JOHNS HOPKINS
WHITING SCHOOL
of ENGINEERING

Introduction to Adversarial Search and Game Play

Outline

- Agents
- Environment
- Search Strategies
- Search Methods

Solving two-Player Games with Search Strategies

- The goal is to develop methods that can make decisions in game play as humans think

- Problems
  - "Unpredictable" opponent
    - Specifying a move for every possible opponent reply
  - Time limits
    - Unlikely to find a goal on time, must approximate

Slides to be added Summer 2022

MINIMAX-DECISION Algorithm

- Search the tree to the end
- Assign utility values to terminal nodes
- Find the best move for MAX (this is MAX's turn) assuming:
  - MAX will make the move that maximizes utility
  - MIN will make the move that minimizes MAX’s utility

Russell and Norvig, 2020

MINIMAX-DECISION Algorithm Cont.

- MiniMax the heart of almost *every* computer board game

- Idea:
  - Choose the move/position with highest minimax value. This achieves the best payoff against the best play

- Applies to games where:
  - Perfect play for deterministic games
  - Players take turns, e.g., 2-player game
  - Have perfect information
    - Chess, Checkers, Tic-Tac-Toe

- But can work for games without perfect information or chance
  - Poker, Monopoly, Dice

- Can work in real-time (i.e., not turn based) with timer (*iterative deepening*)

Russell and Norvig, 2020

Properties of MINIMAX-DECISION

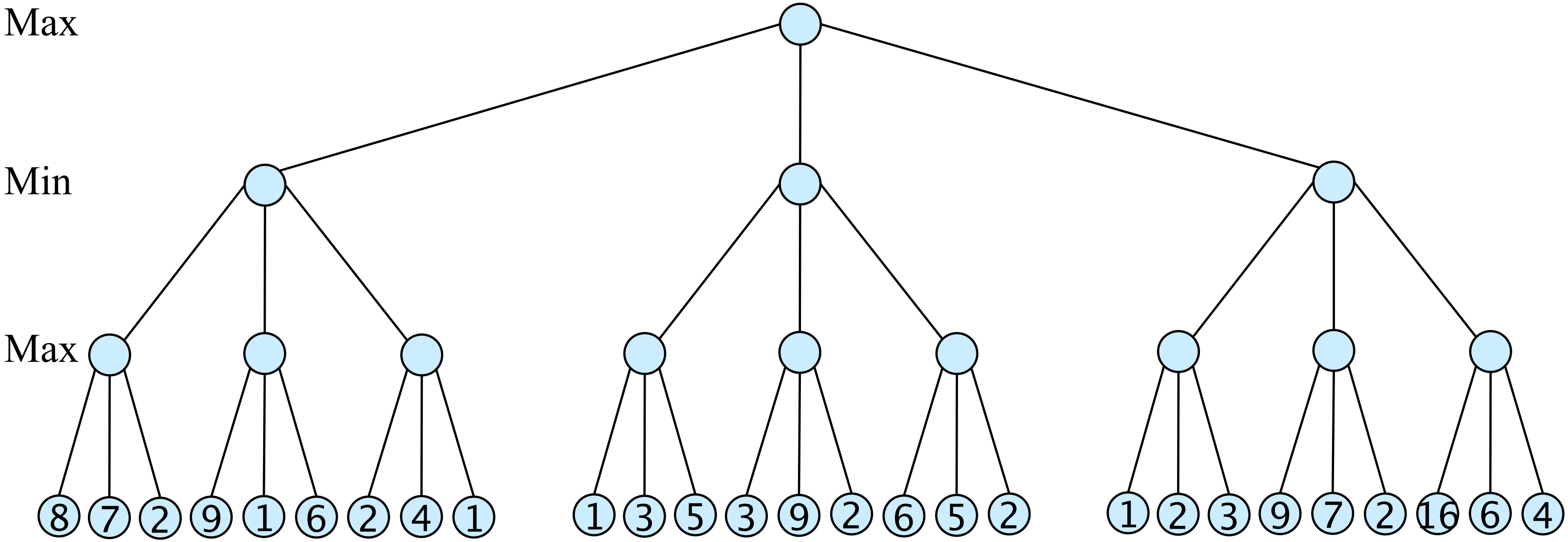
- Complete? Yes (if tree is finite)
- Optimal? Yes (against an optimal opponent)
- Time complexity? $O(b^m)$
- Space complexity? $O(bm)$ (depth-first exploration)
- Standard approach is
  - Apply a cutoff test (depth limit, quiescence)
  - Evaluate nodes at cutoff (evaluation function estimates desirability of position)

MINIMAX-SEARCH Algorithm

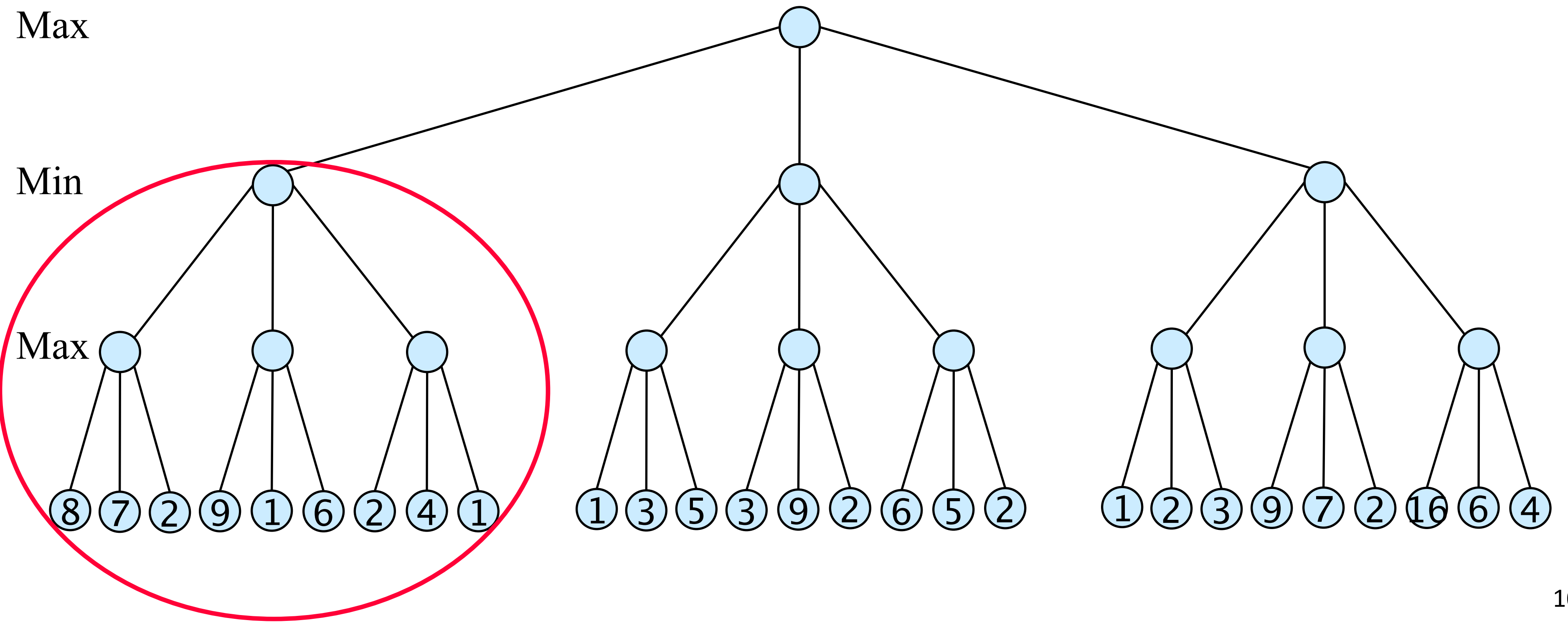
```
function MINIMAX-SEARCH(game, state) returns an action
    player ← game.TO-MOVE(state)
    value, move ← MAX-VALUE(game, state)
    return move
function MAX-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← −∞
    for each a in game.ACTIONS(state) do
        v2 , a2 ← MIN-VALUE(game, game.RESULT(state, a))
        if v2 > v then
            v, move ← v2, a
    return v, move
function MIN-VALUE(game, state) returns a (utility, move) pair
    if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
    v ← +∞
    for each a in game.ACTIONS(state) do
        v2 , a2 ← MAX-VALUE(game, game.RESULT(state, a))
        if v2 < v then
            v, move ← v2, a
    return v, move
```

Russell and Norvig, 2020

MINIMAX-DECISION Example (Winston, 1992)



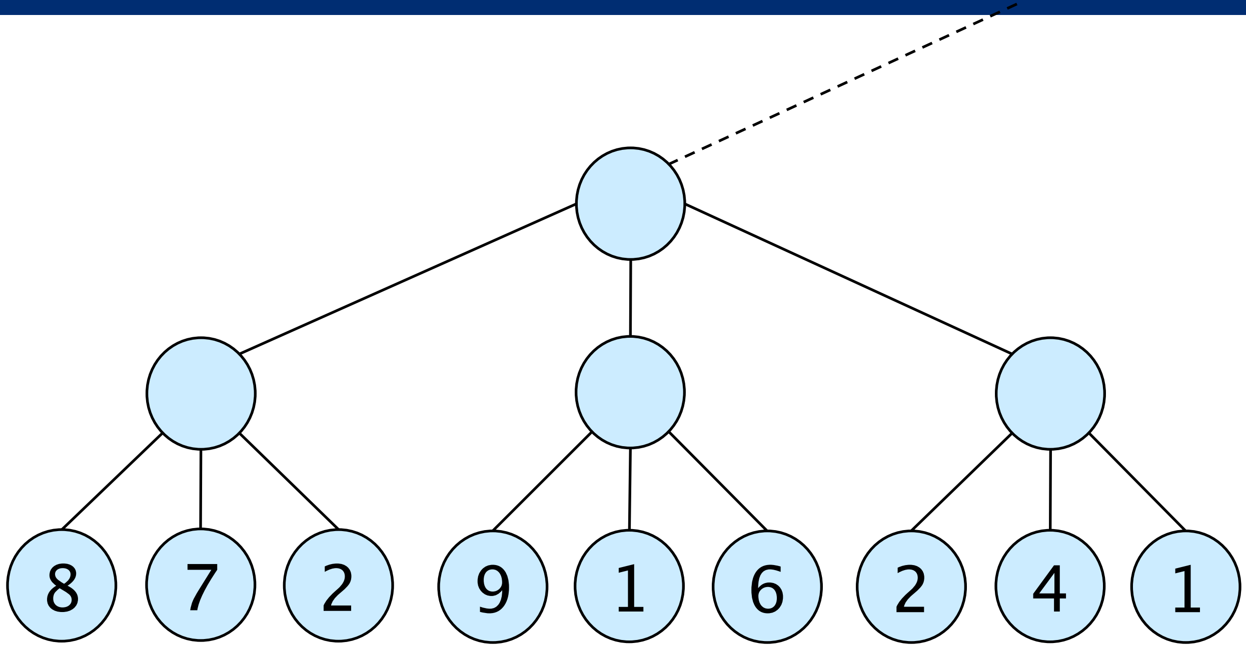
MINIMAX-DECISION Example Cont.



MINIMAX-DECISION Example Cont.

