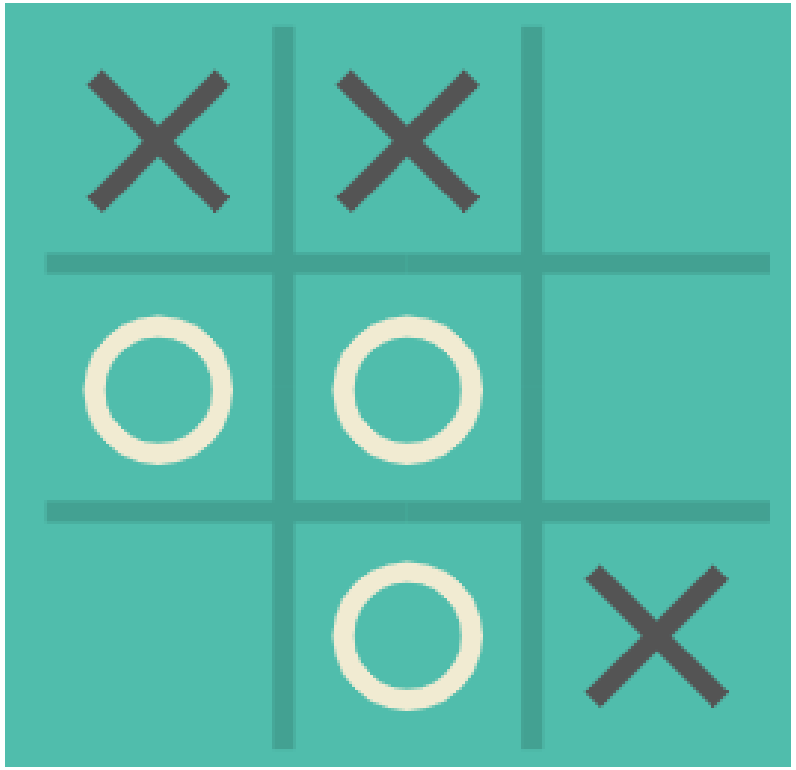
Let O be the min player

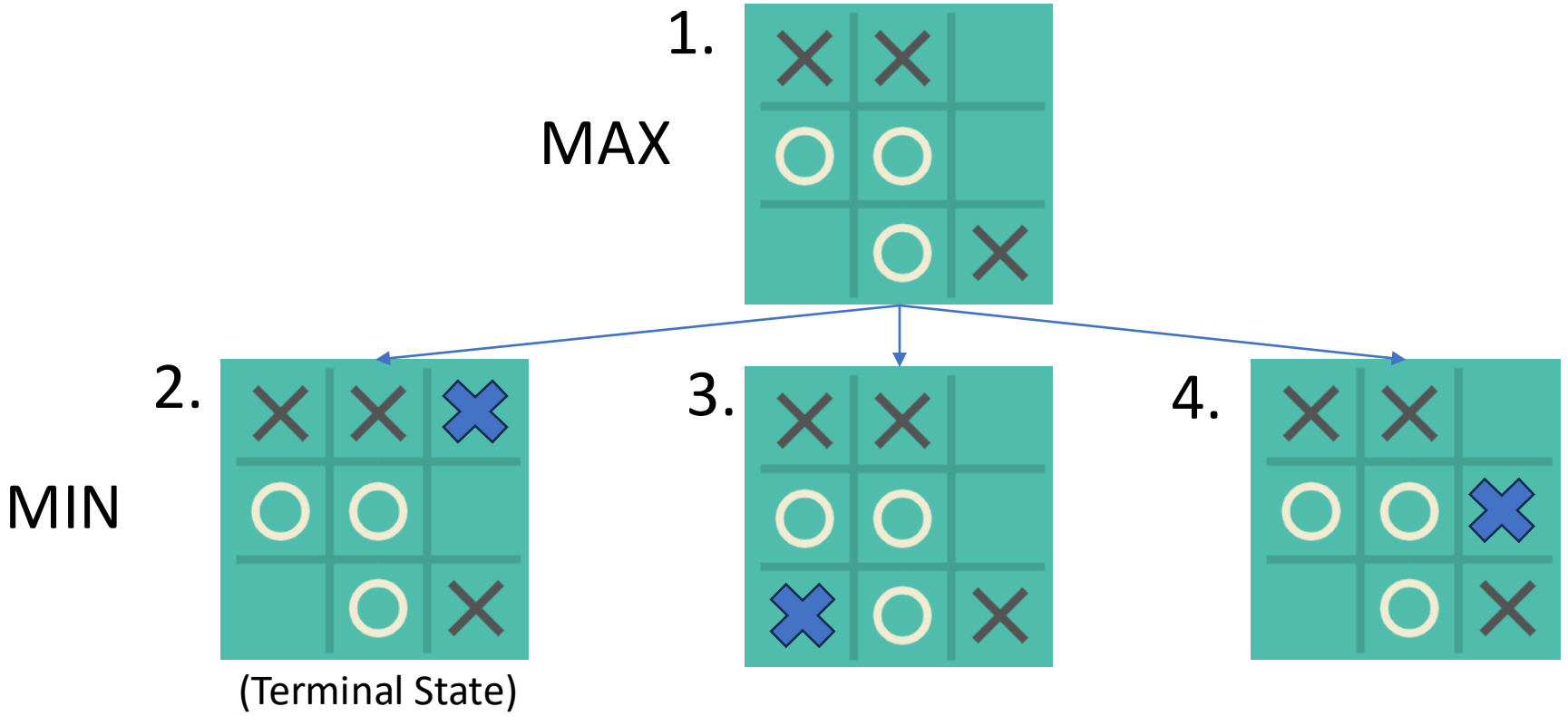
X's turn to play

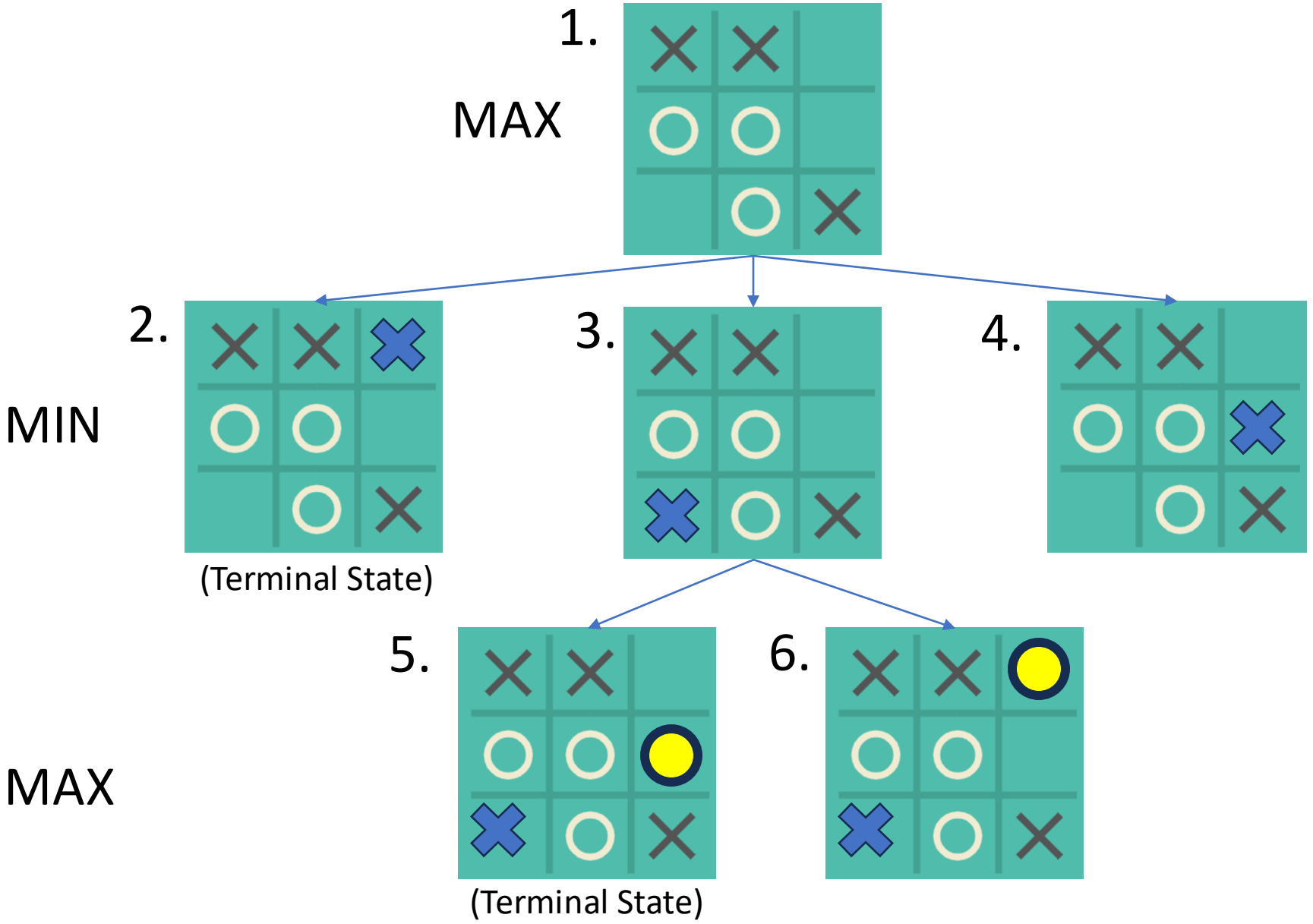
Game score = 1 if X wins

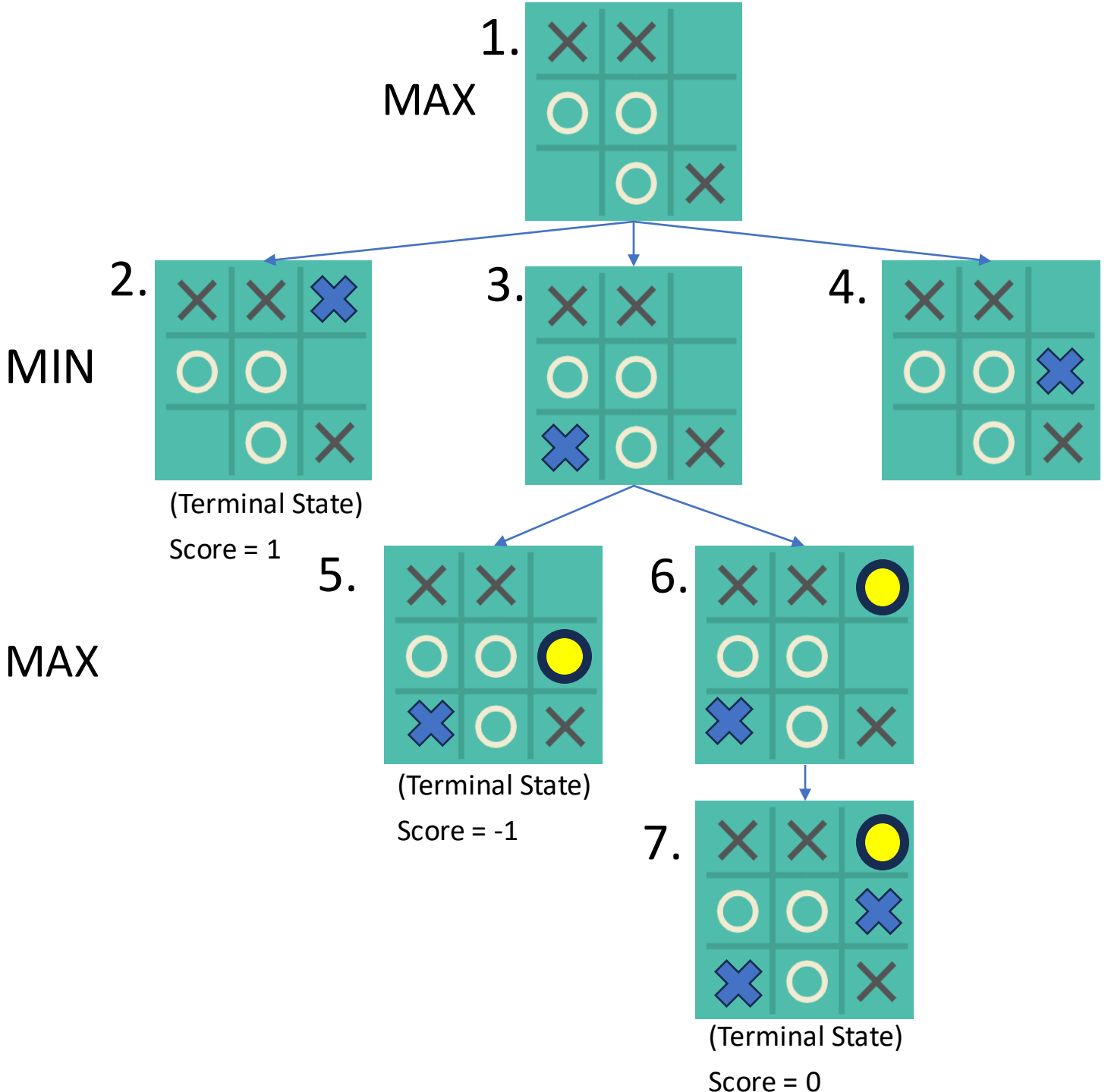
Game score = -1 if O wins

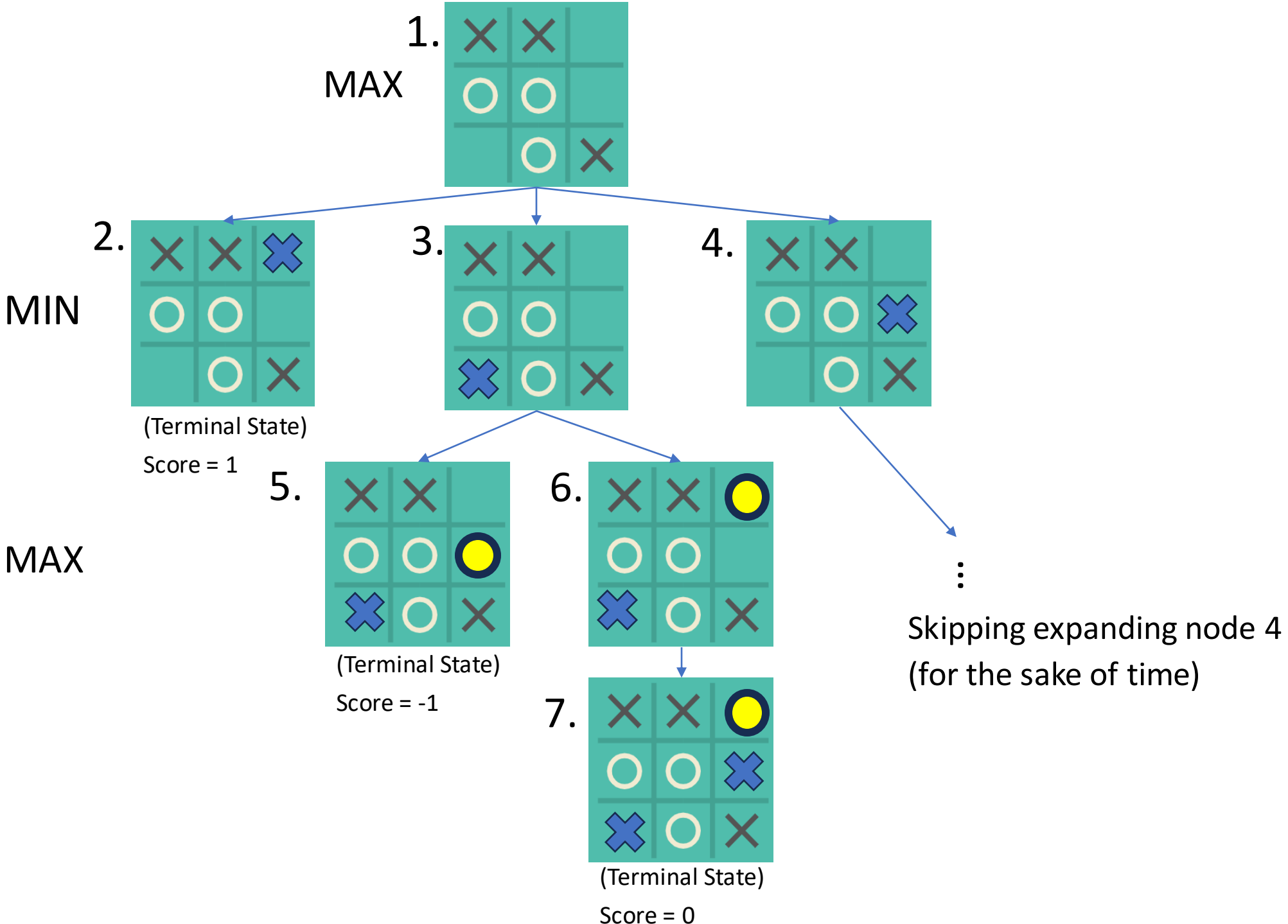
Game score = 0 otherwise

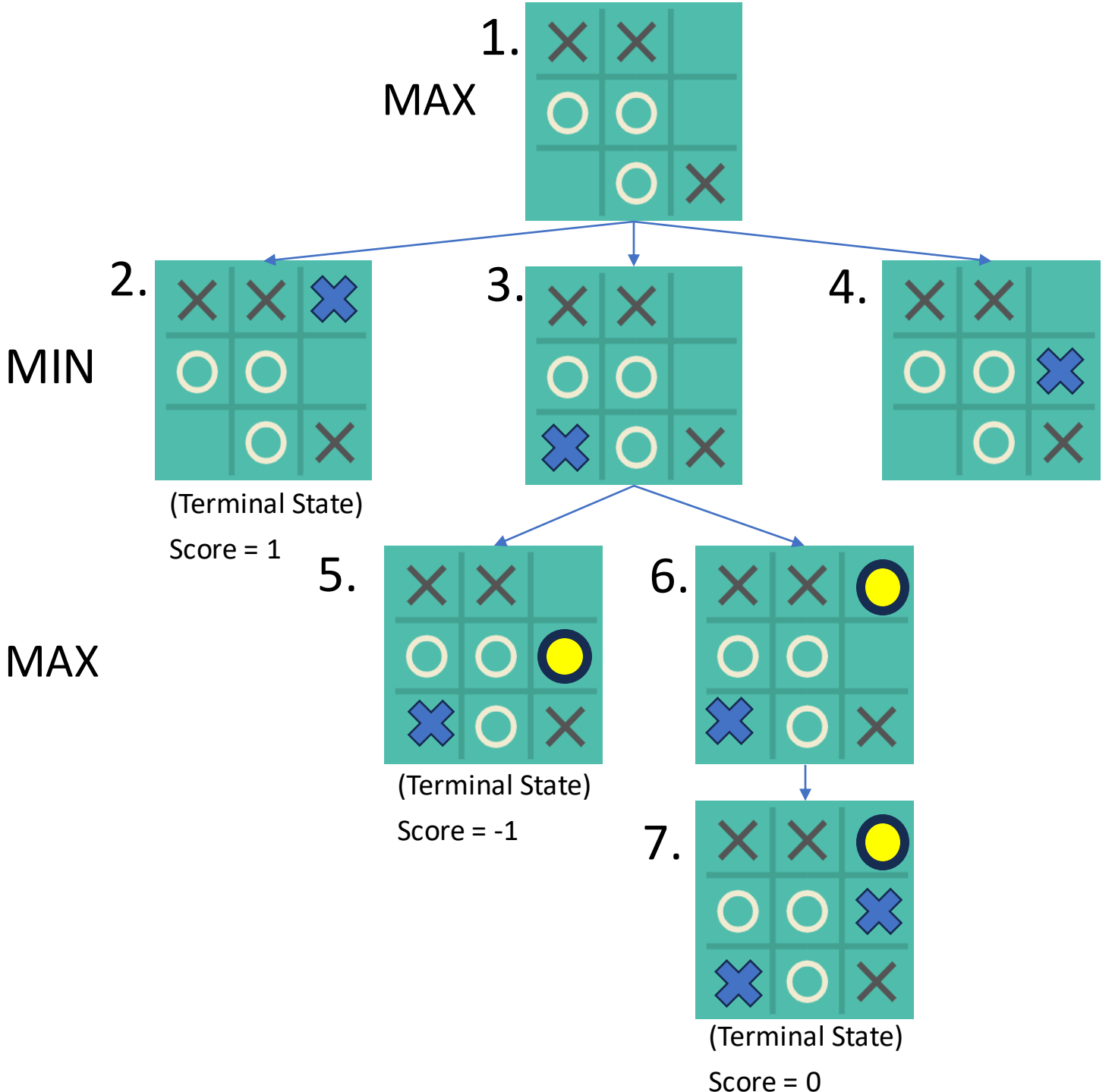



1.
MAX
2.
MIN
(Terminal State)
3.
4.


1.
MAX
2.
MIN
(Terminal State)
3.
4.
5.
MAX
(Terminal State)
6.


1.
MAX
2.
MIN
(Terminal State)
Score = 1
3.
4.
5.
MAX
(Terminal State)
Score = -1
6.
7.
(Terminal State)
Score = 0


1.
MAX
2.
MIN
(Terminal State)
Score = 1
3.
4.
5.
MAX