Minimizing Level

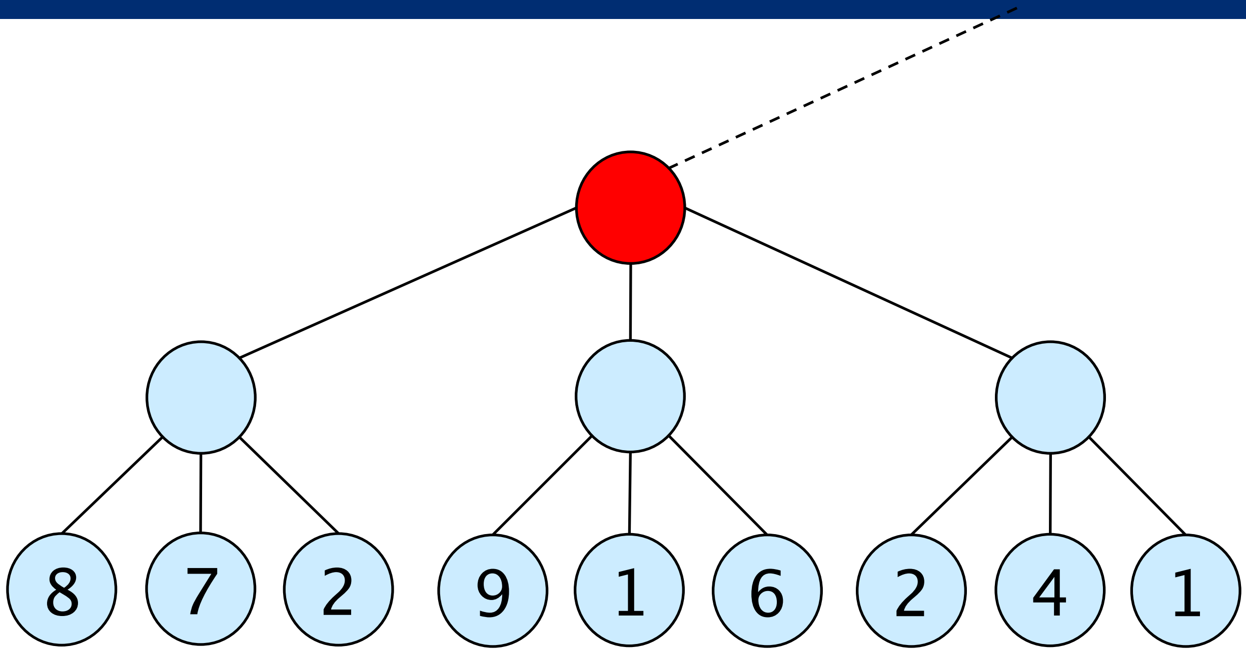
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

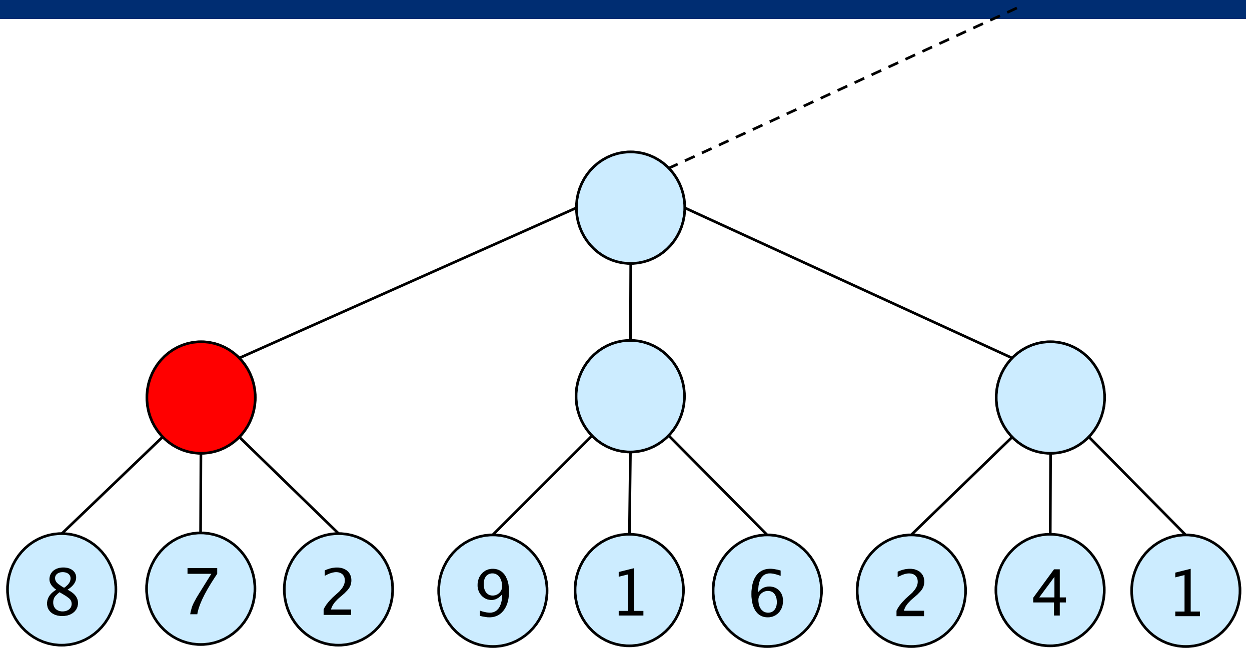
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

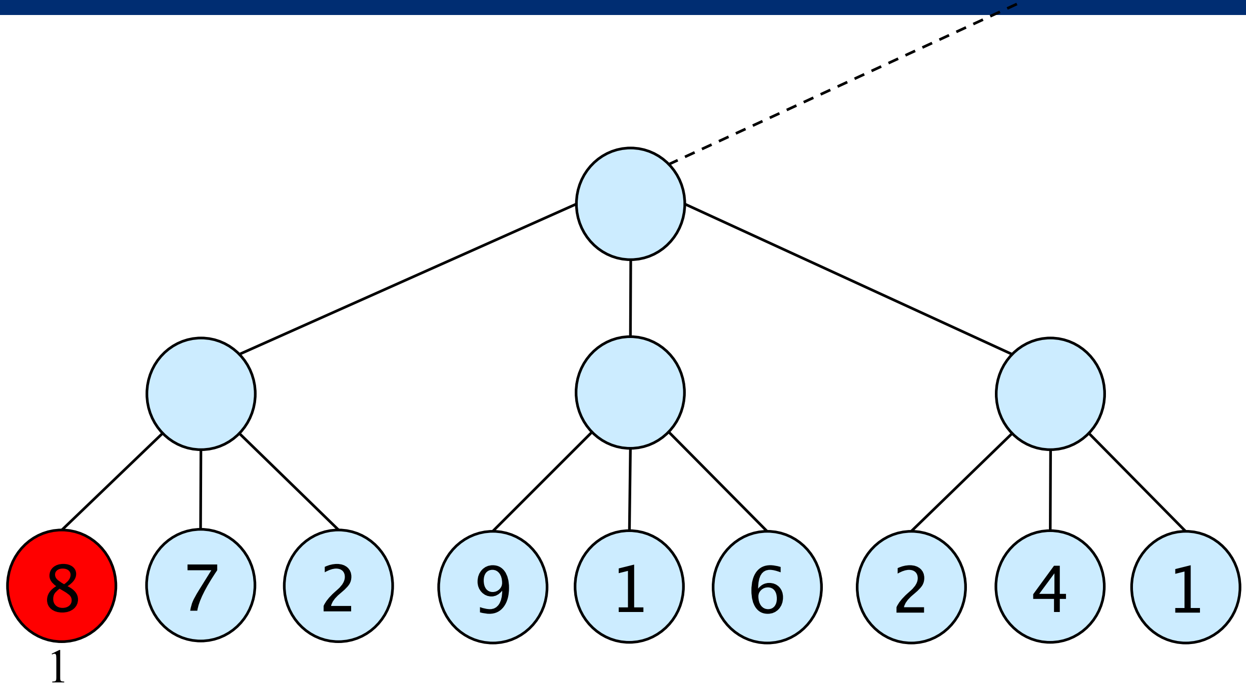
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

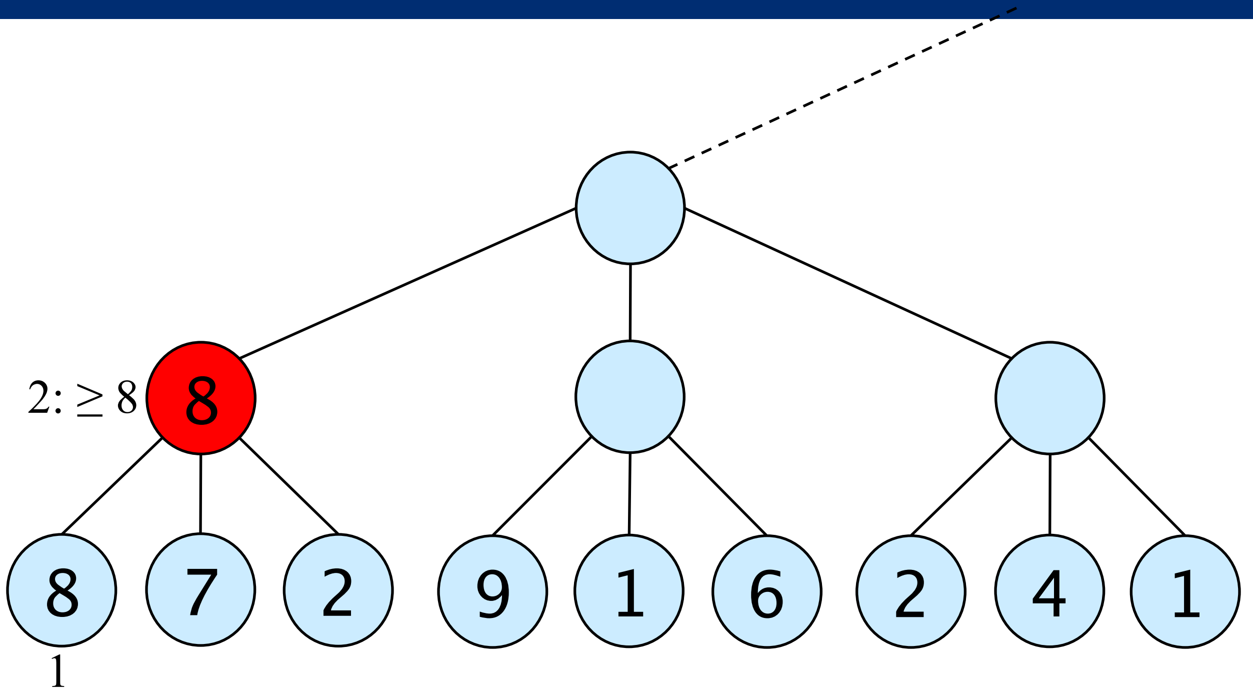
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

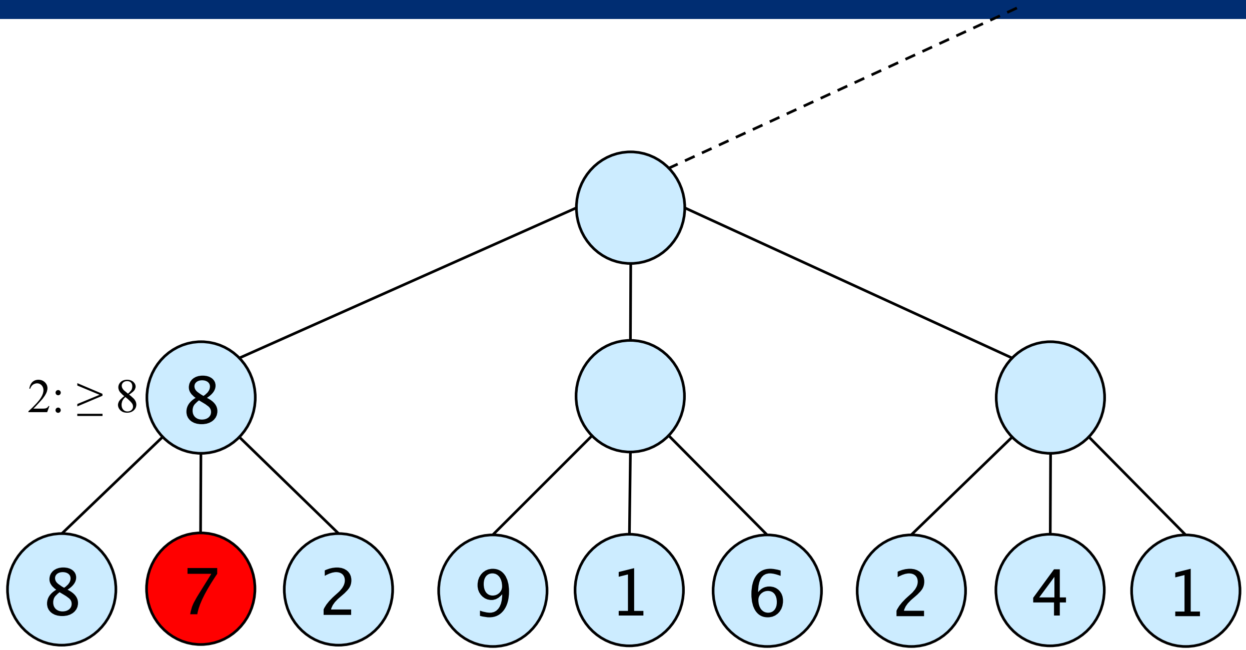
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

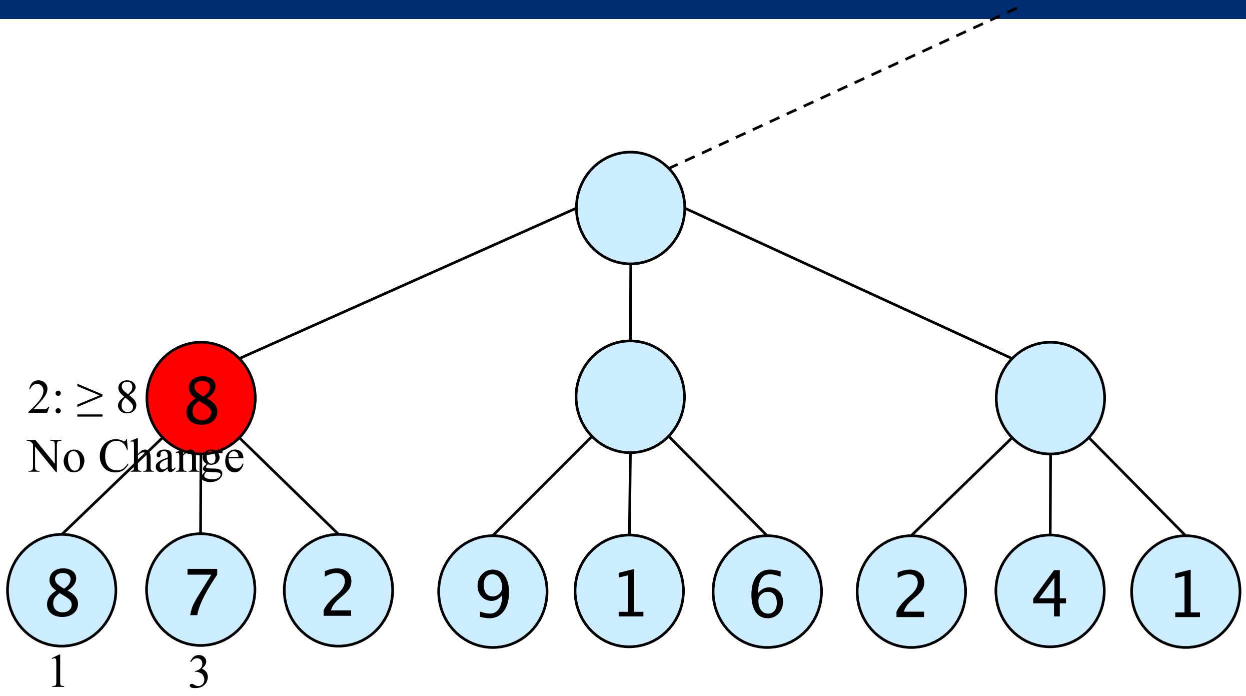
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

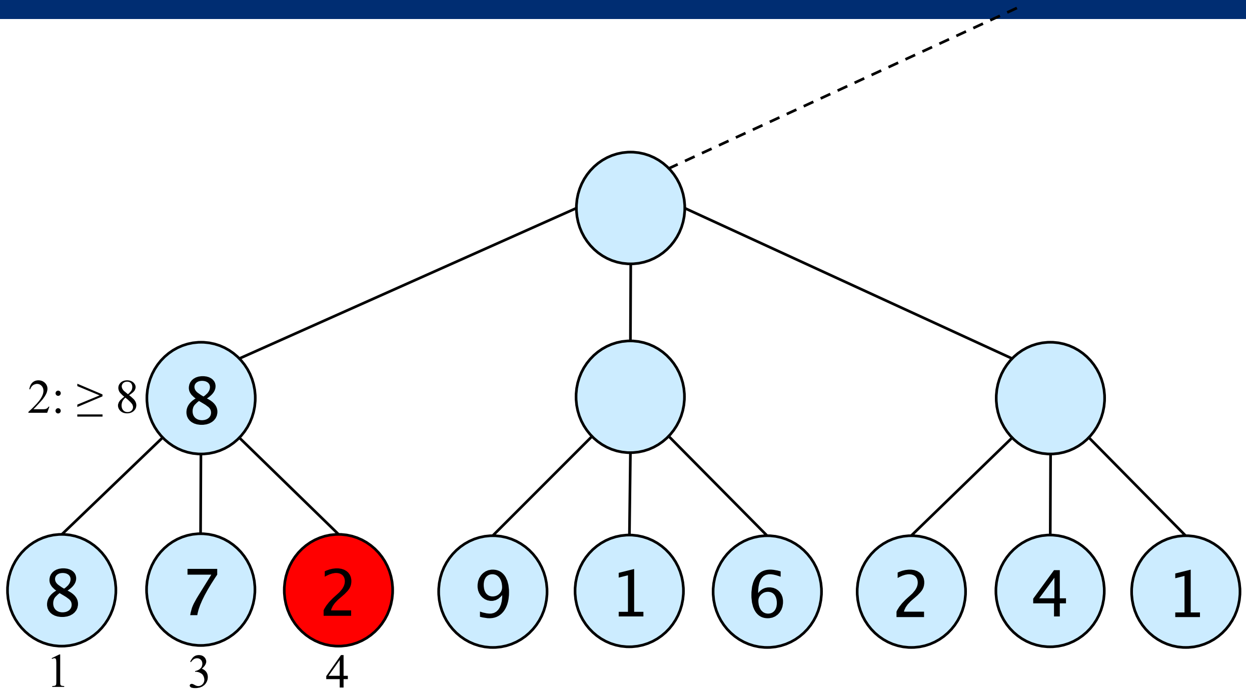
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

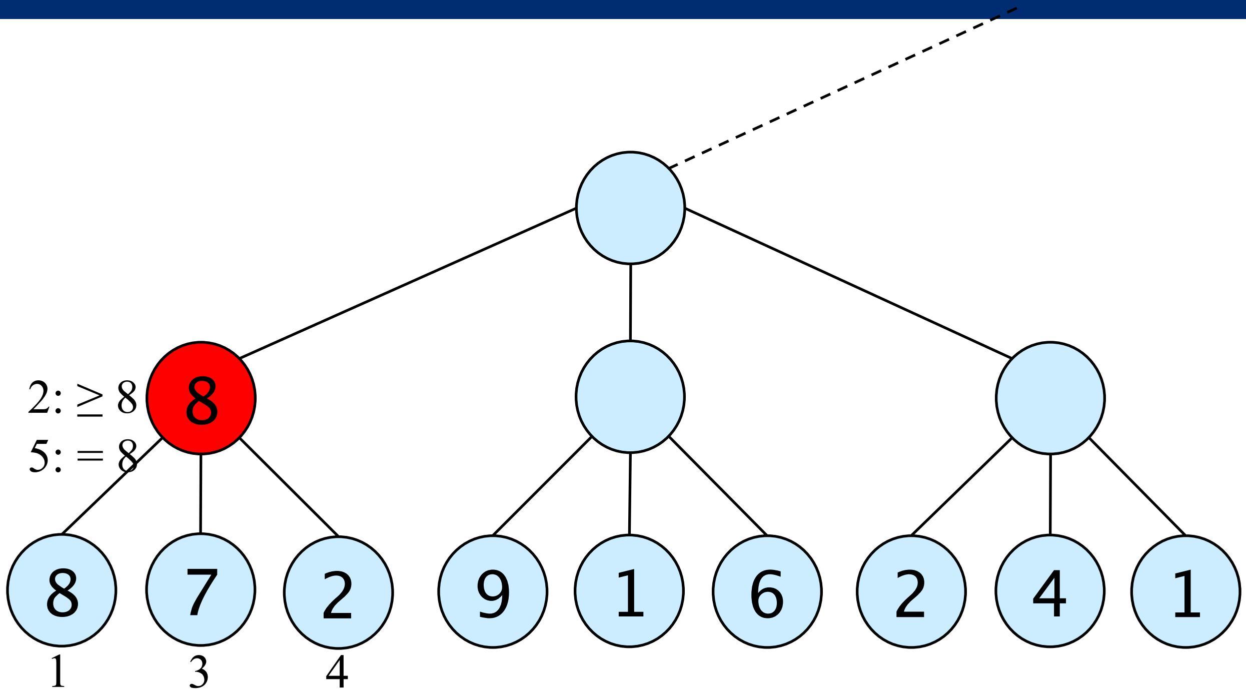
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

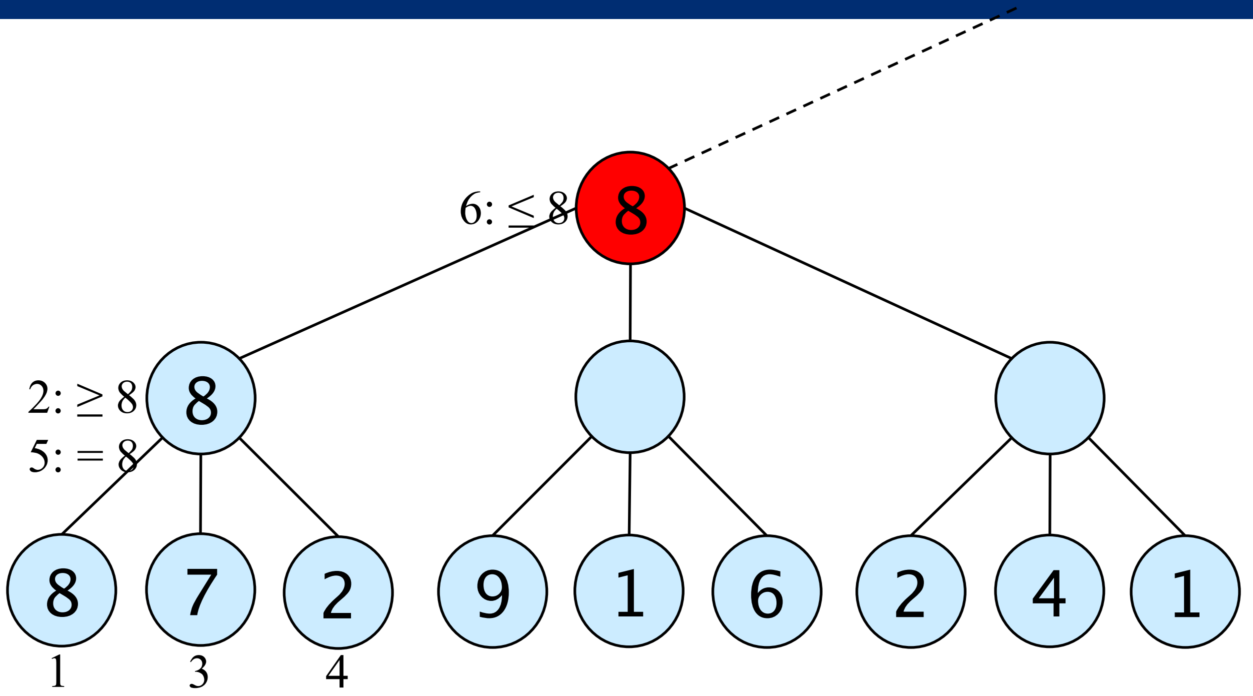
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