(Terminal State)
Score = -1
6.
7.
(Terminal State)
Score = 0
Skipping expanding node 4
(for the sake of time)



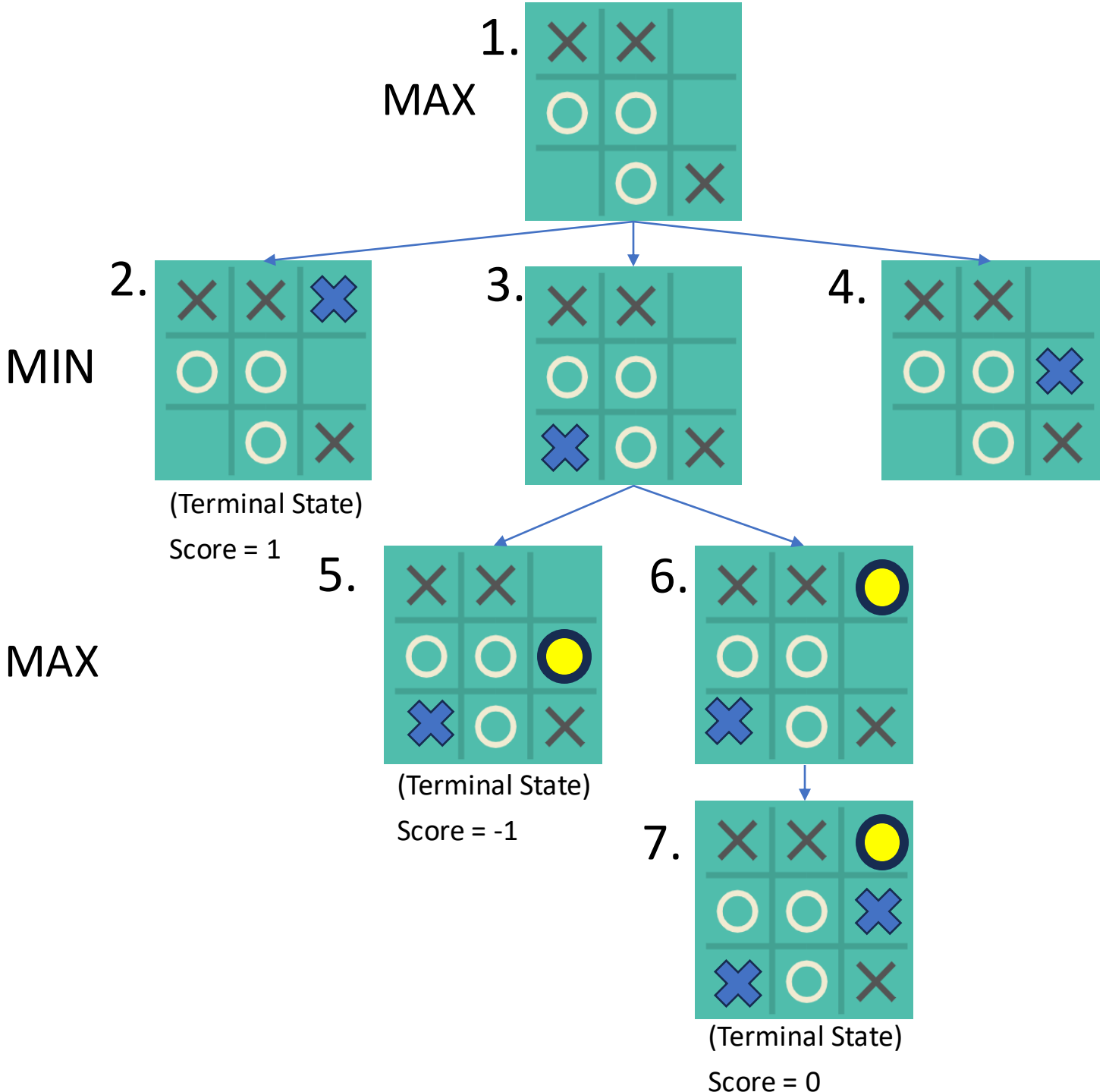
Minimax values:

Node 2: +1 (terminal node)

Node 5: -1 (terminal node)

Node 7: 0 (terminal node)

Node 6: ?



Minimax values:

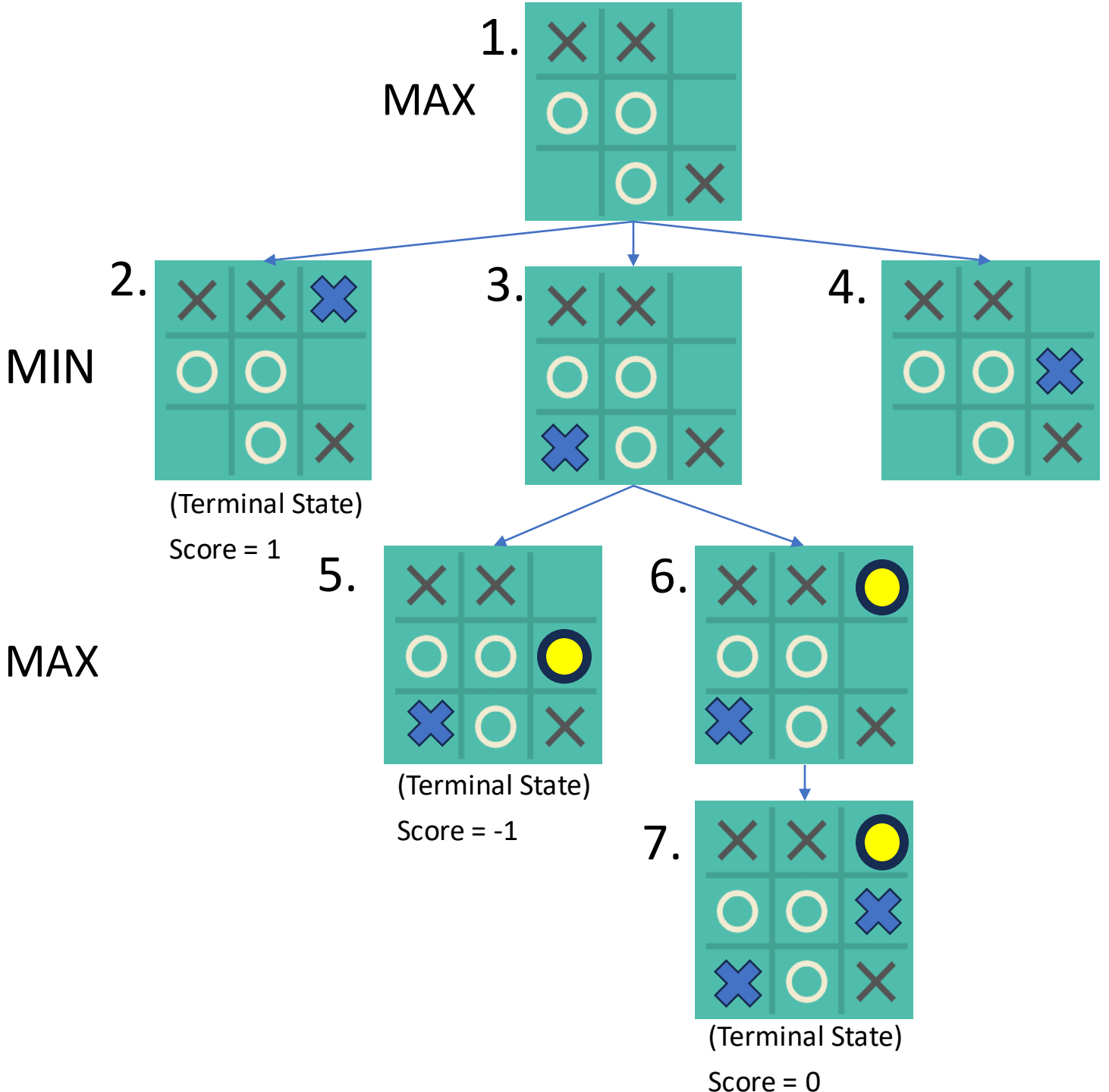
Node 2: +1 (terminal node)

Node 5: -1 (terminal node)

Node 7: 0 (terminal node)

Node 6: 0 (Max(Node 7))

Node 3: ?



Minimax values:

Node 2: +1 (terminal node)

Node 5: -1 (terminal node)

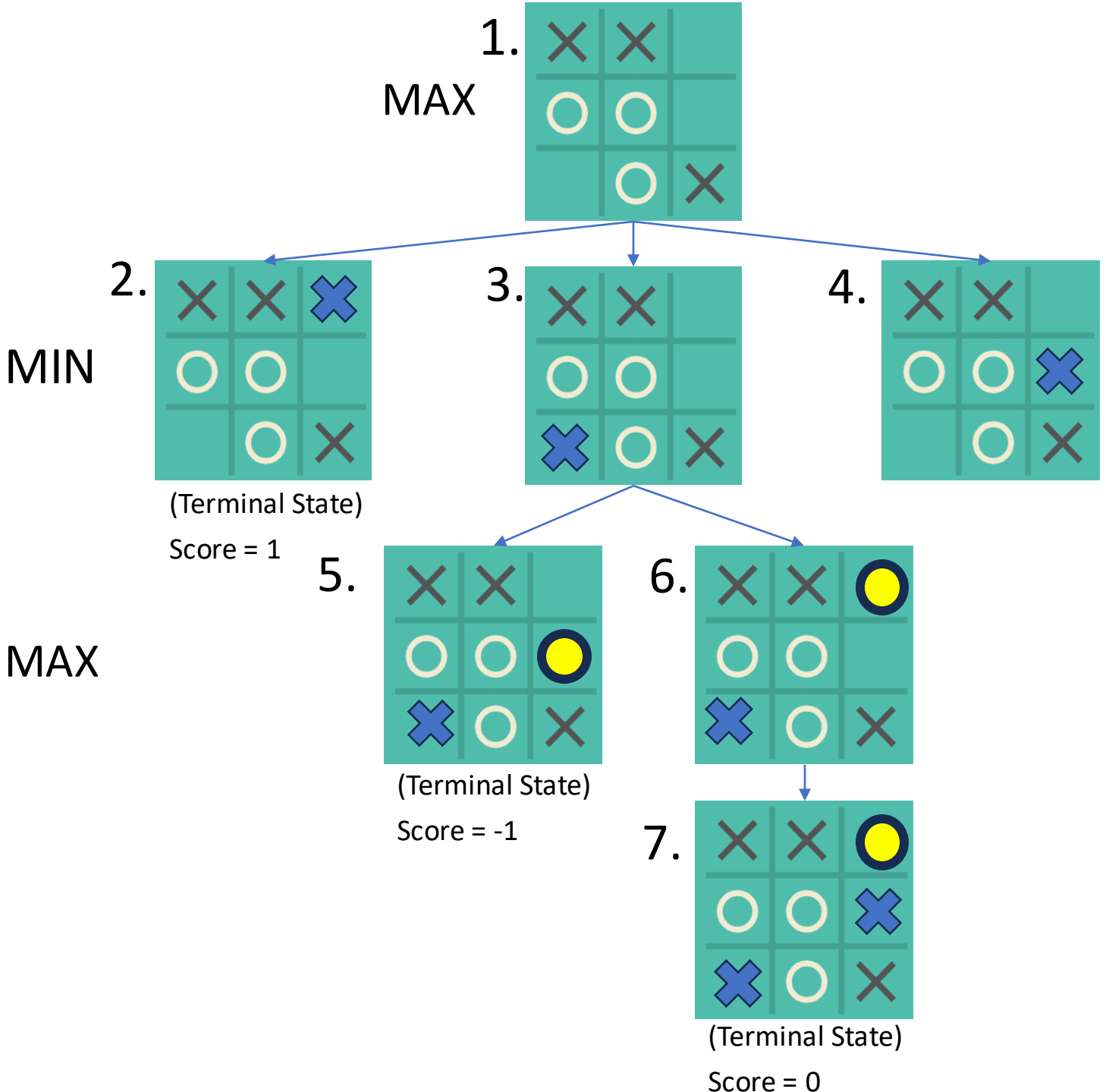
Node 7: 0 (terminal node)

Node 6: 0 (Max(Node 7))

Node 3: -1 (Min(Node 5, 6))

Node 1: ?