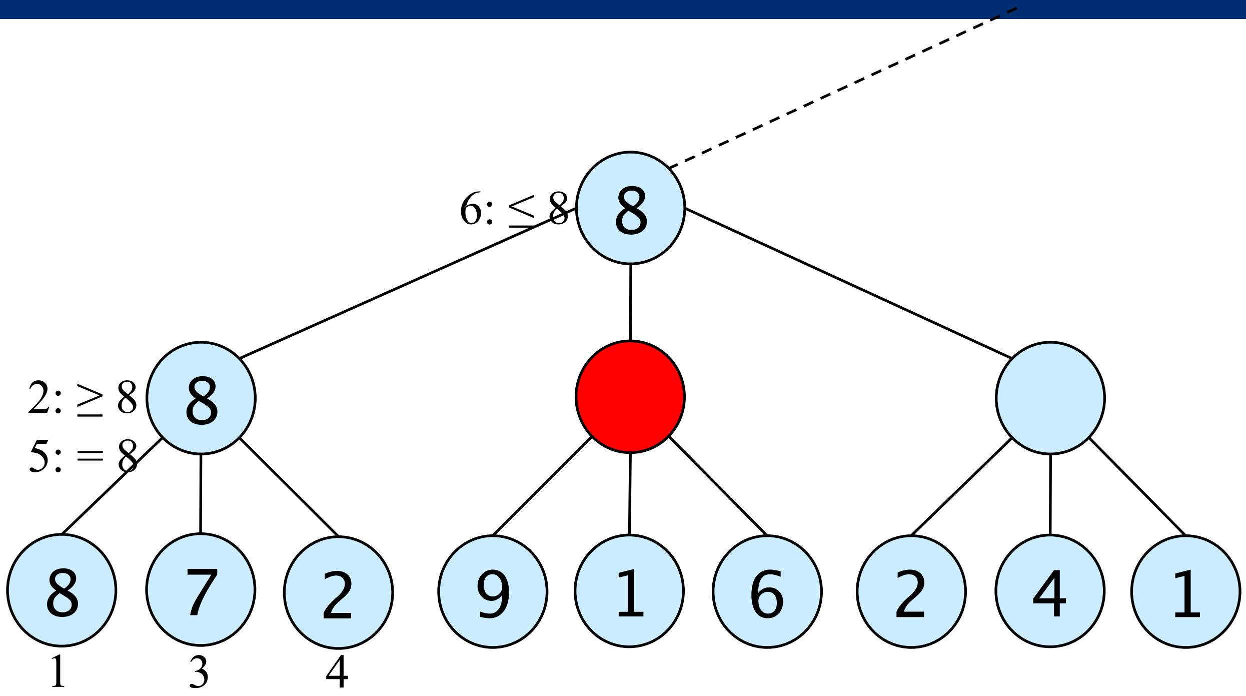
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

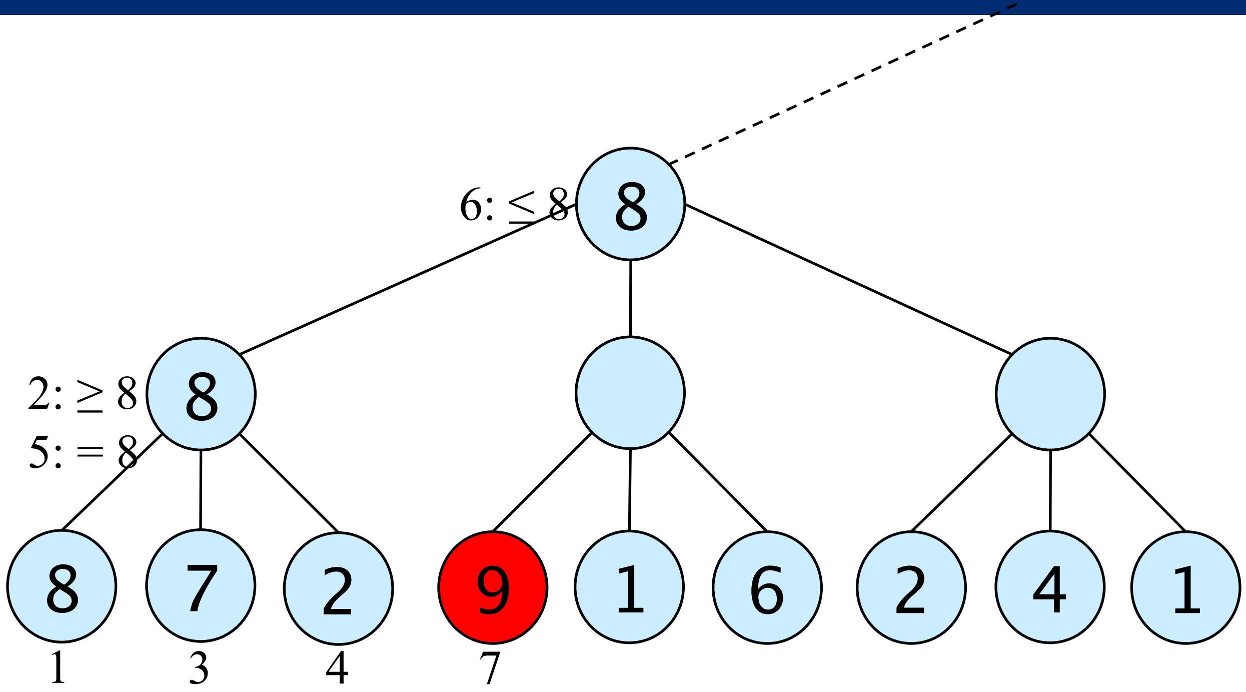
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

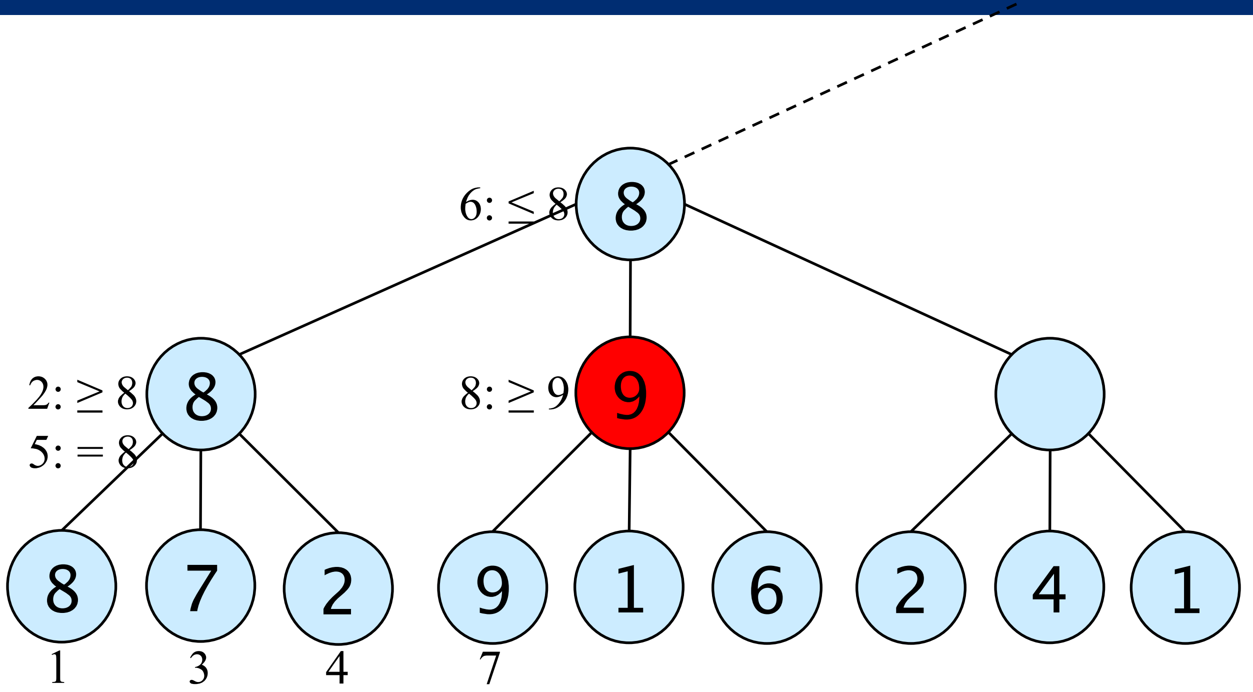
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