Hint: Minimax value of node 4 is 0



Minimax values:

Node 2: +1 (terminal node)

Node 5: -1 (terminal node)

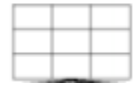
Node 7: 0 (terminal node)

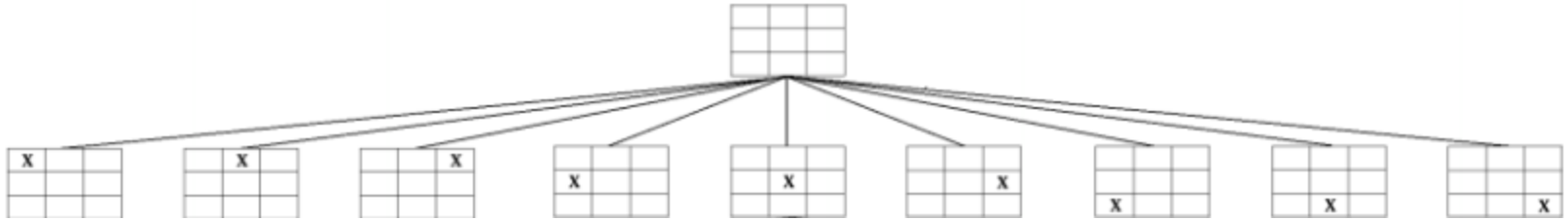
Node 6: 0 (Max(Node 7))

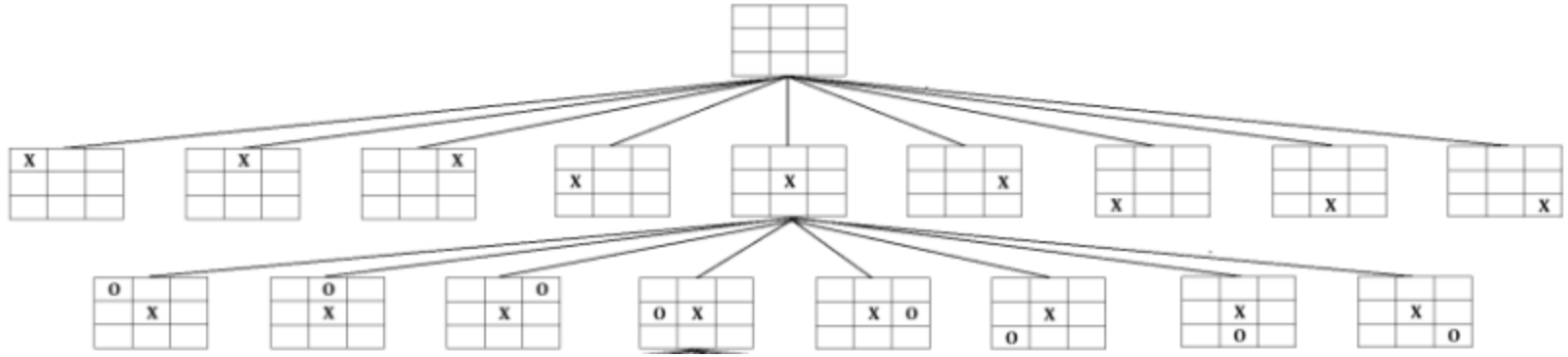
Node 3: -1 (Min(Node 5, 6))

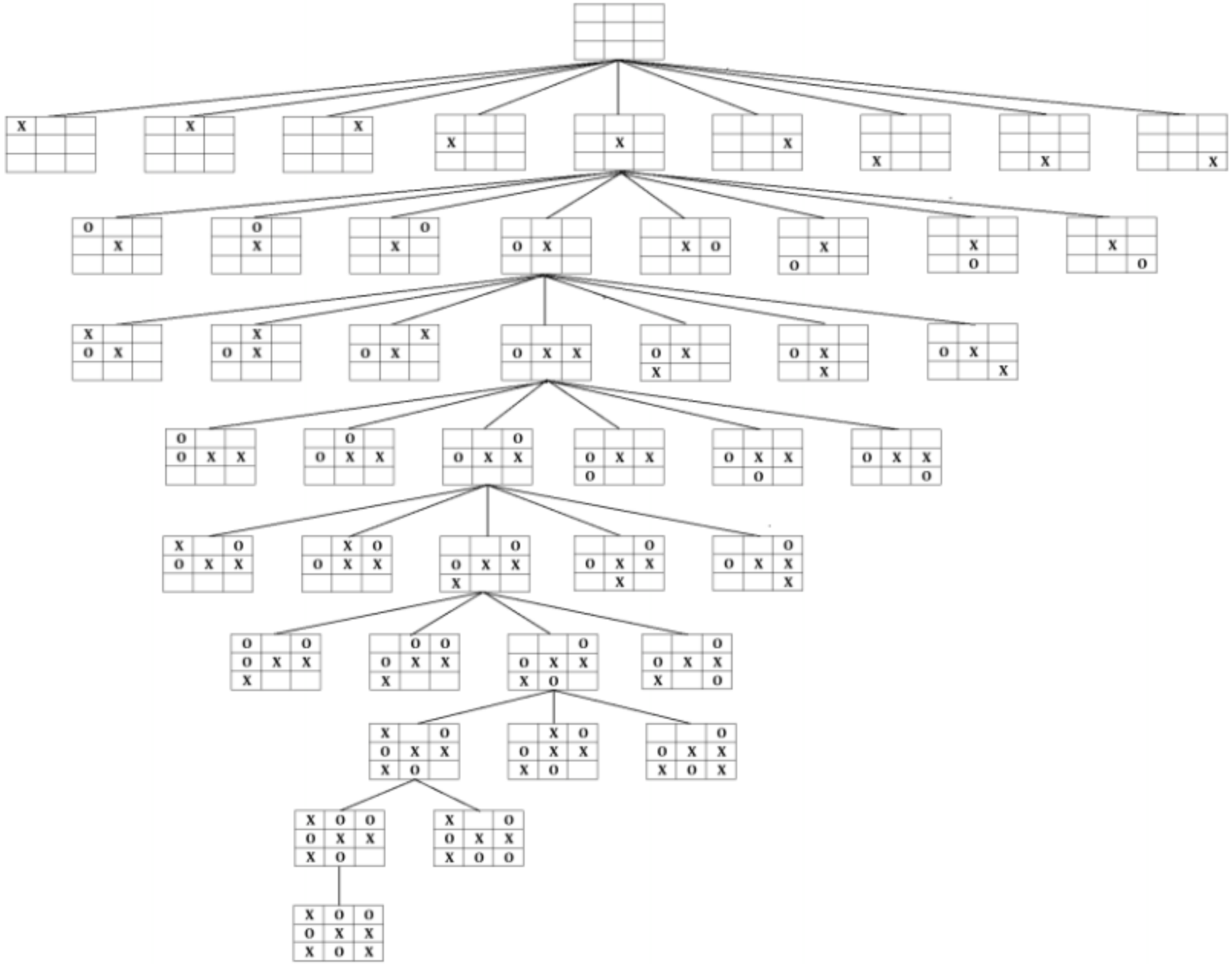
Node 4: 0

Node 1: +1 (Max(Node 2, 3, 4)







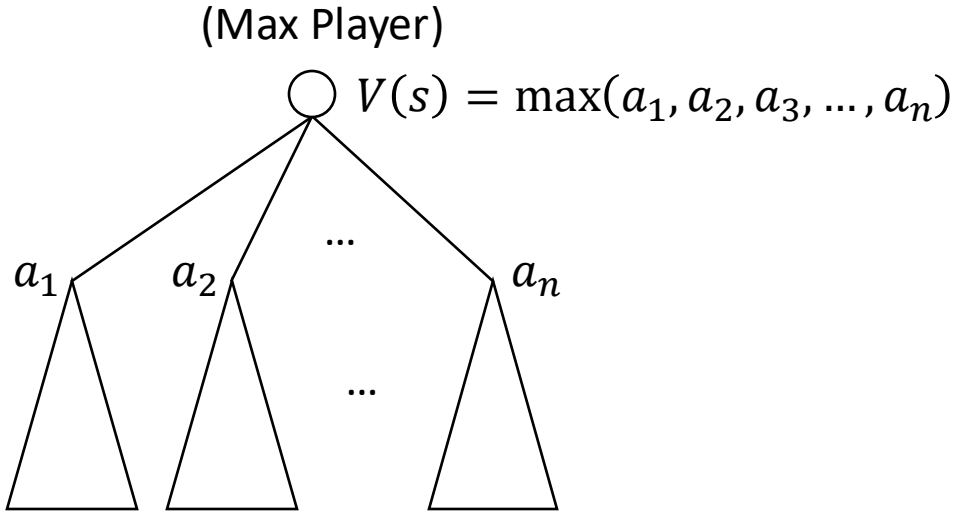


(Winner - X)

Minimax

Value of Nodes:

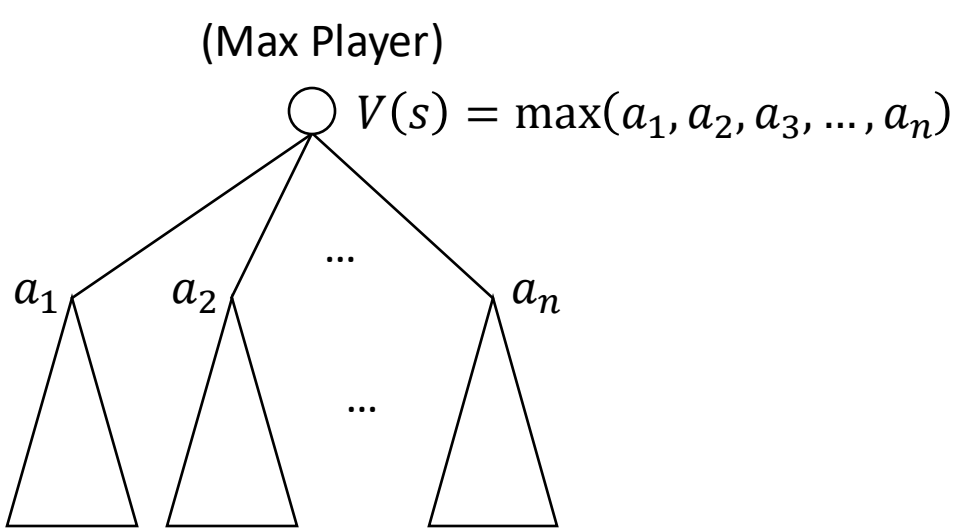
- Max Node: $\max(a_1, a_2, a_3, \dots, a_n)$

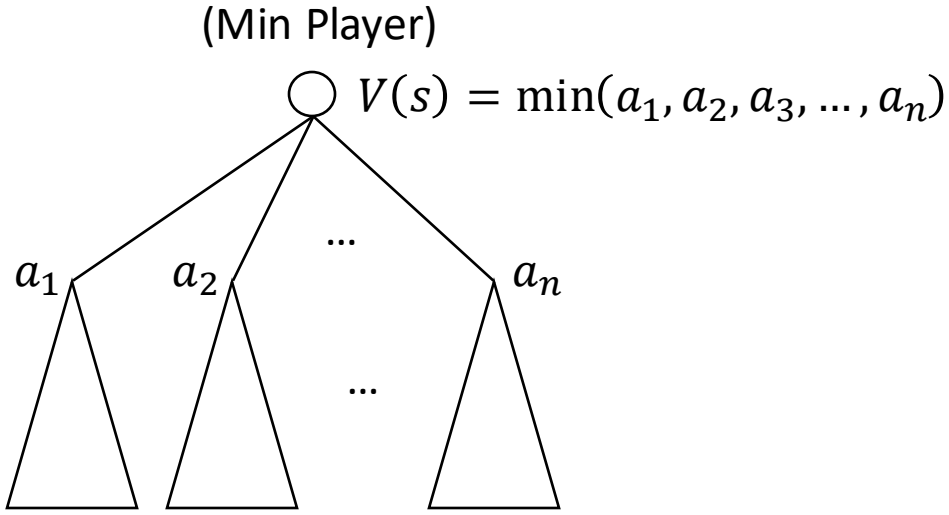


Minimax

Value of Nodes:

- Max Node: $\max(a_1, a_2, a_3, \dots, a_n)$
- Min Node: $\min(a_1, a_2, a_3, \dots, a_n)$

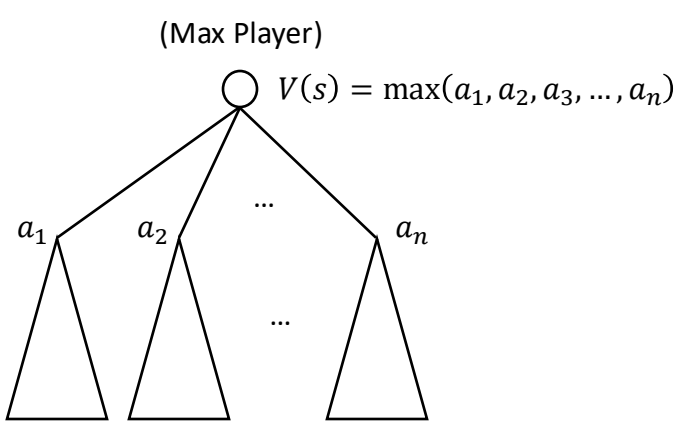


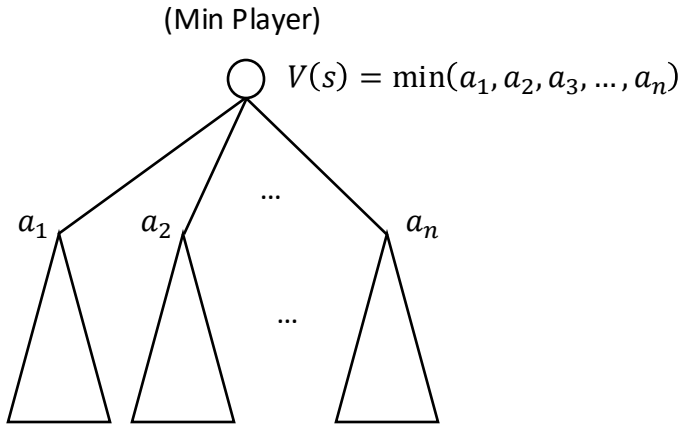


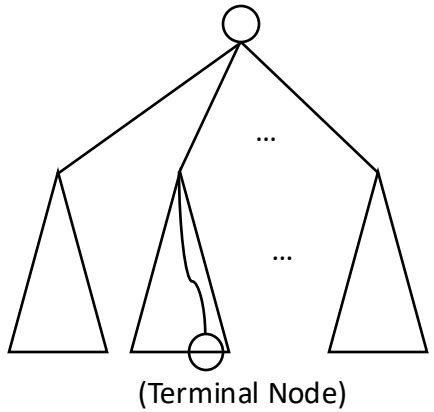
Minimax

Value of Nodes:

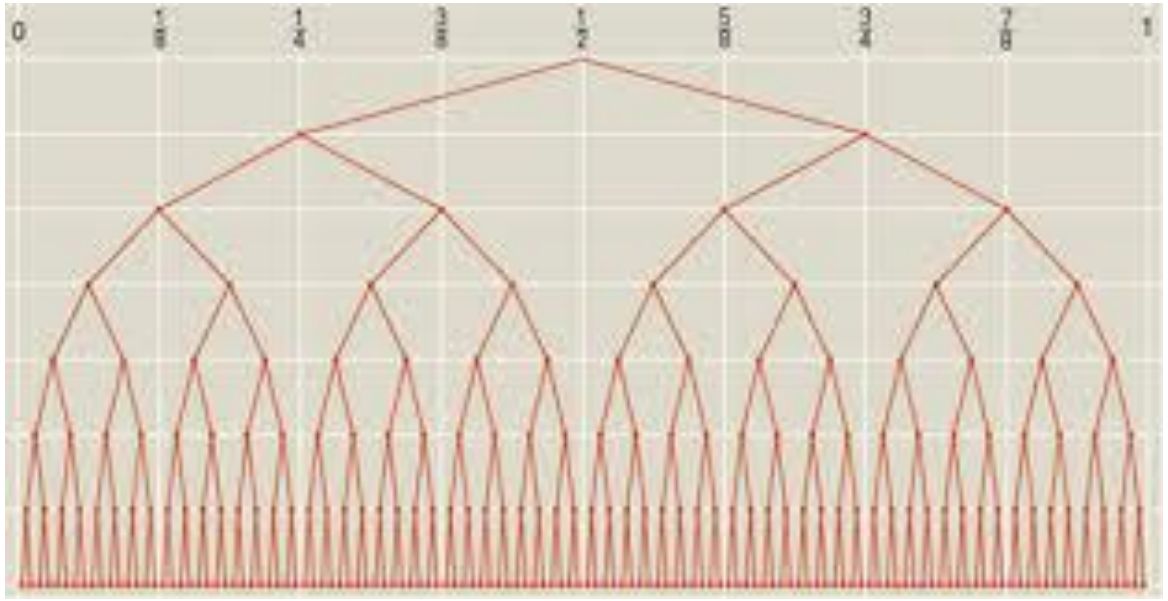
- Max Node: $\max(a_1, a_2, a_3, \dots, a_n)$
- Min Node: $\min(a_1, a_2, a_3, \dots, a_n)$
- Terminal Node (Leaf): Payoff/Outcome







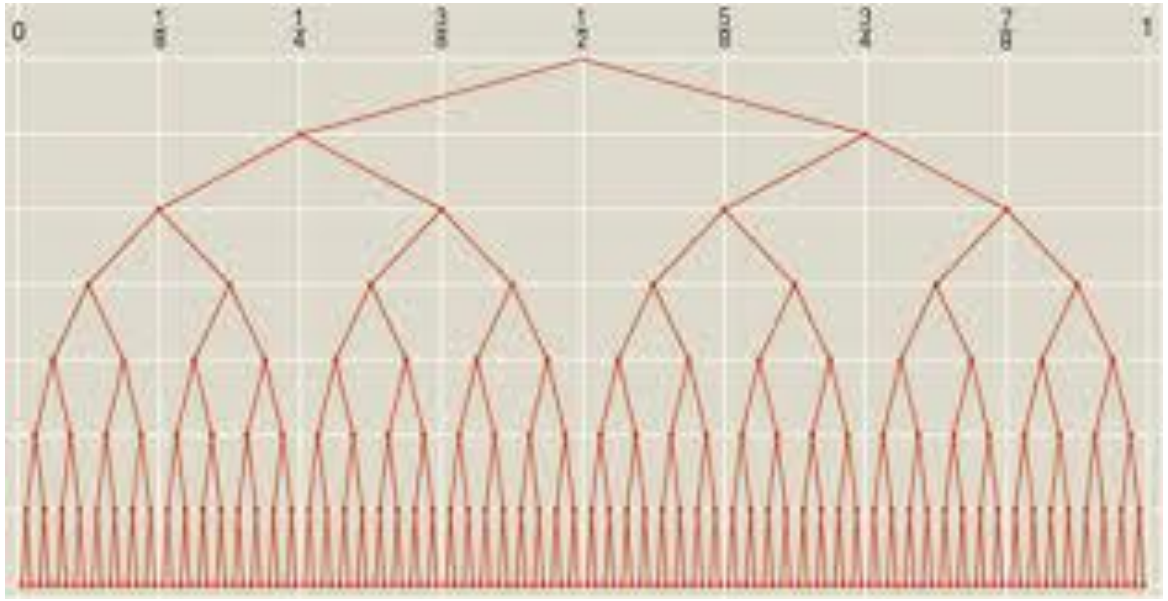
Computing Minimax: Analysis



If each node has n possible actions, how many nodes do we have to explore to compute minimax for a tree with depth d?

Recall: Depth is number of edges/actions from root node to terminal node

Computing Minimax: Analysis



If each node has n possible actions, how many nodes do we have to explore to compute minimax for a tree with depth d?

$$1 + n + n^2 + n^3 + n^4 \ldots = \sum_{i=0}^{d} n^i = \frac{n^{d+1}-1}{n-1}$$

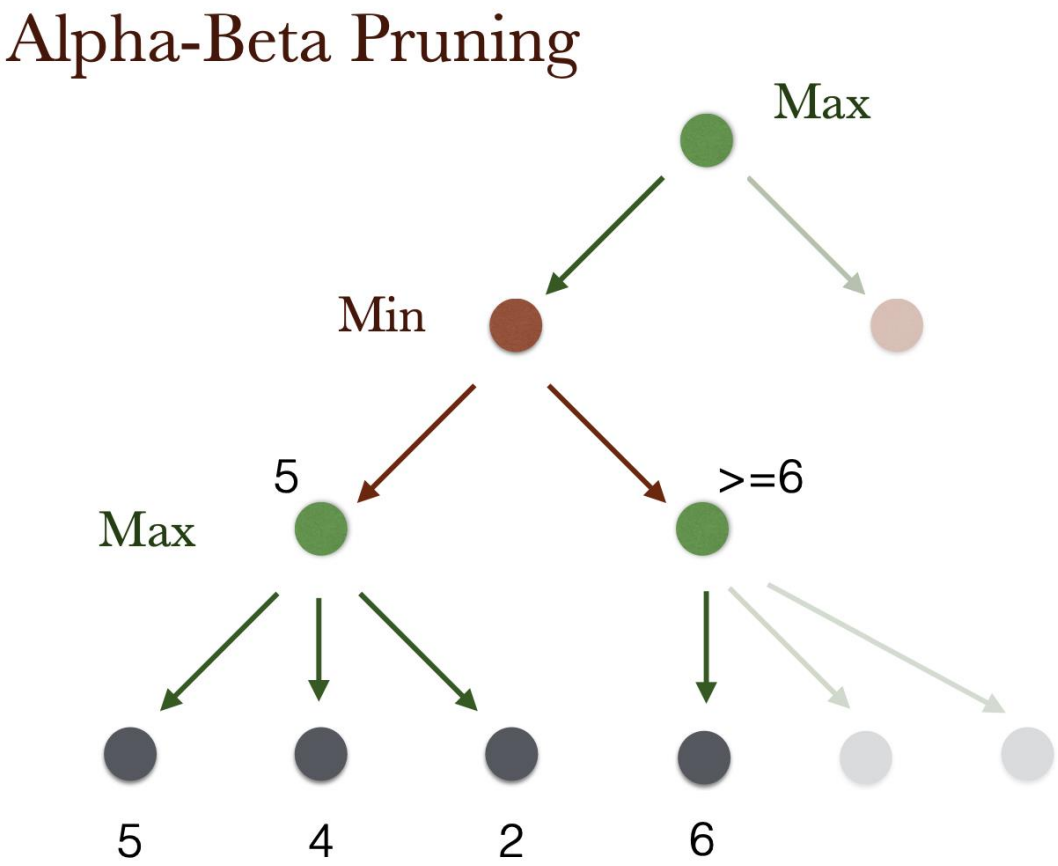
In practice: Use alpha-beta pruning (Not applicable in expectimax) or heuristic after a certain depth.

Recall: Depth is number of edges/actions from root node to terminal node

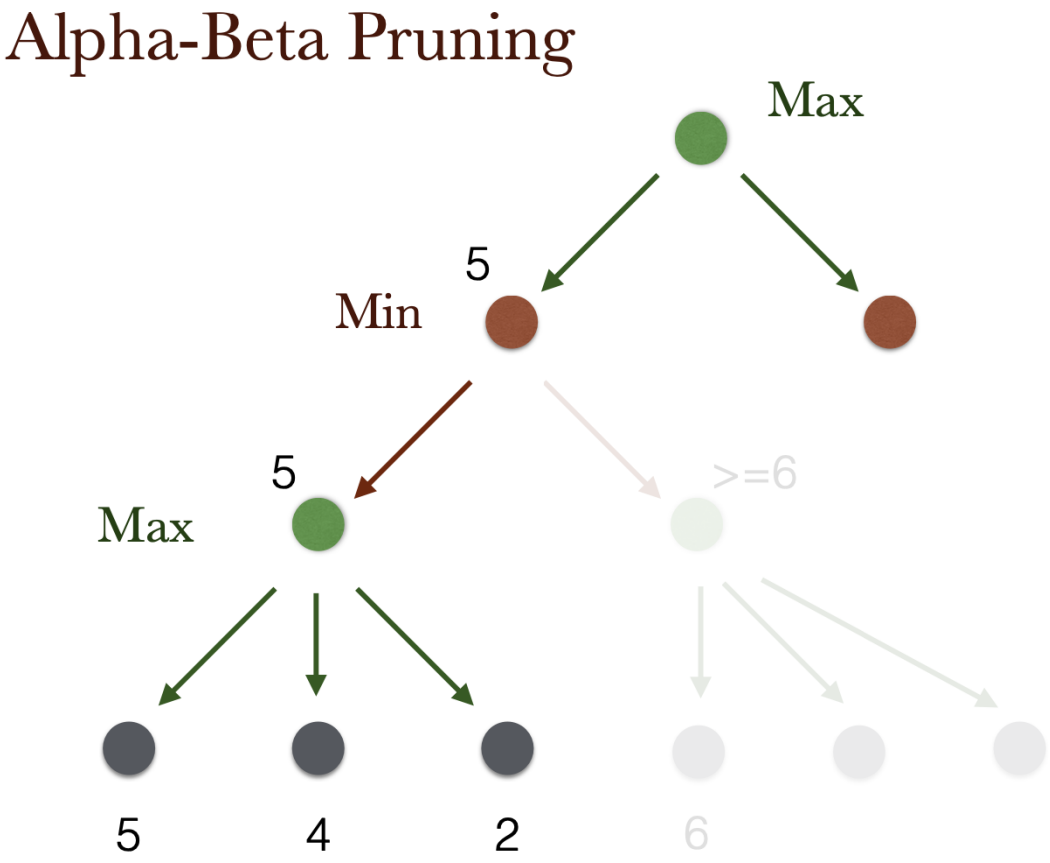
Pruning

- Horizontal: Alpha-Beta Pruning
- Vertical: Cut-off at some fixed depth and use heuristics to evaluate the intermediate nodes.

Alpha – Beta Pruning



Alpha – Beta Pruning



Alpha – Beta Pruning

```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

First call:

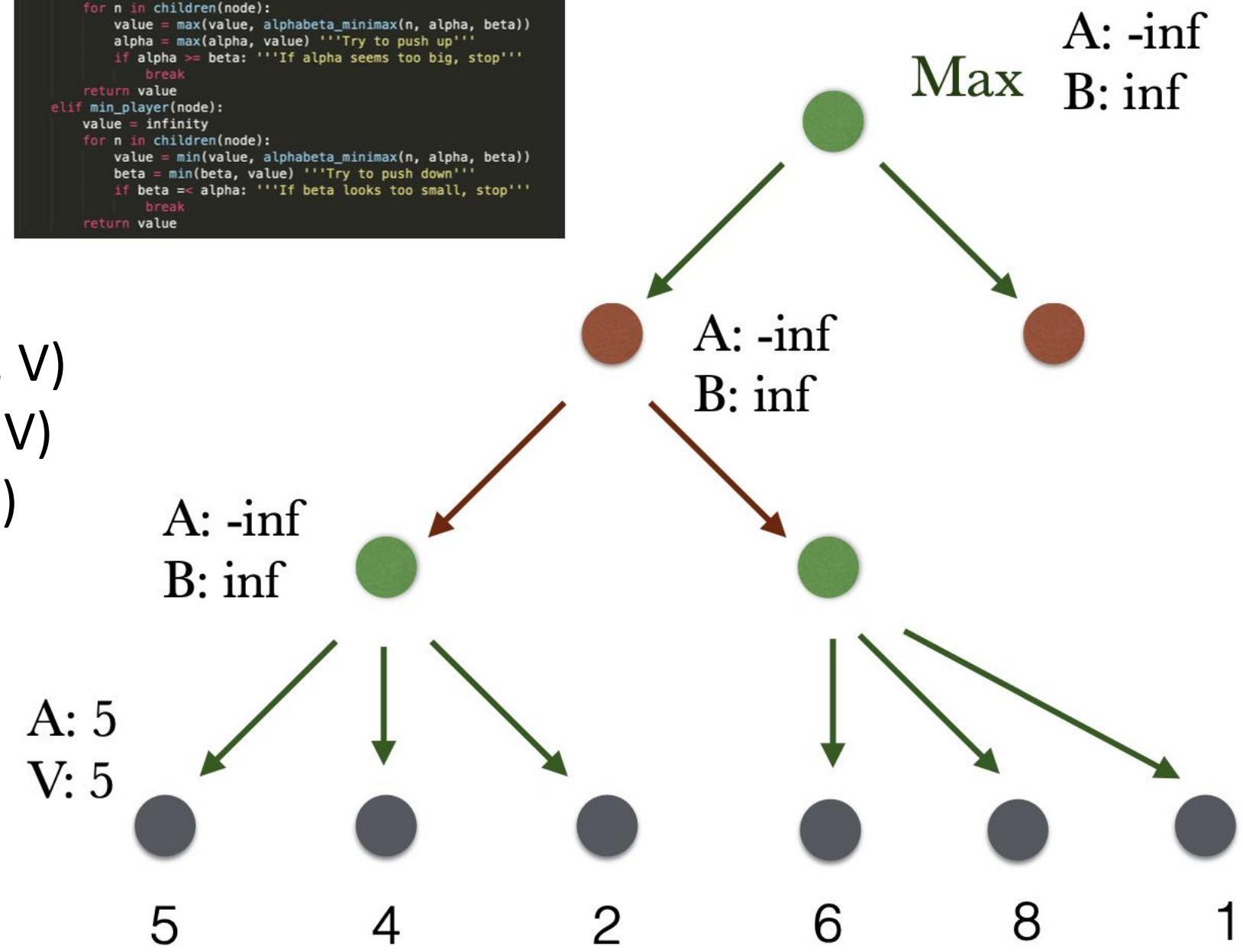
```
alpha = -infinity '''fallback for Max'''
beta = infinity '''fallback for Min'''
node = root
```