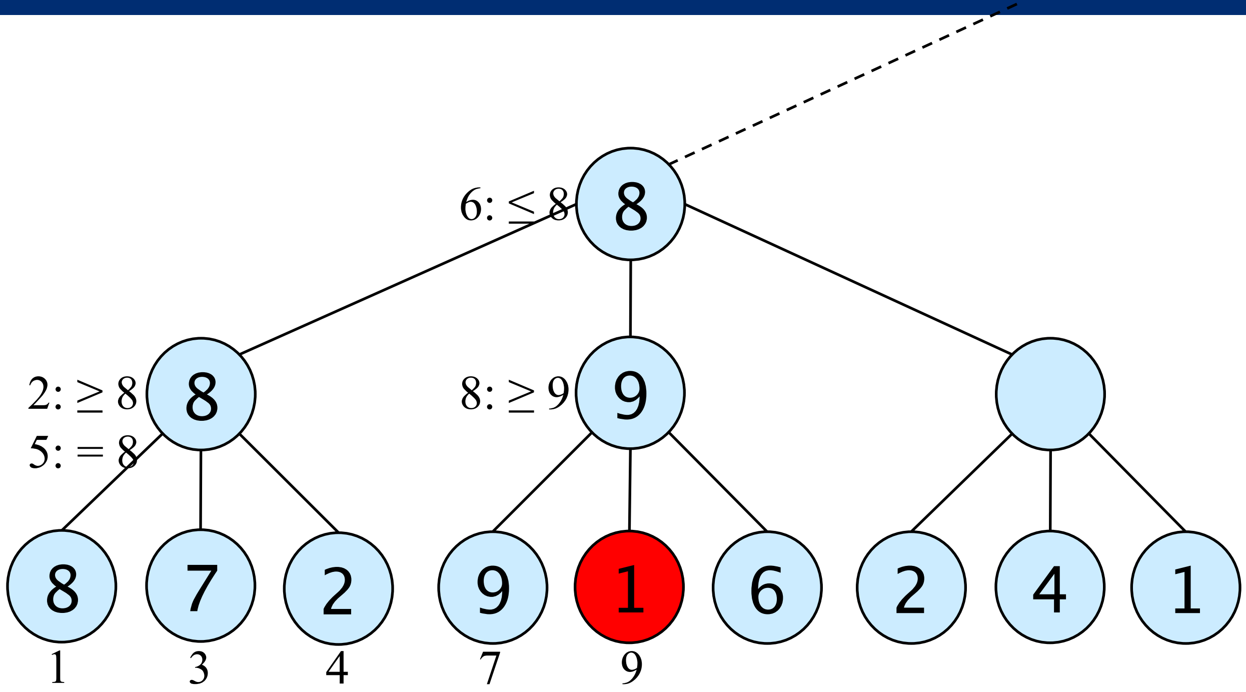
Maximizing Level



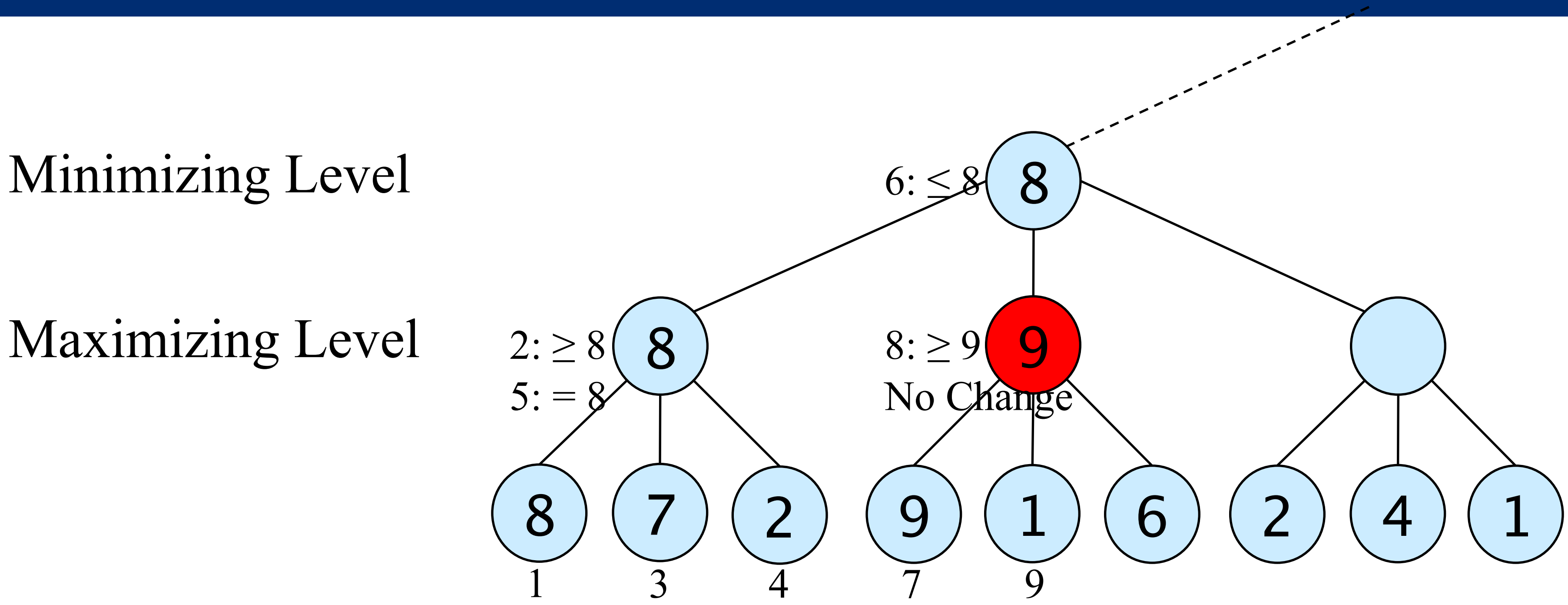
MINIMAX-DECISION Example Cont.

Minimizing Level

Maximizing Level



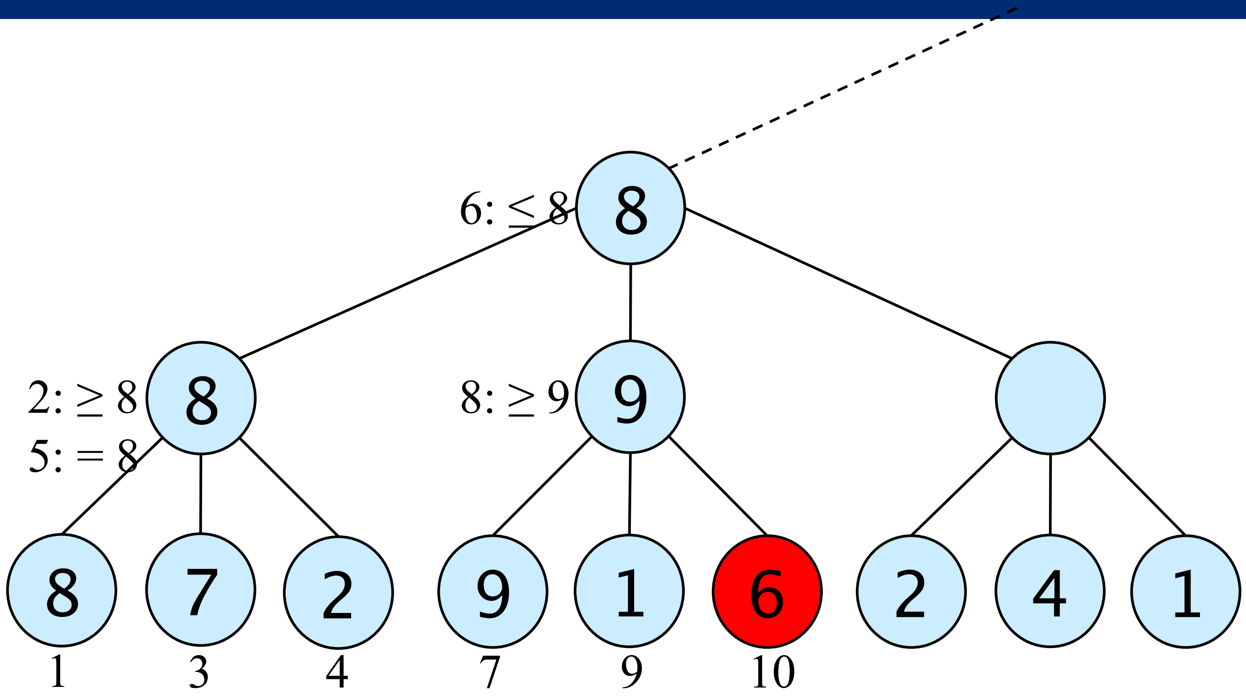
MINIMAX-DECISION Example Cont.



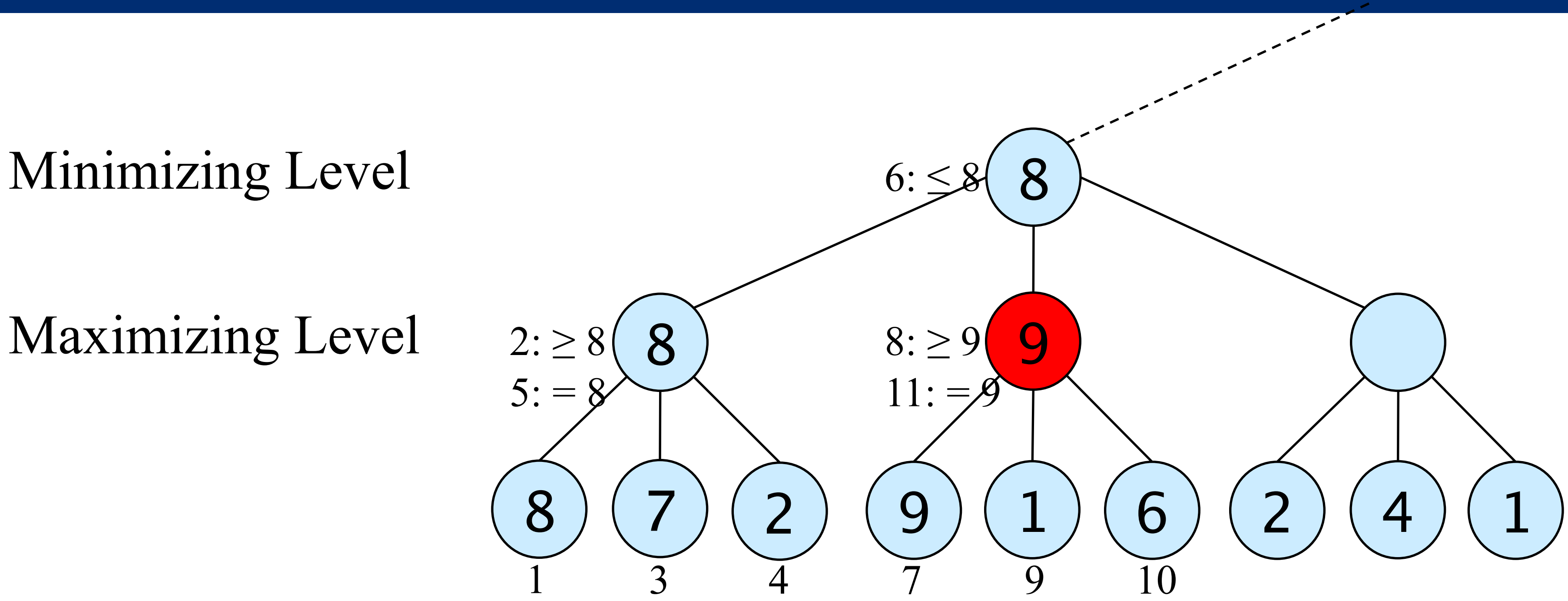
MINIMAX-DECISION Example Cont.

Minimizing Level

Maximizing Level



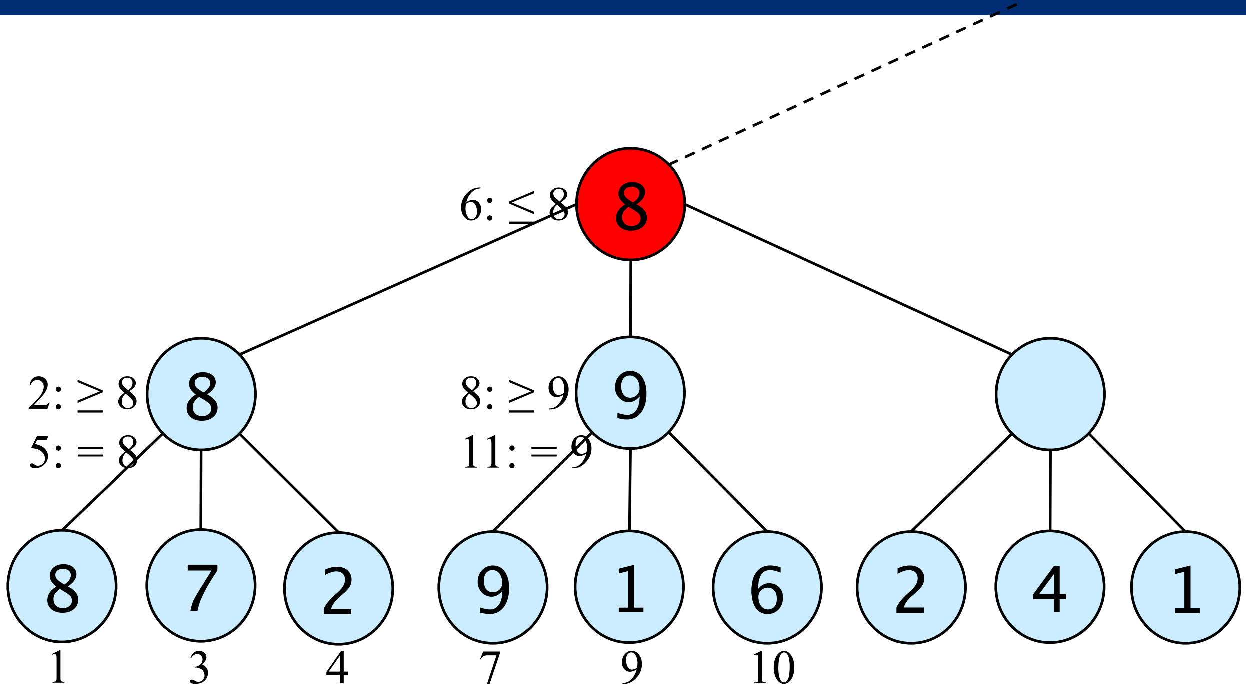
MINIMAX-DECISION Example Cont.