```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

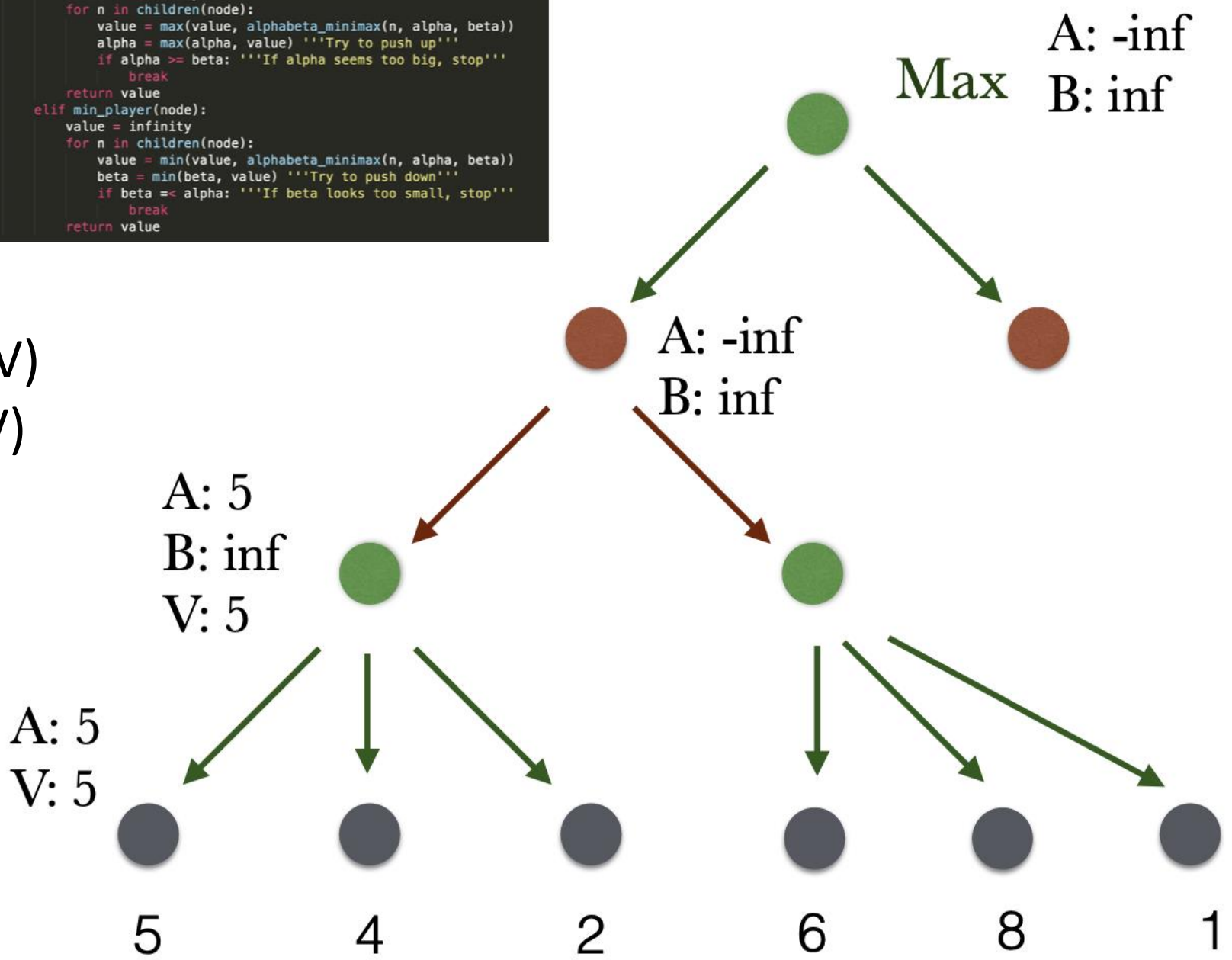


```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

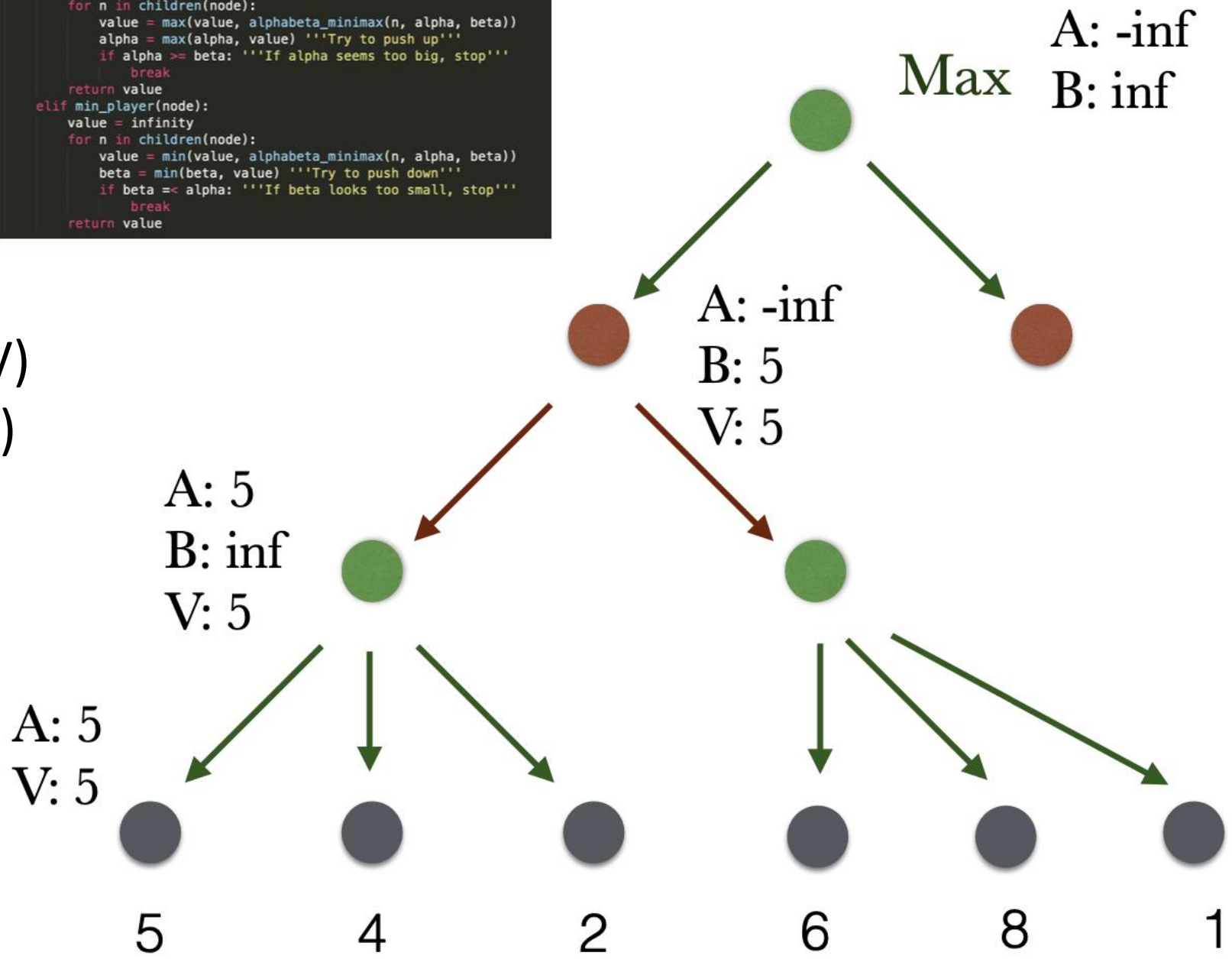


```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

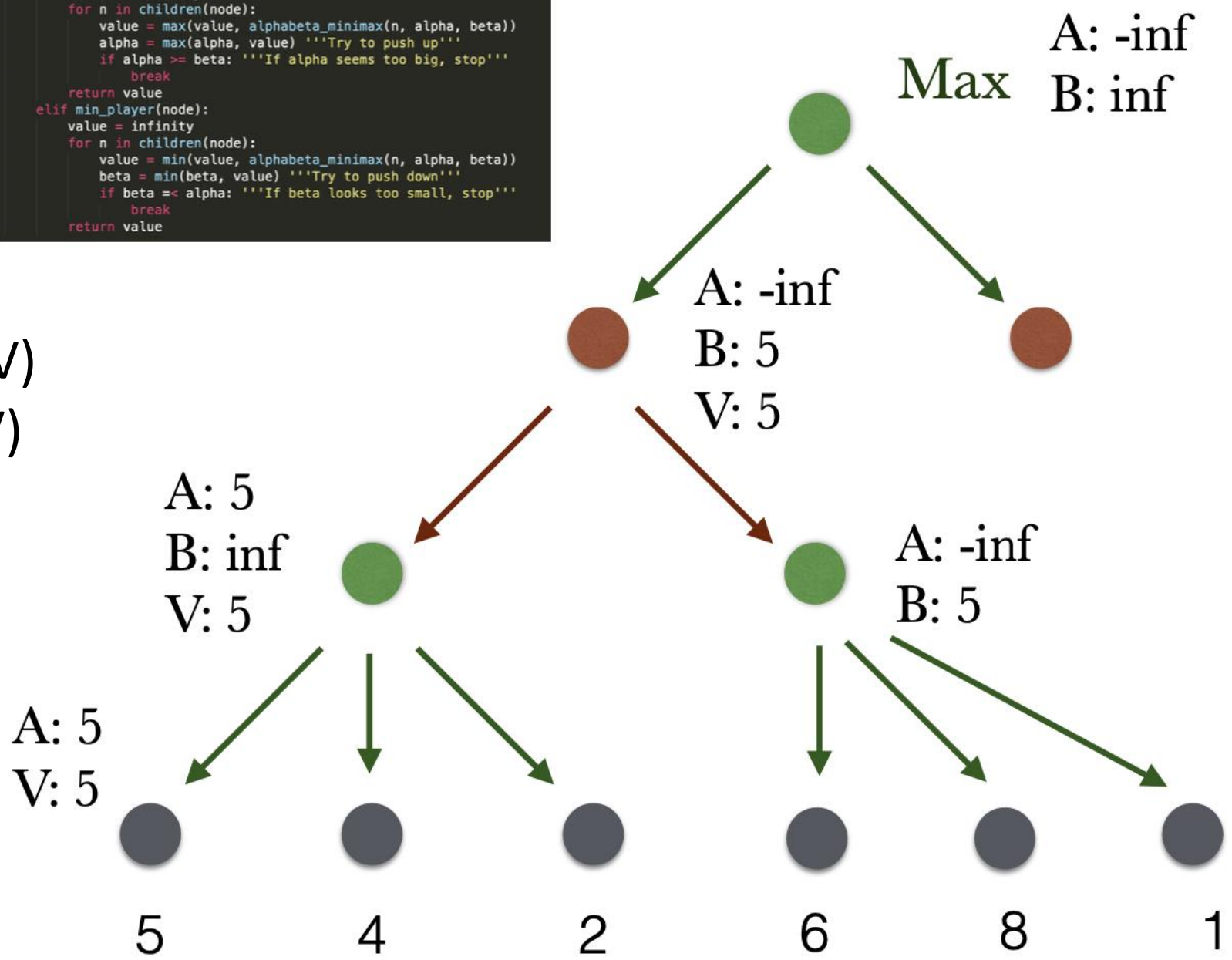


```python
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

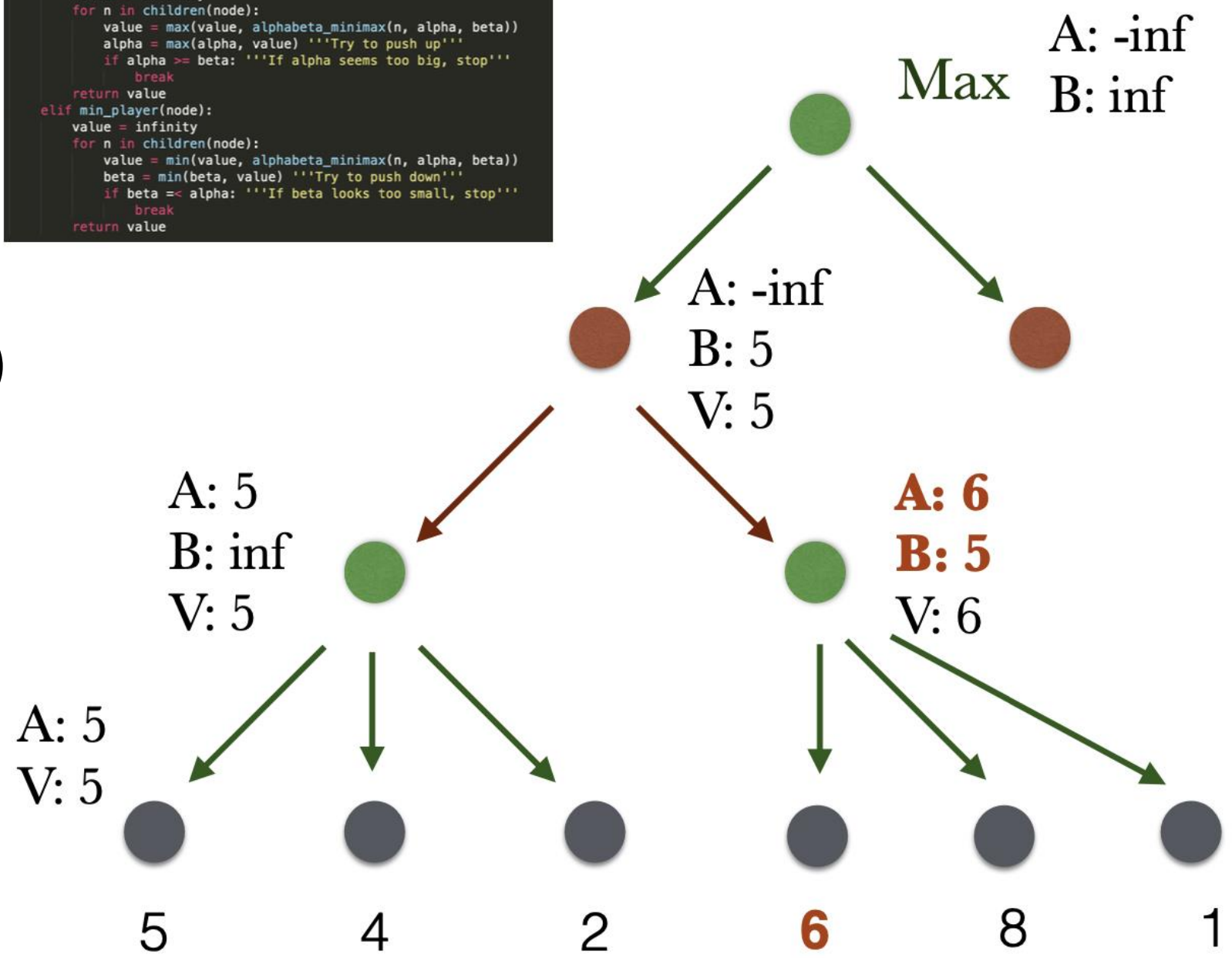


```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

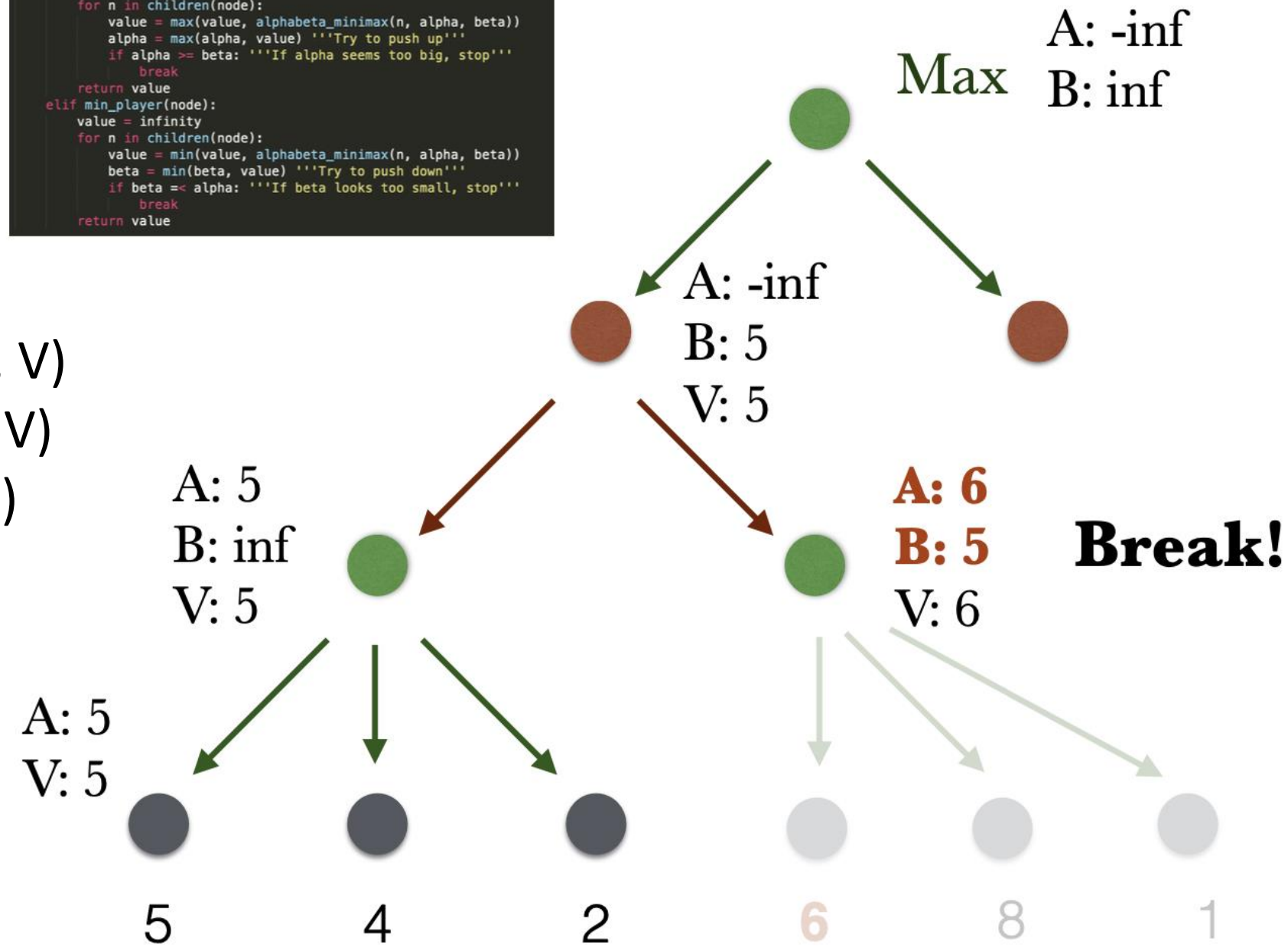


```
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

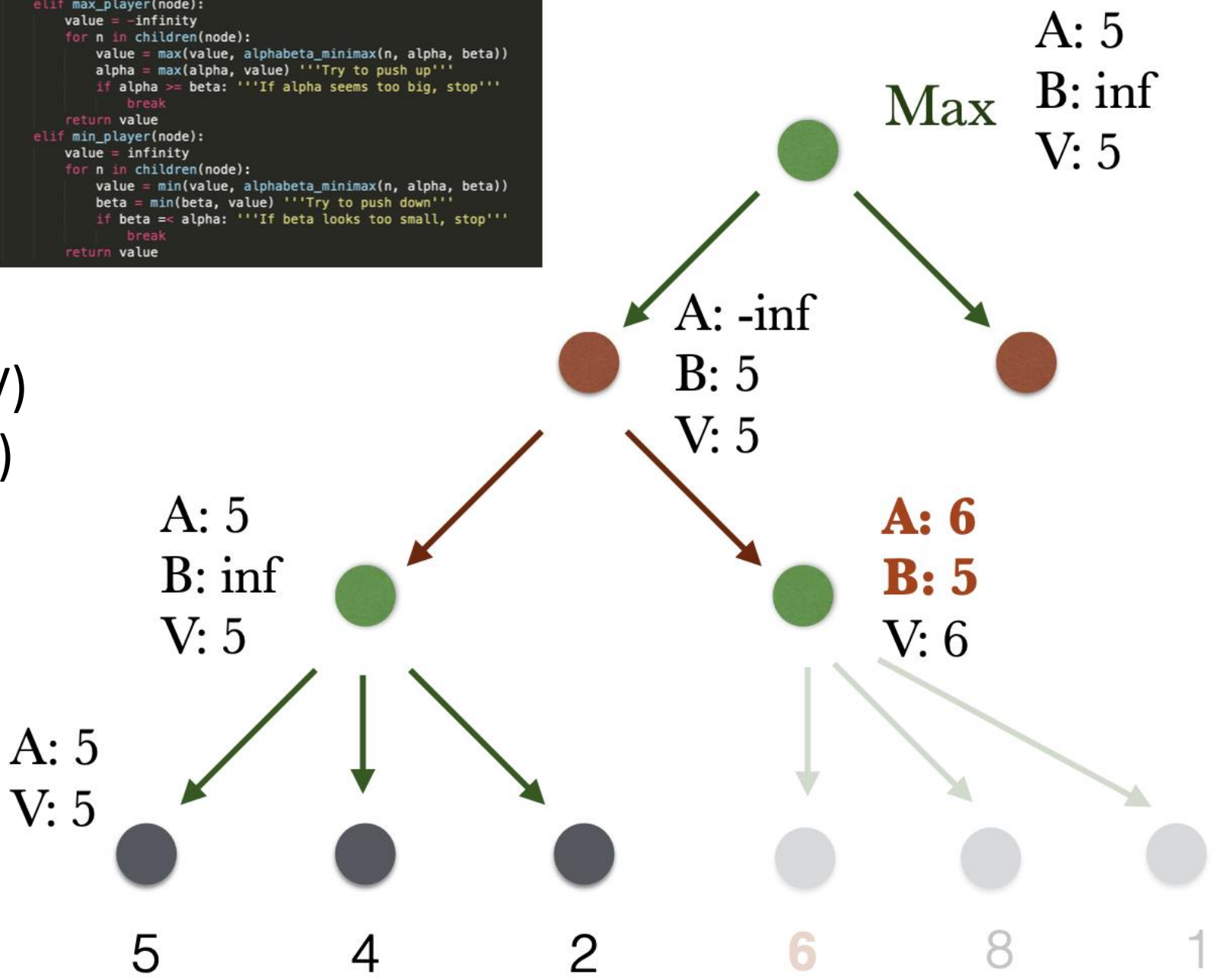


```python
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

B = min(B, V)

V = max(V')

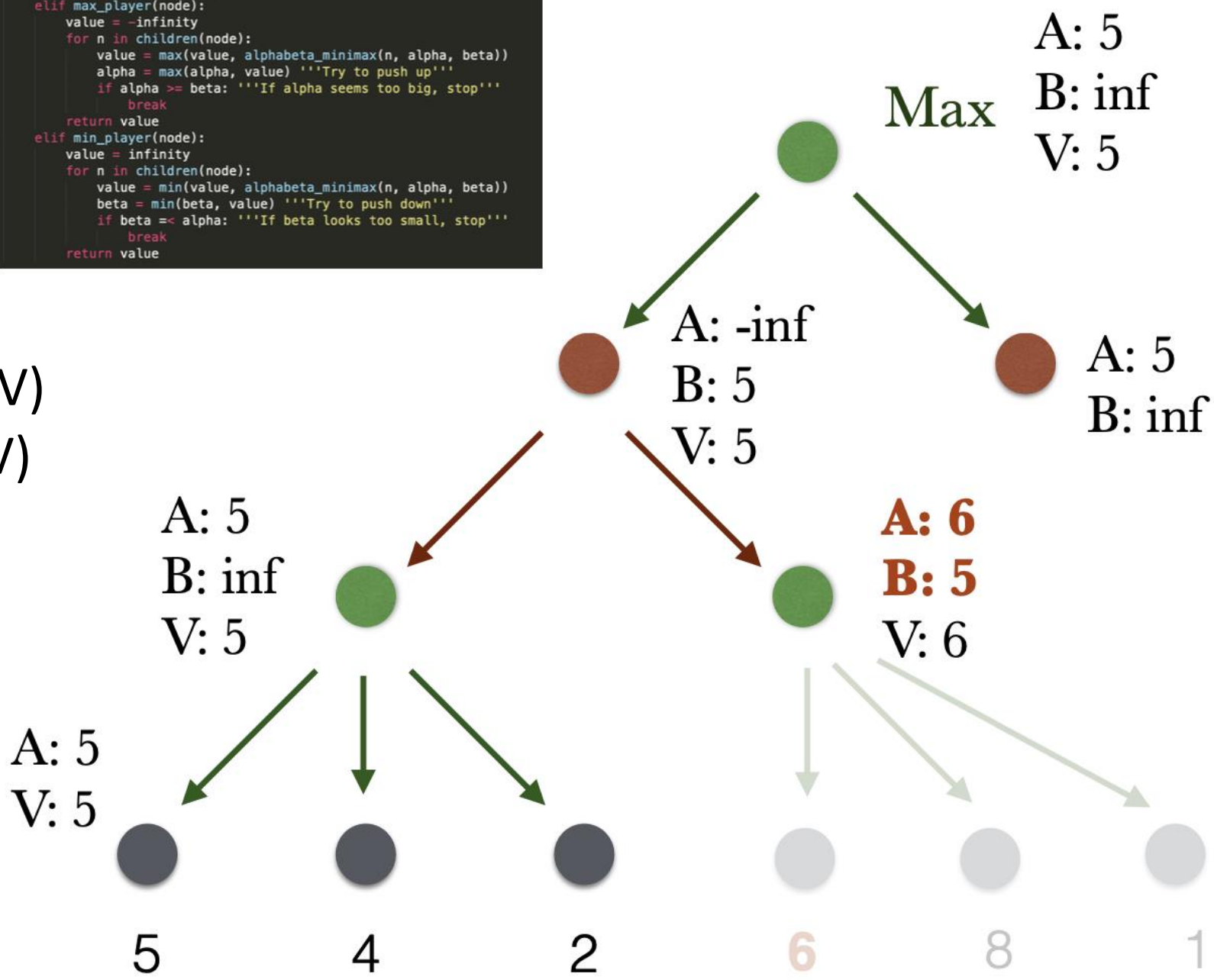


```python
def alphabeta_minimax(node, alpha, beta):
    if terminal(node):
        return payoff(node)
    elif max_player(node):
        value = -infinity
        for n in children(node):
            value = max(value, alphabeta_minimax(n, alpha, beta))
            alpha = max(alpha, value) '''Try to push up'''
            if alpha >= beta: '''If alpha seems too big, stop'''
                break
        return value
    elif min_player(node):
        value = infinity
        for n in children(node):
            value = min(value, alphabeta_minimax(n, alpha, beta))
            beta = min(beta, value) '''Try to push down'''
            if beta =< alpha: '''If beta looks too small, stop'''
                break
        return value
```

A = max(A, V)

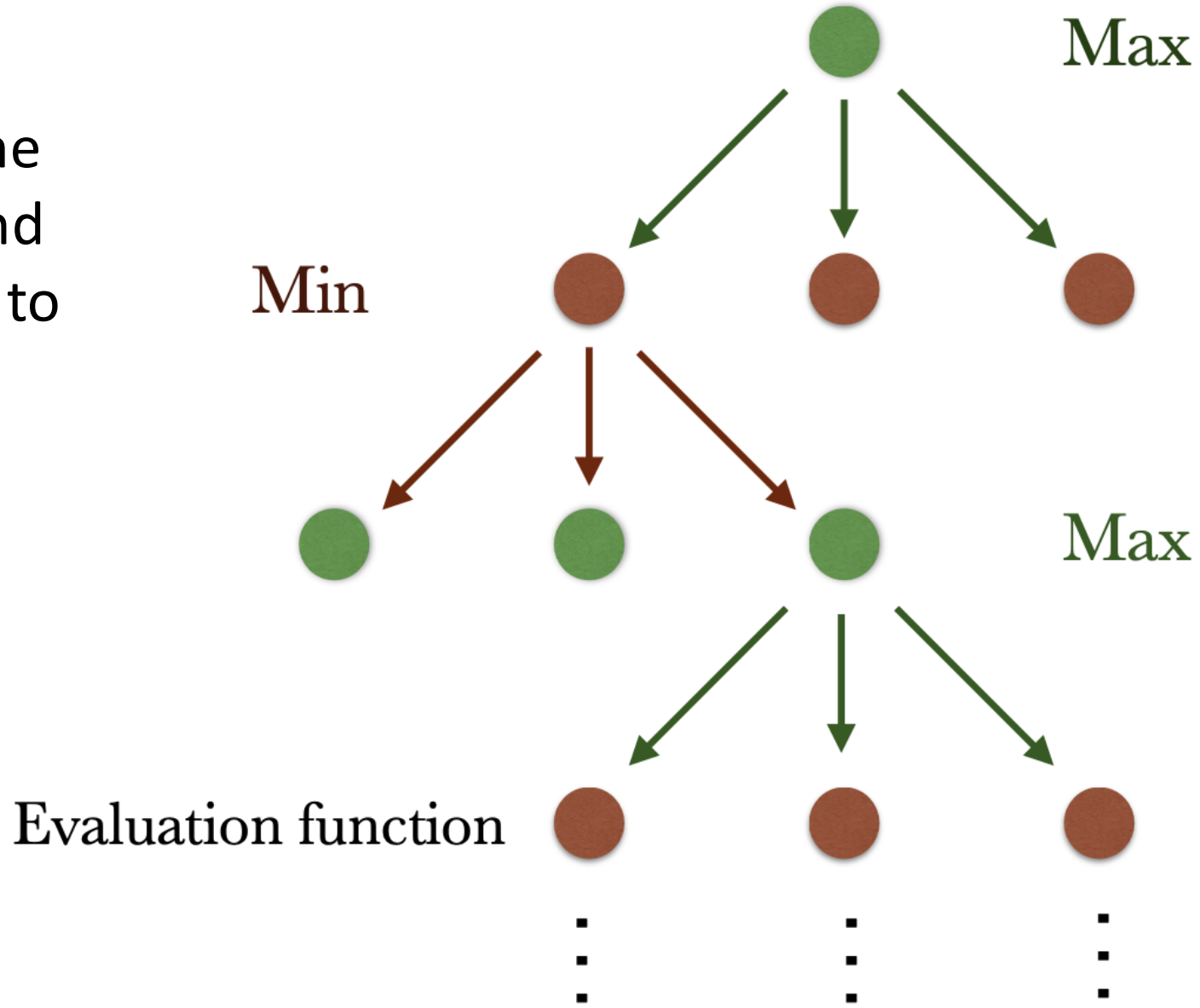
B = min(B, V)

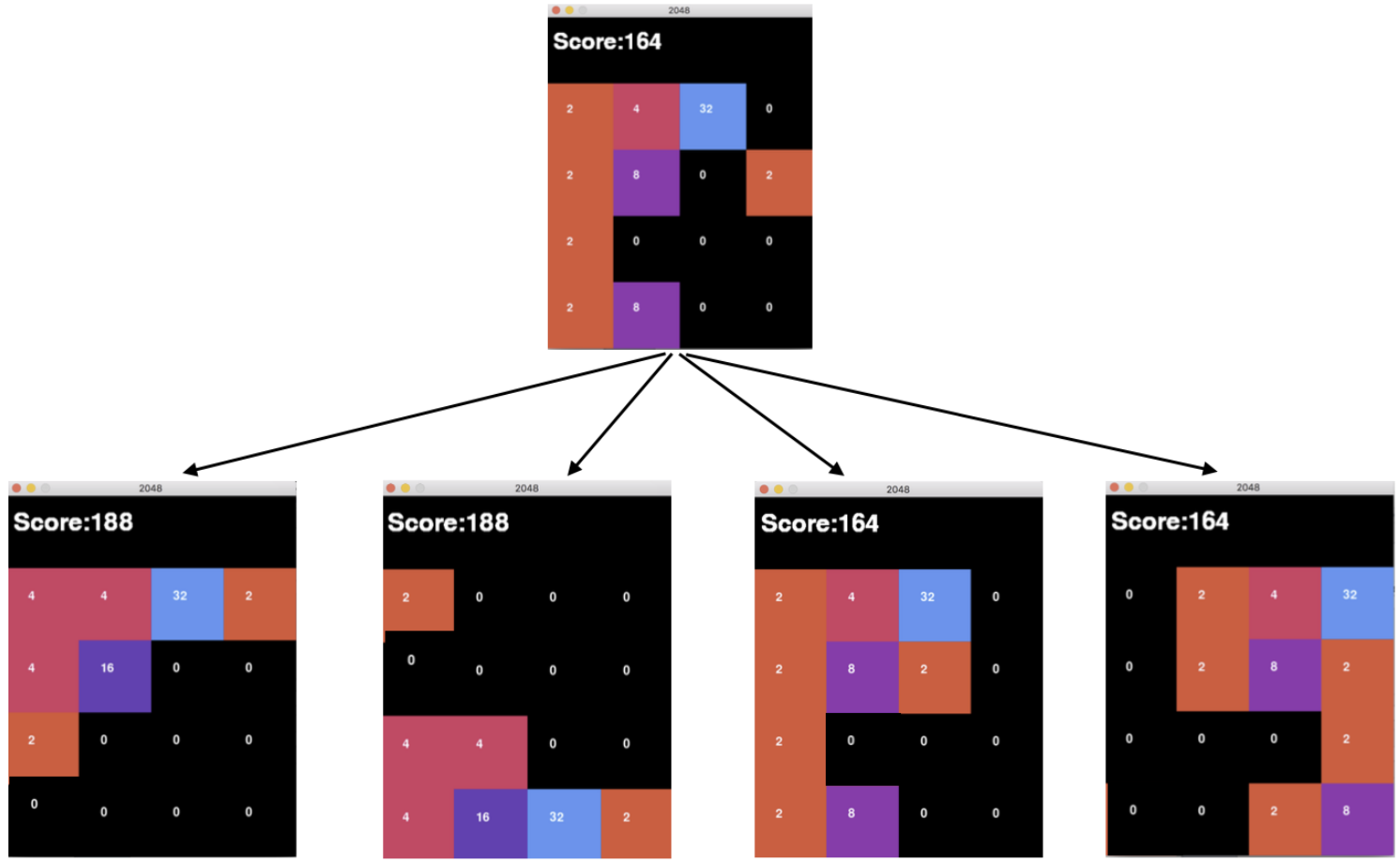
V = max(V')



Vertical Cut-off

Cut-off at some fixed depth and use heuristics to evaluate the intermediate nodes.