MINIMAX-DECISION Example Cont.

Minimizing Level

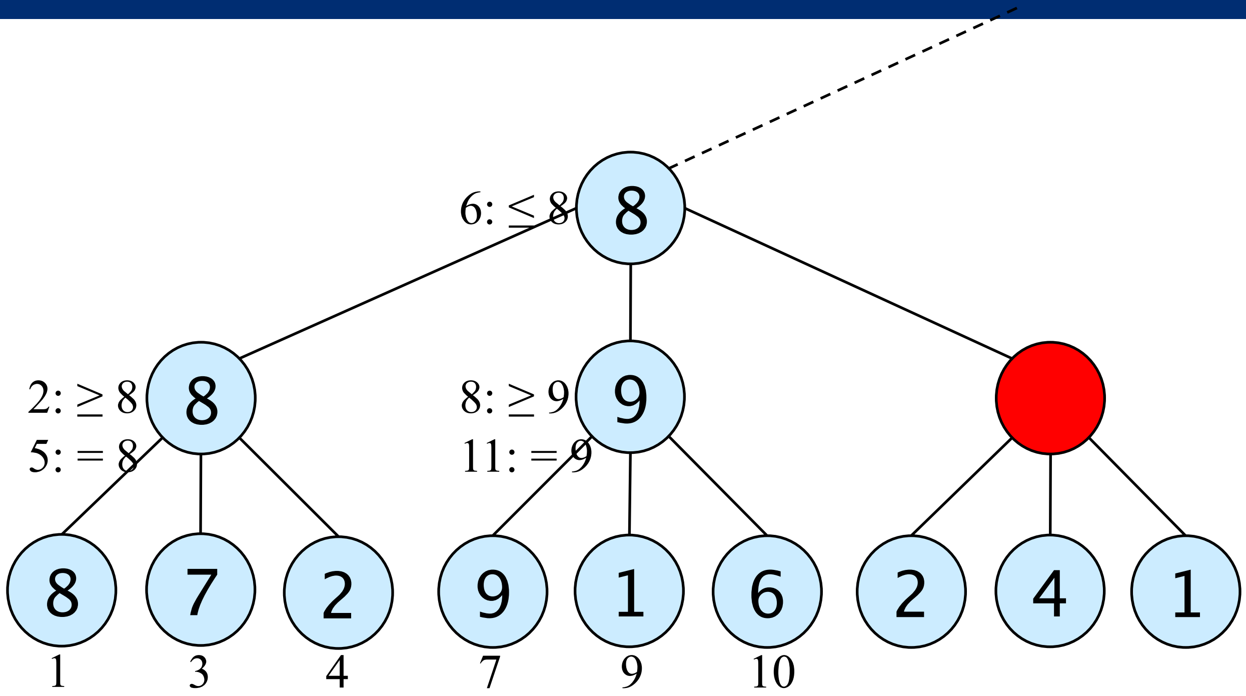
Maximizing Level



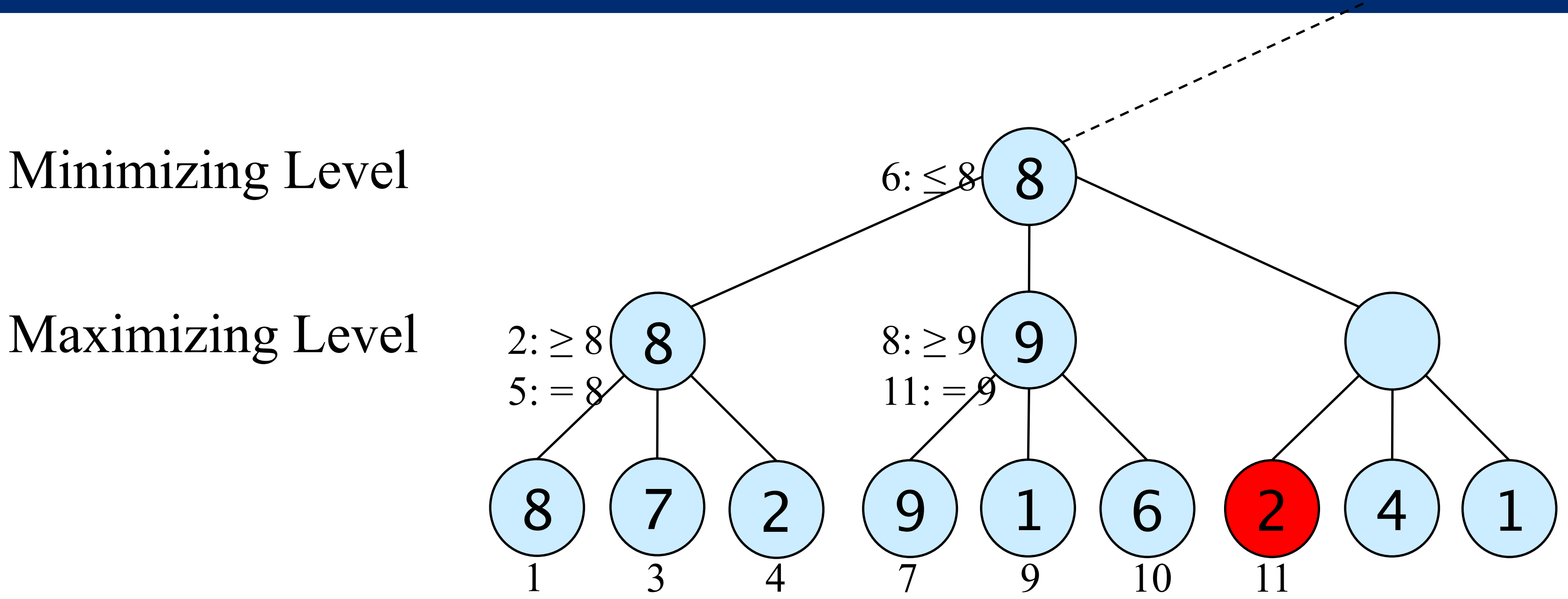
MINIMAX-DECISION Example Cont.

Minimizing Level

Maximizing Level



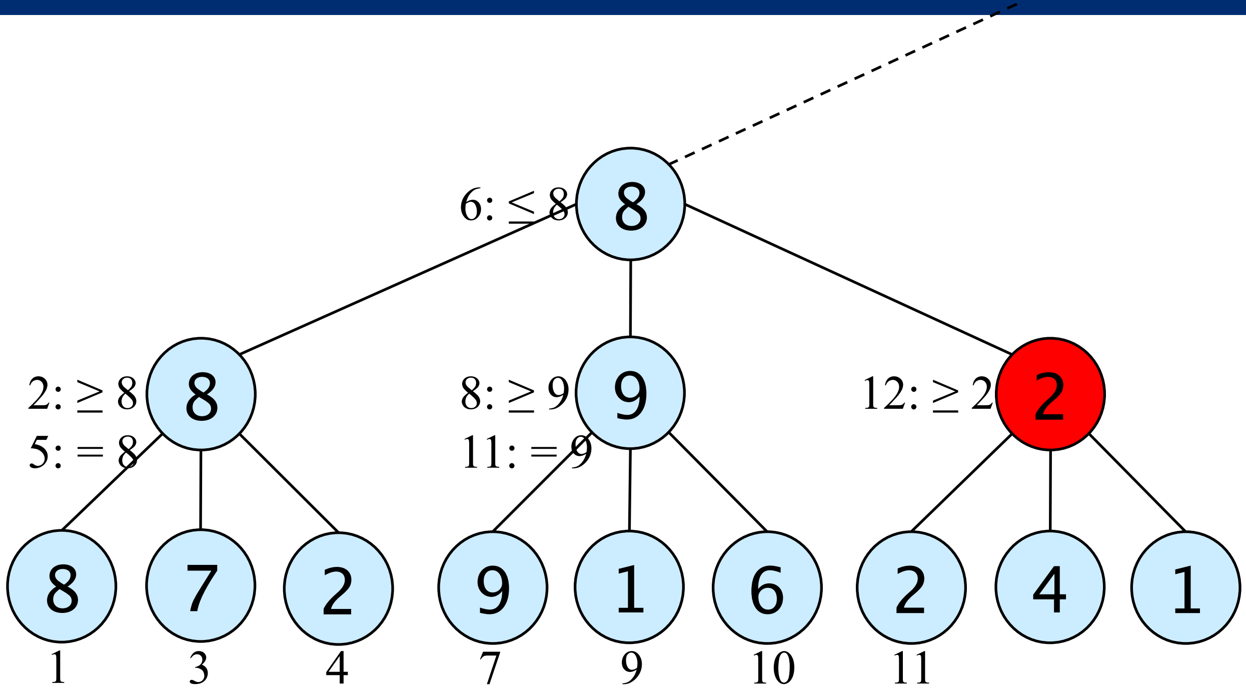
MINIMAX-DECISION Example Cont.



MINIMAX-DECISION Example Cont.

Minimizing Level

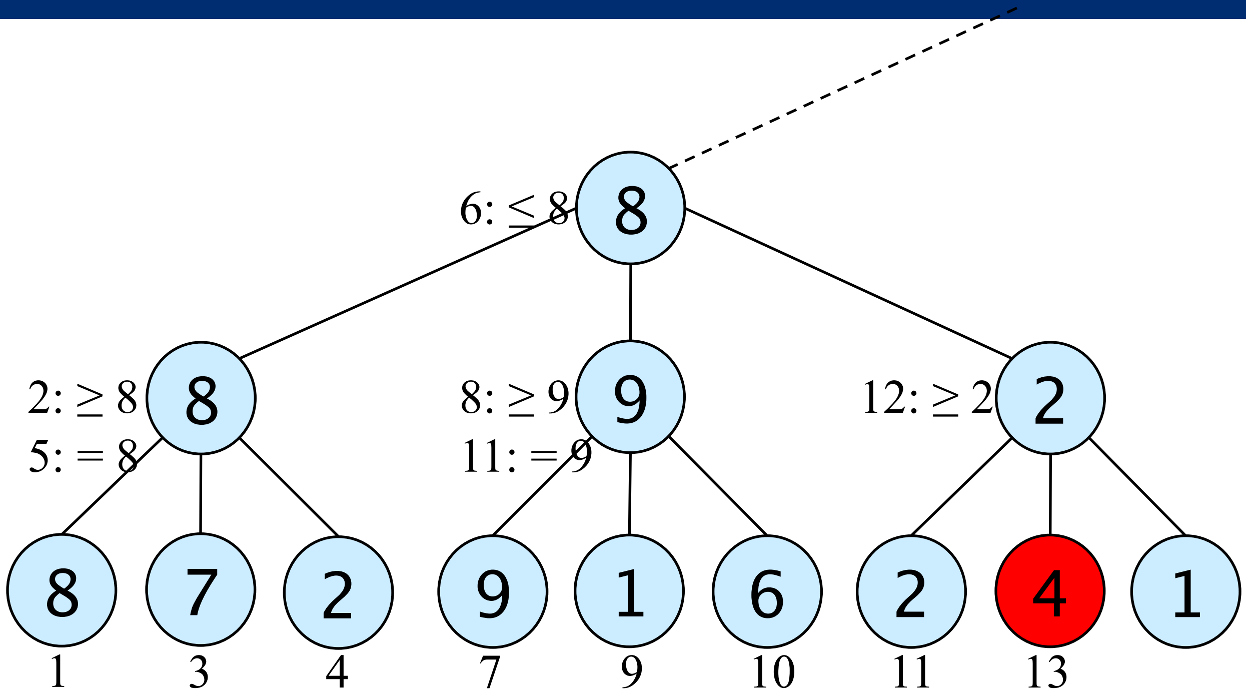
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