Evaluation function: score + more heuristics

Expectimax

- Instead of MIN PLAYER, we have **CHANCE PLAYER**
- At each CHANCE PLAYER's state, the action to take is chosen from the set of all possible actions **with probabilities**

Expectimax

- Instead of MIN PLAYER, we have **CHANCE PLAYER**
- At each CHANCE PLAYER's state, the action to take is chosen from the set of all possible actions **with probabilities**

~~Min Node: $\min(s_1', s_2', \ldots, s_n')$~~ → Chance Node: $\sum_{s'} P(s') \cdot V(s')$ (Expected Value)

PA: 2048

- MAX PLAYER: 4 possible moves → **{Up, Down, Left, Right}**

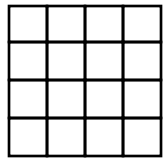
  (Some moves might be invalid → Remove them from the tree)
- CHANCE PLAYER (Agent): Choose the next state **uniformly random** from the empty spots

PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
→ Remove them from the tree)

- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



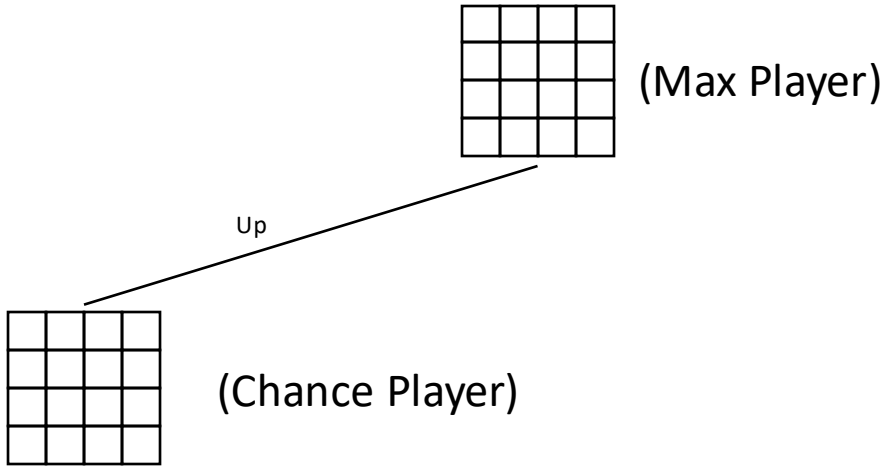
(Max Player)

PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
→ Remove them from the tree)

- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots

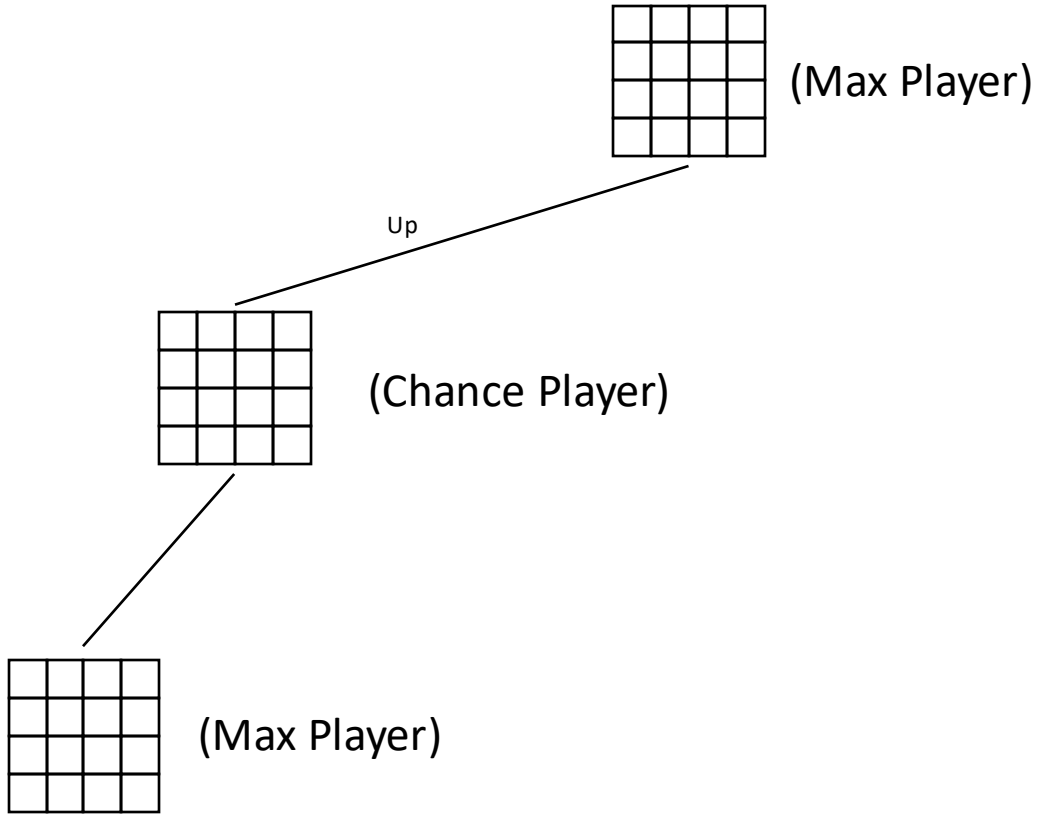


PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
→ Remove them from the tree)

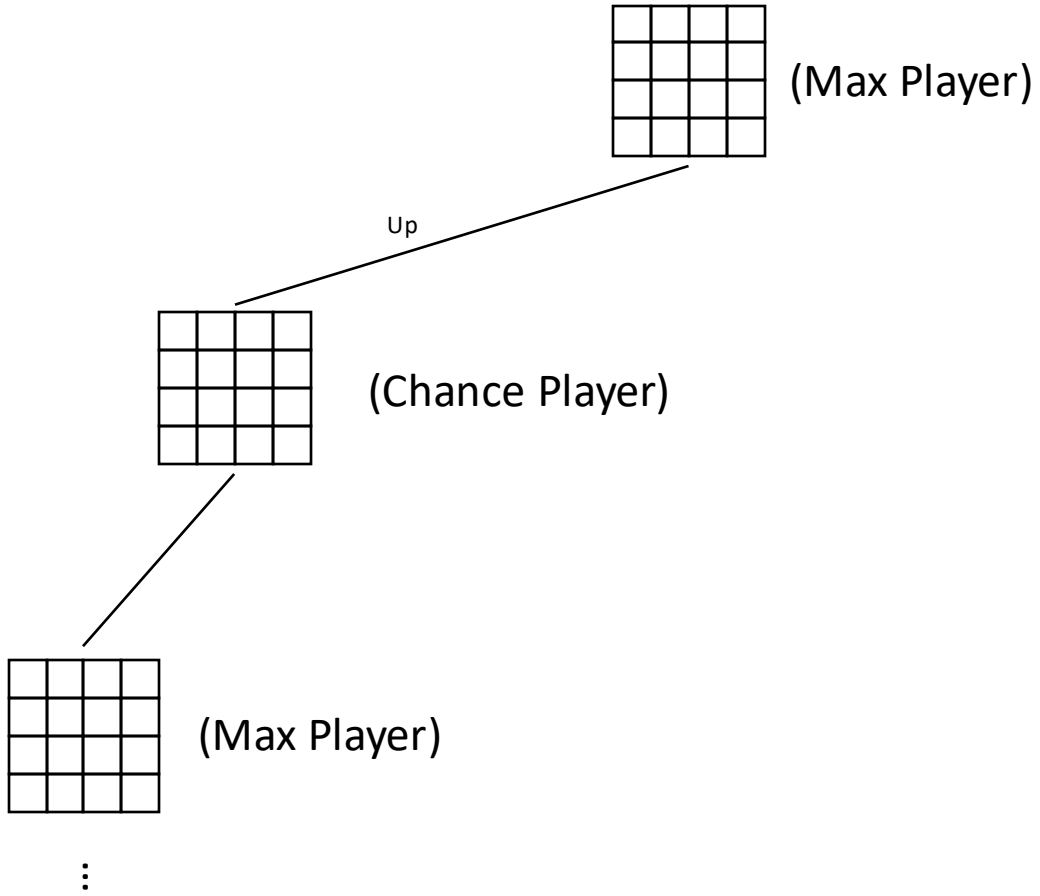
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

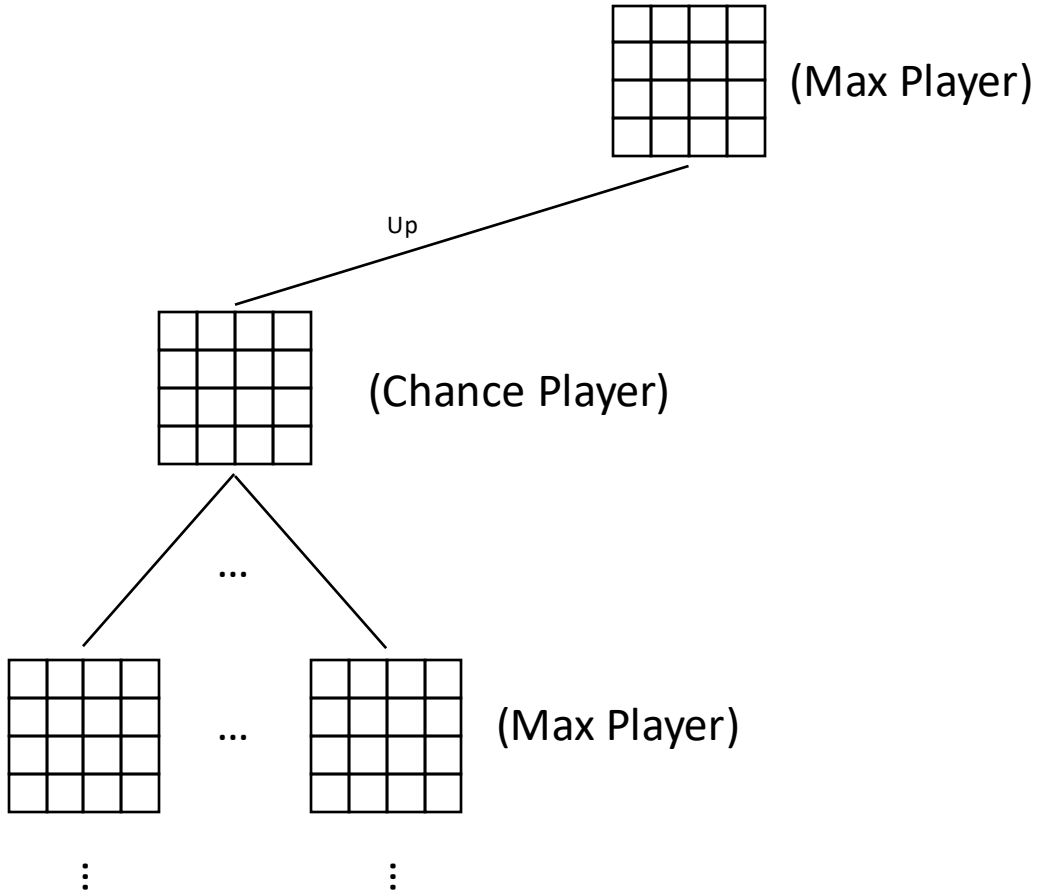
  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots

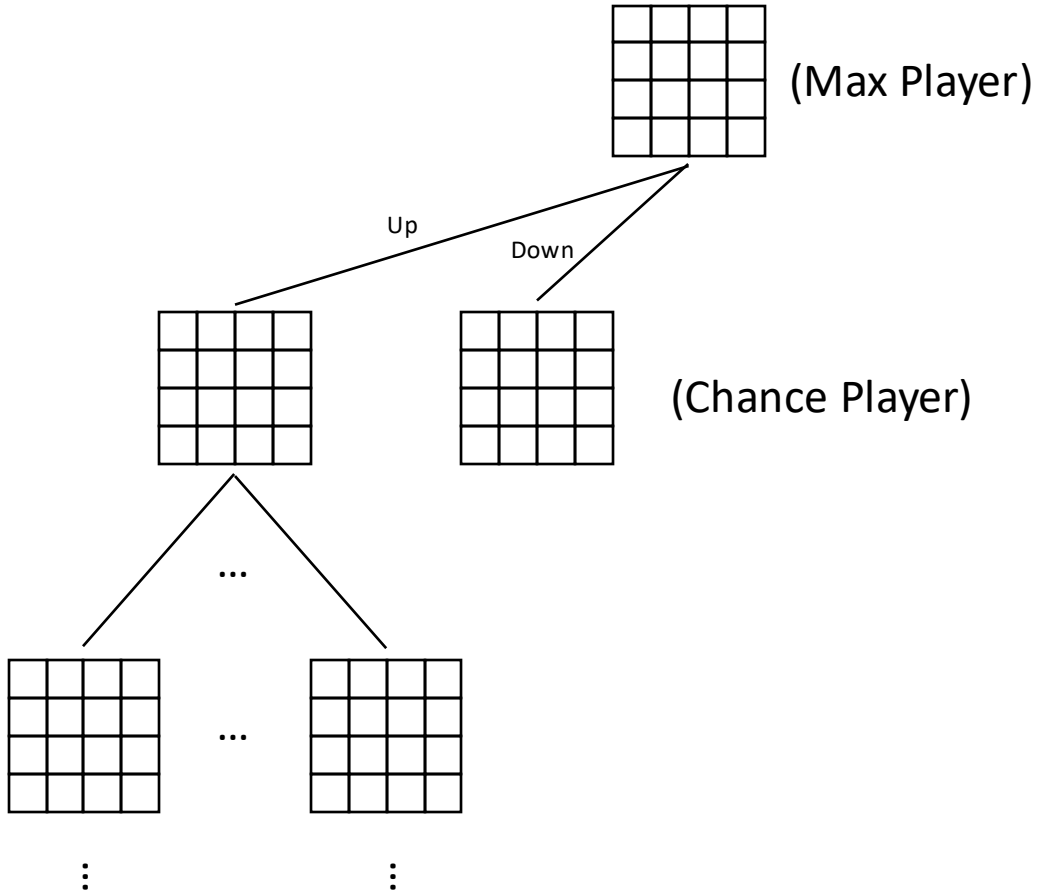


PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
→ Remove them from the tree)

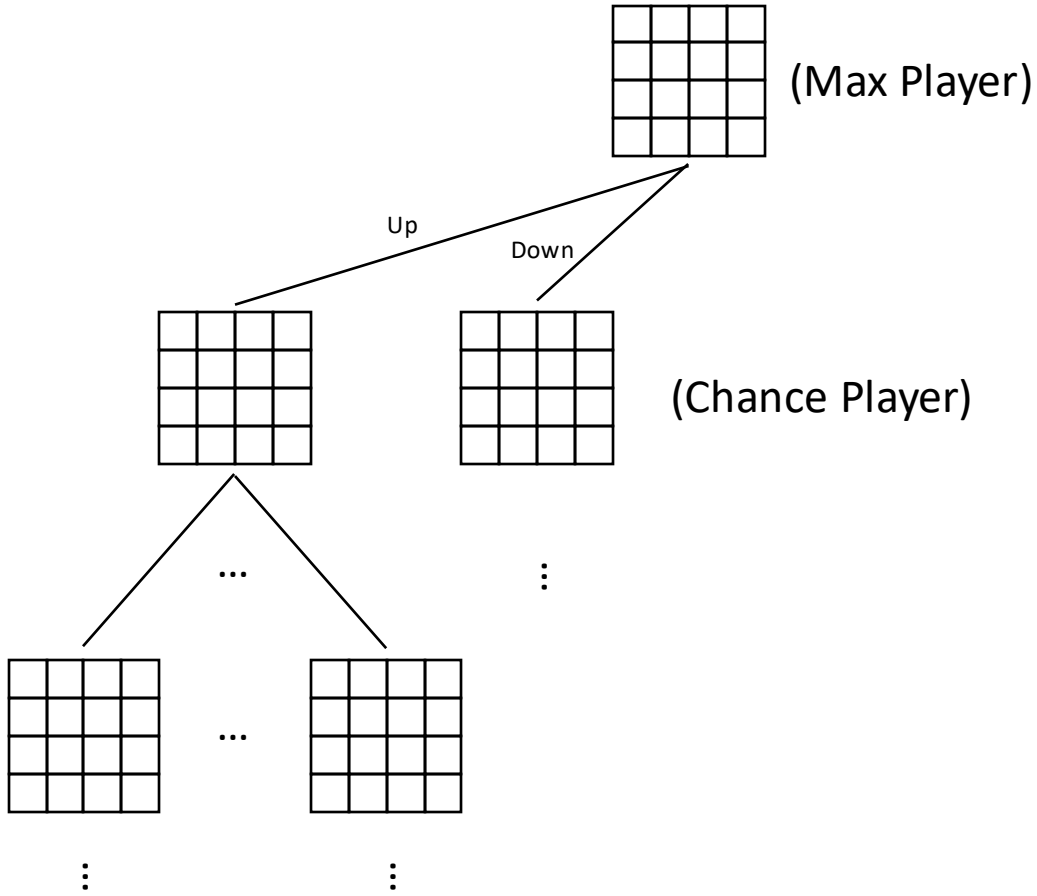
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

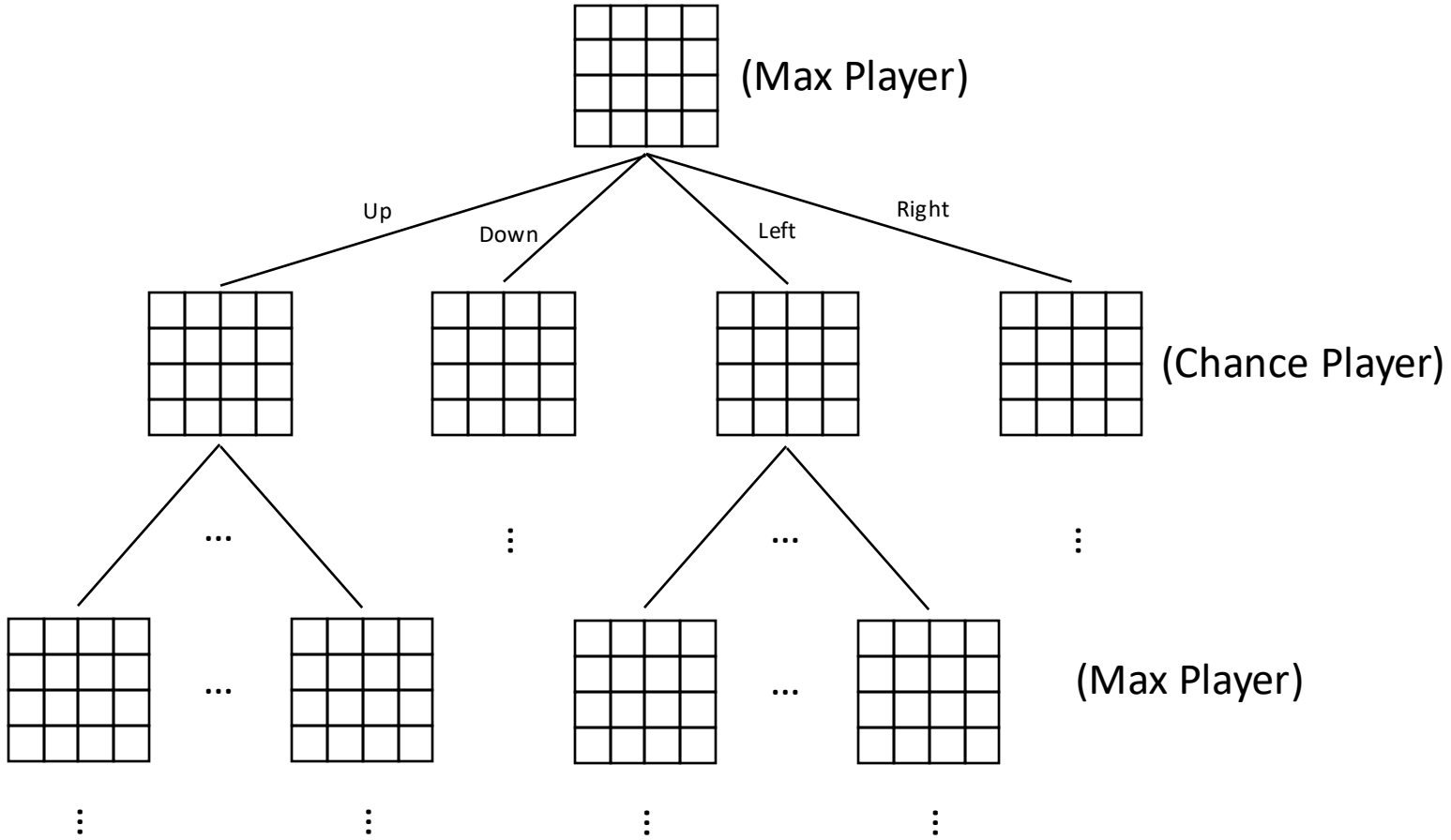
  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`build_tree`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

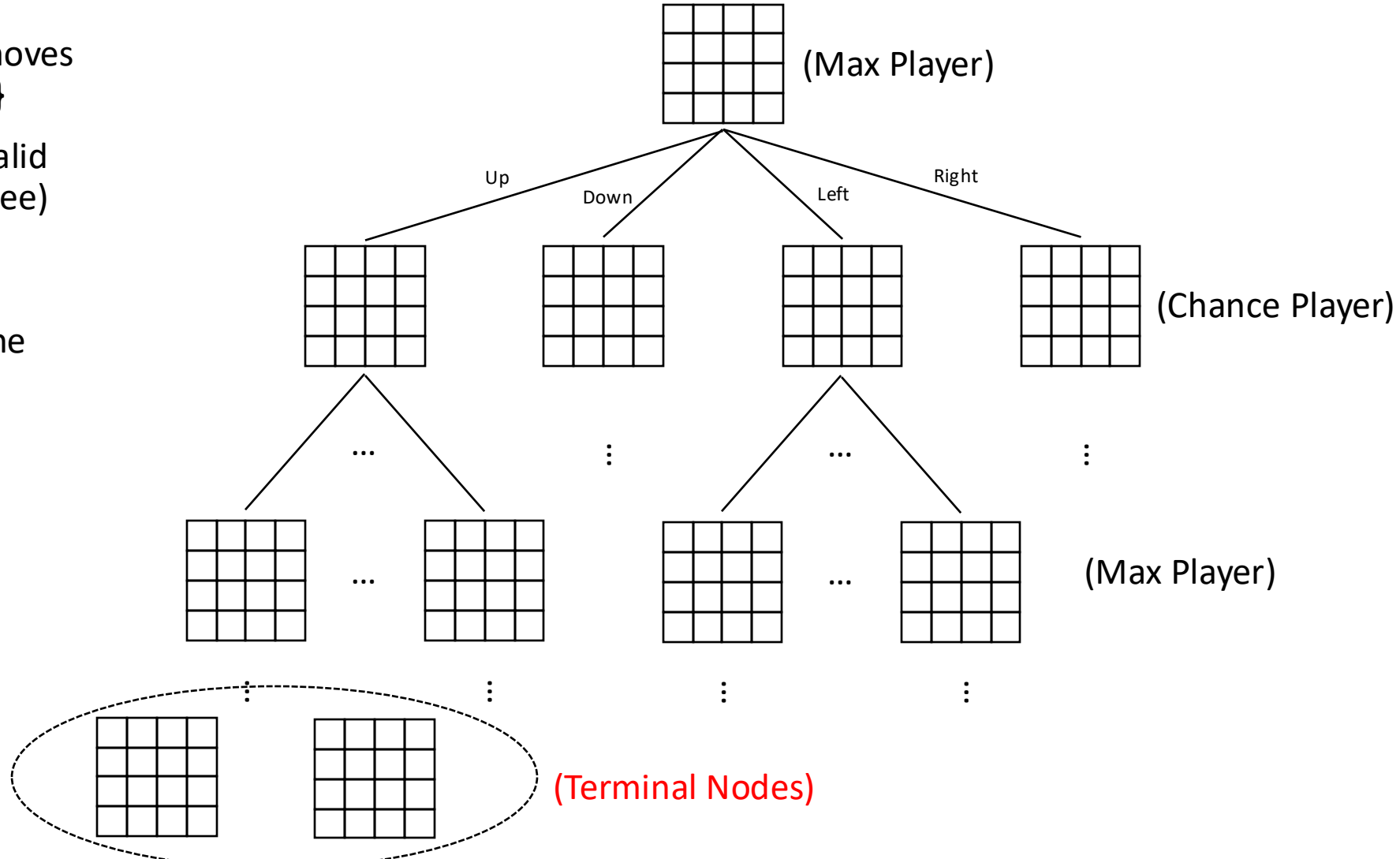
  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`expectimax`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

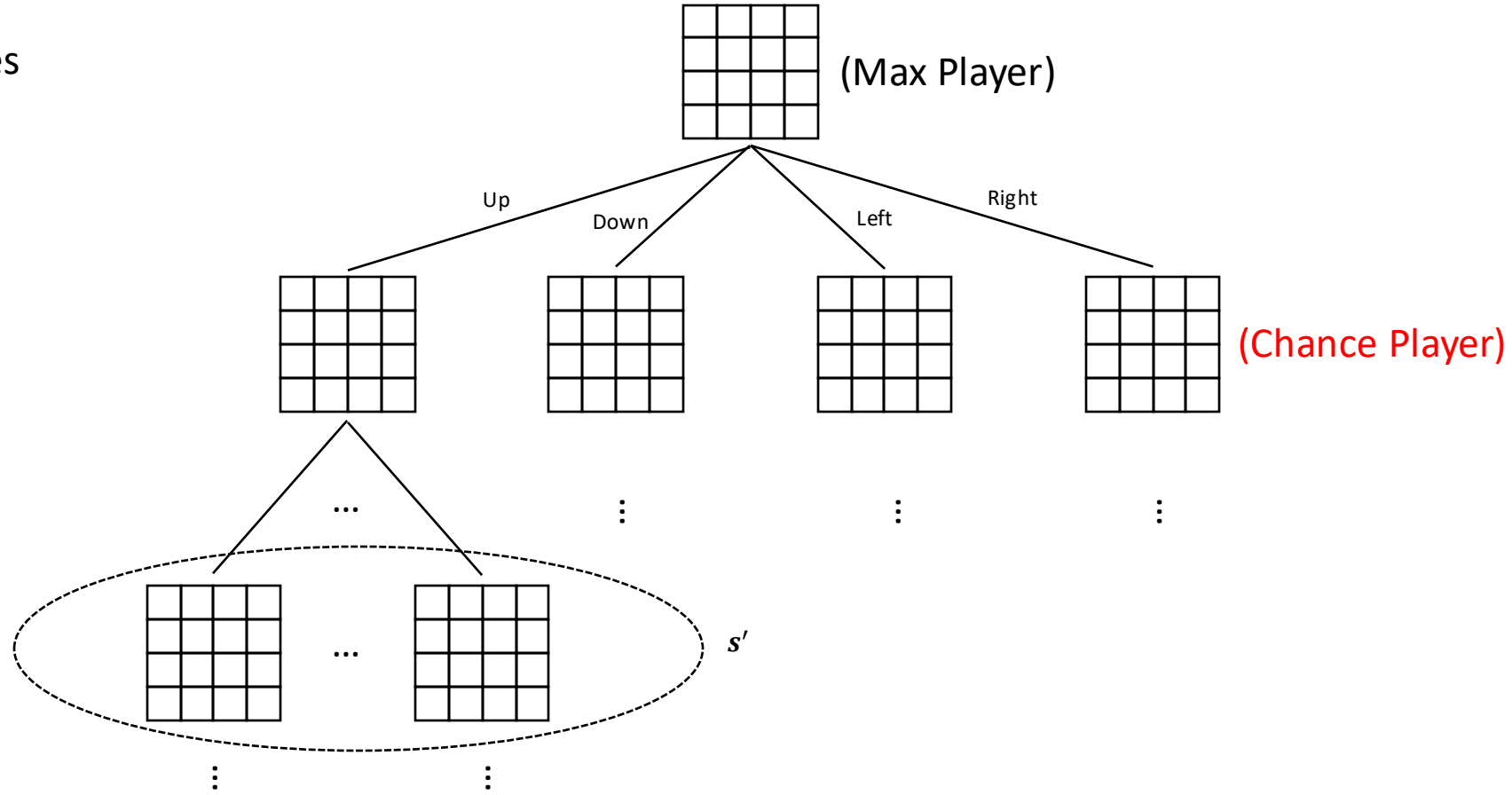
  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`expectimax`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

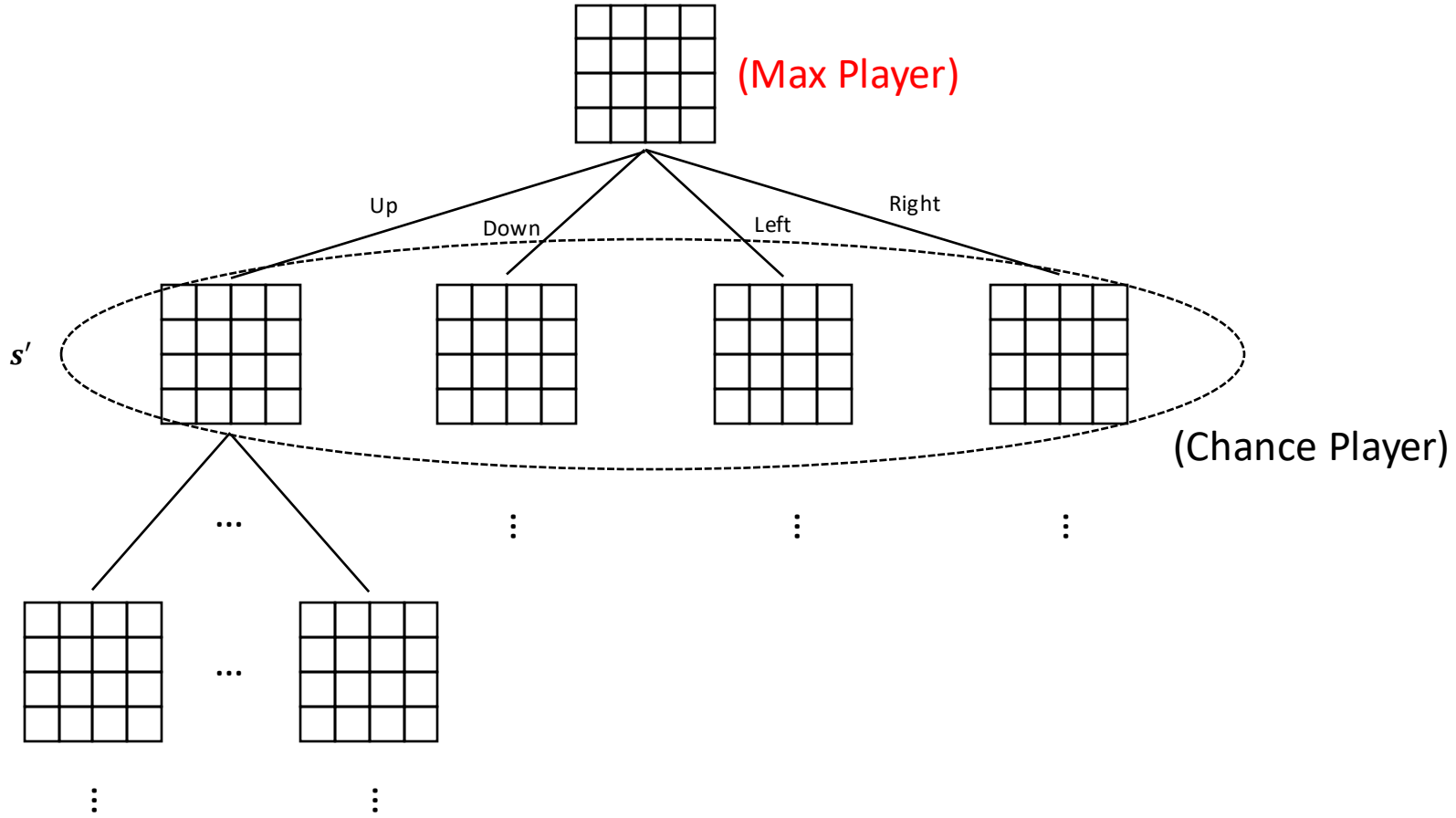
  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



PA: 2048 (`expectimax`)

- MAX PLAYER: 4 possible moves
  → **{Up, Down, Left, Right}**

  (Some moves might be invalid
  → Remove them from the tree)
- CHANCE PLAYER (Agent):
  Choose the next state
  uniformly random from the
  empty spots



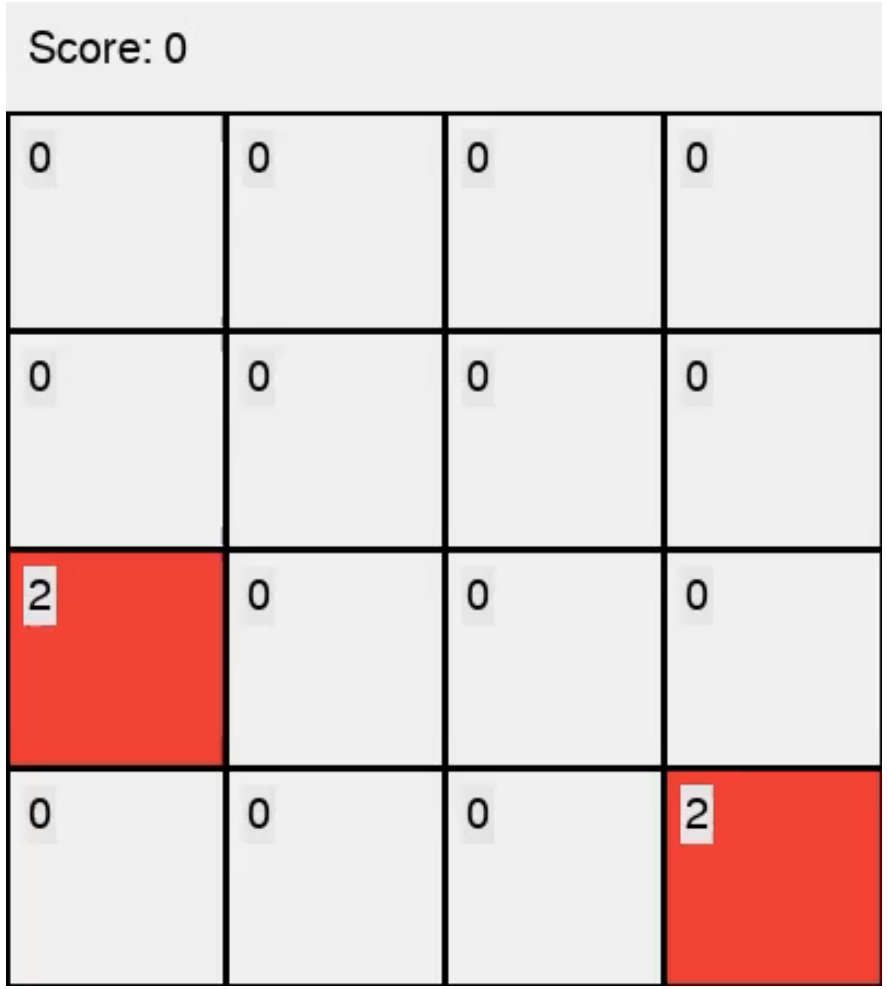
PA: 2048

Helpful Functions in `game.py`:

- `current_state()`: Returns the current state (tile_matrix, score)
- `move(direction)`: Returns `True` if the move was successfully taken
- `set_state(tile_matrix, score)`: Sets the state of the game
- `get_open_tiles()`: Returns a list of empty positions

  `[(__, __), (__, __), …, (__, __)]`
- `Game(tile_matrix, score)`: Creates a new game object (useful for “simulating” moves)

PA: 2048



PA: 2048 Extra Credit

- Good luck ☺
- Teaching staff will not be giving hints ☹