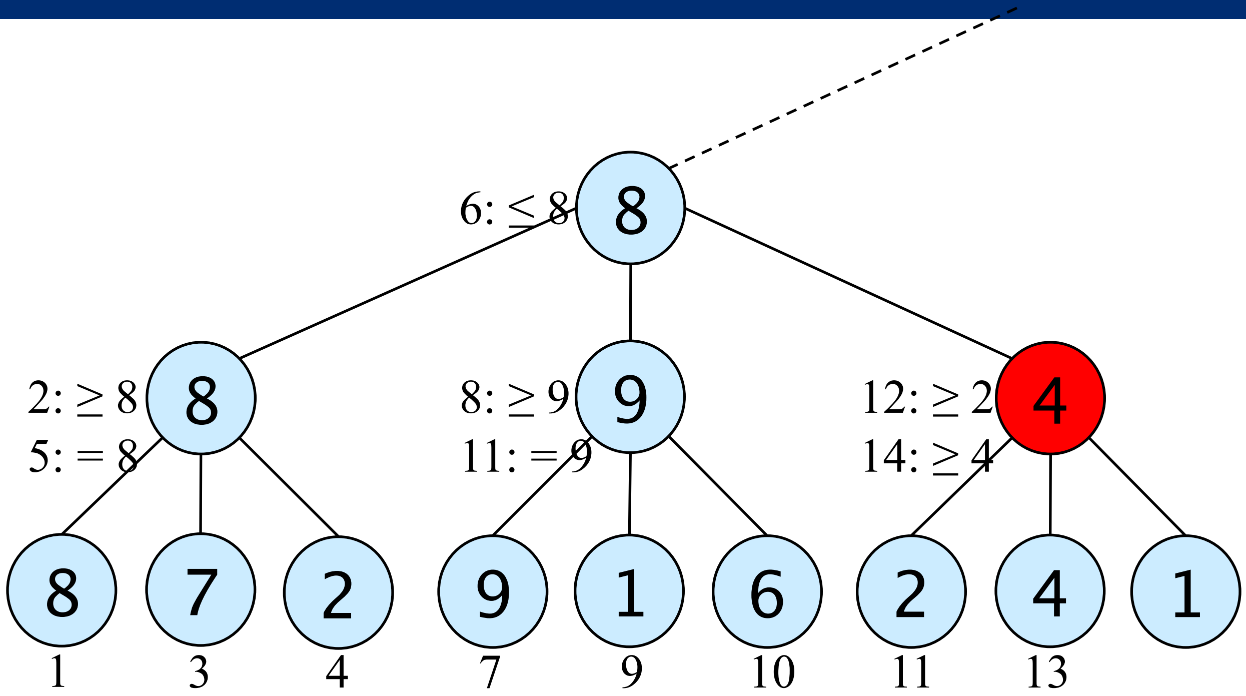
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

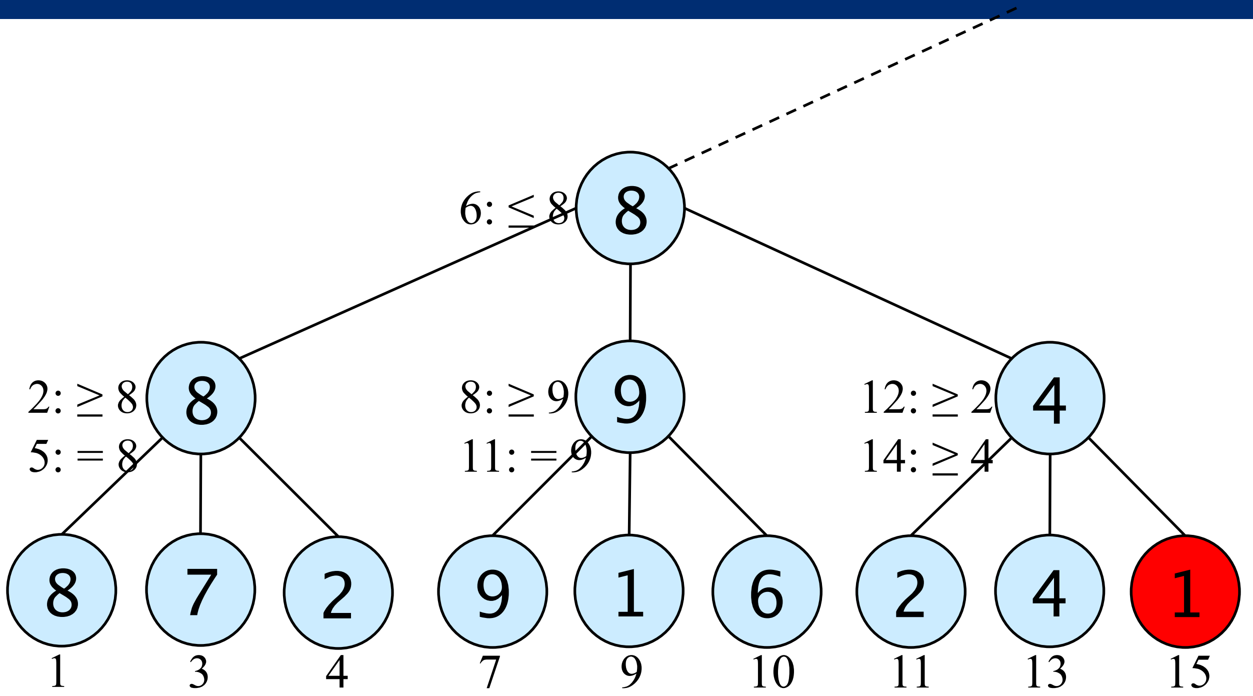
Maximizing Level



MINIMAX-DECISION Example Cont.

Minimizing Level

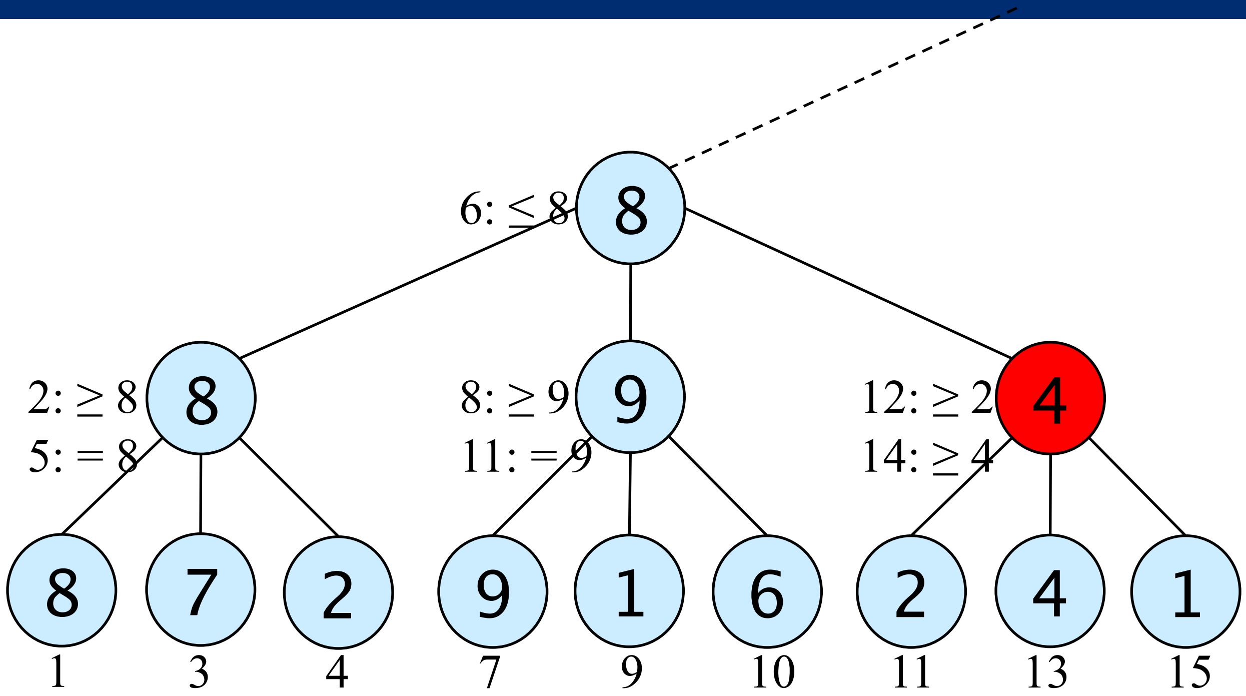
Maximizing Level



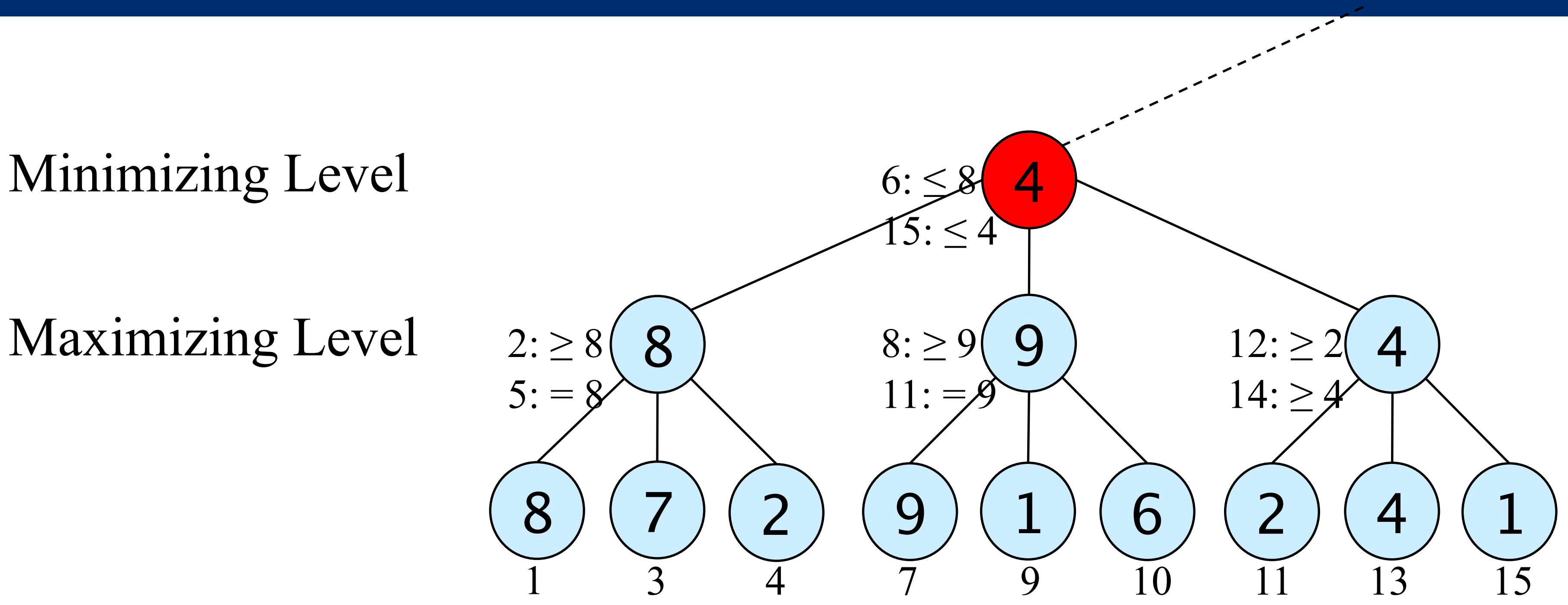
MINIMAX-DECISION Example Cont.

Minimizing Level

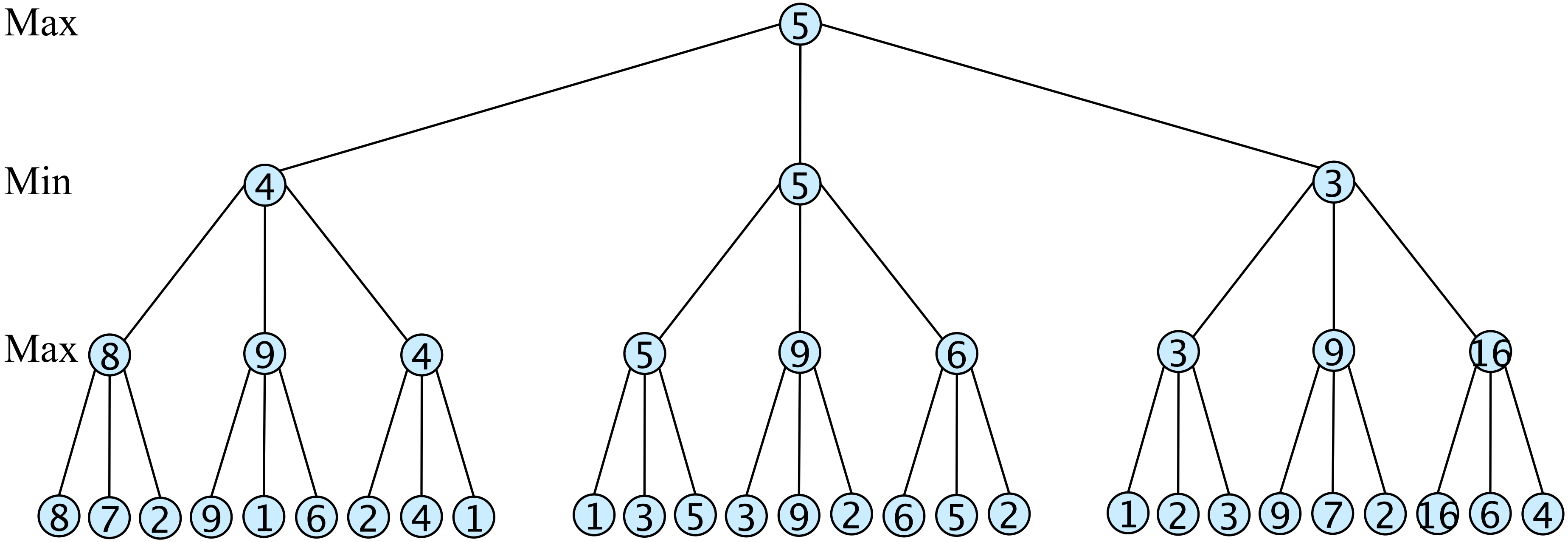
Maximizing Level



MINIMAX-DECISION Example Cont.



Complete Example of MINIMAX-DECISION

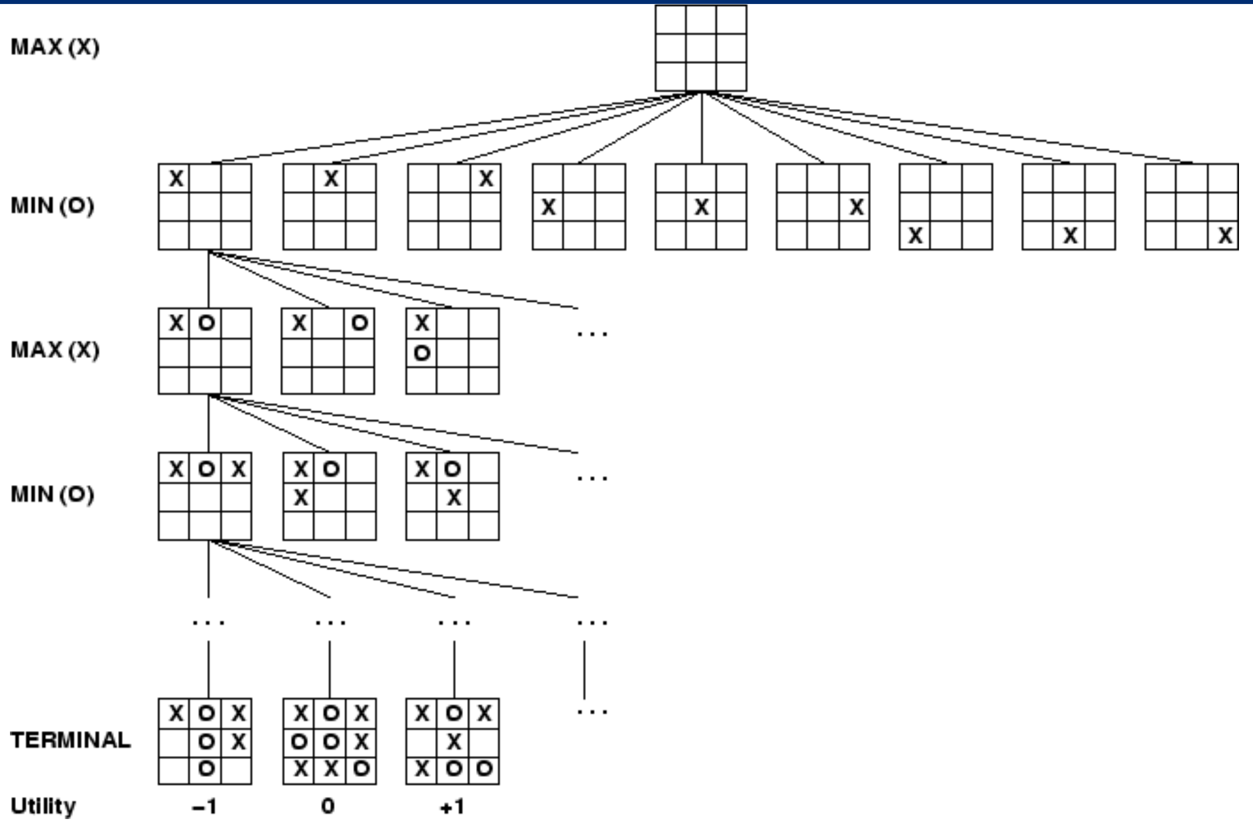


MINIMAX-DECISION Example Cont.

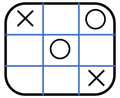
Partial Game Tree for Tic-Tac-Toe

- Two players, MAX and MIN

- In this case, assume we are searching ahead 5 moves (ply=5) Moves (and levels) alternate between two players



Russell and Norvig, 2020



$state = s1$

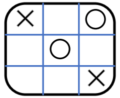
$player \leftarrow X$

```
function MIN-MAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

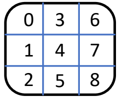
Numbers indicates unique location on the board



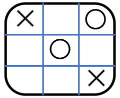
state = s1
player ← X

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.



Numbers indicates unique location on the board



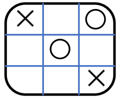
$state = s1$
$player \leftarrow$ X
$v = -\infty$

```
function MIN-MAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



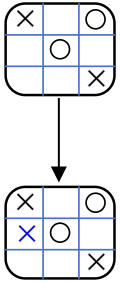
state = *s*1
player ← X
$v = -\infty$
game.Actions(*s*1) → {1,2,3,5,7}

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



state = s1
player ← X
v = −∞
game.Actions(s1) → {1,2,3,5,7}

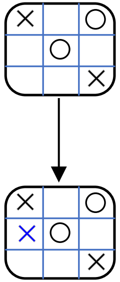
a = 1
*s*2 = *game*.Results(*s*1,1)
player = X

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
     if v2 > v then
        v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
     if v2 < v then
        v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



state = s1
player ← X
v = −∞
game.Actions(s1) → {1,2,3,5,7}

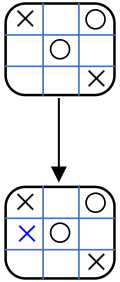
a = 1
*s*2 = *game*.Results(s1,1)
game.Is-Terminal(*s*1) → False

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

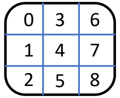


state = s1
player ← X
v = −∞
game.Actions(s1) → {1,2,3,5,7}

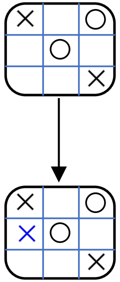
a = 1
*s*2 = *game*.Results(s1,1)
v = +∞

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
     if v2 > v then
        v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
     if v2 < v then
        v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.



Numbers indicates unique location on the board



state = s1
player ← X
v = −∞
game.Actions(s1) → {1,2,3,5,7}

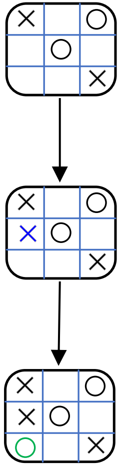
a = 1
s2 = *game*.Results(s1,1)
v = +∞
game.Actions(s2) → {2,3,5,7}

```
function MIN-MAX-SEARCH(game, state) returns an action
   player ← game.TO-MOVE(state)
   value, move ← MAX-VALUE(game, state)
   return move

function MAX-VALUE(game, state) returns a (utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← −∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
      if v2 > v then
         v, move ← v2, a
   return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
   if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
   v ← +∞
   for each a in game.ACTIONS(state) do
      v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
      if v2 < v then
         v, move ← v2, a
   return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



state = s1
player ← X
v = −∞
game.Actions(s1) → {1,2,3,5,7}

a = 1
s2 = *game*.Results(s1,1)
v = +∞
game.Actions(s2) → {2,3,5,7}

a = 2
*s*3 = *game*.Results(*s*2, 2)

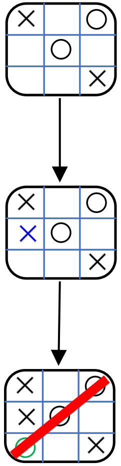
```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **Depth-first search** nature in Min-Max Search so far



state = s1
player ← X
$v = -\infty$
game.Actions(*s*1) → {1,2,3,5,7}

a = 1
*s*2 = *game*.Results(*s*1,1)
$v = +\infty$
game.Actions(*s*2) → {2,3,5,7}

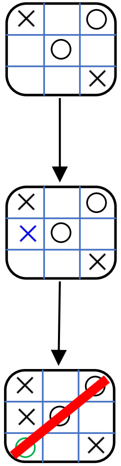
a = 2
*s*3 = *game*.Results(*s*2,2)
(*v*2,*a*2) = (-1,*null*)
game.Is-Terminal(*s*3) → True
game.Utility(*s*3,O) → (-1, null)

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



state = s1
player ← X
$v = -\infty$
game.Actions(s1) → {1,2,3,5,7}

a = 1
s2 = *game*.Results(s1,1)
$v = +\infty$
game.Actions(s2) → {2,3,5,7}

a = 2
s3 = *game*.Results(s2,2)
(v2,a2) = (-1,*null*)
$v2 = -1 < v = +\infty$
v = -1, *move* = 2

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move
```

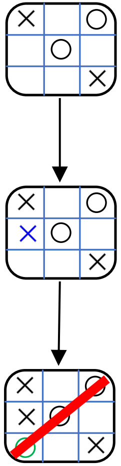
```
function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
     if v2 > v then
        v, move ← v2, a
  return v, move
```

```
function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
     if v2 < v then
        v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



$state = s1$
$player \leftarrow X$
$v = -\infty$
$game.\text{Actions}(s1) \rightarrow \{1,2,3,5,7\}$

$a = 1$
$s2 = game.\text{Results}(s1,1)$
$v = +\infty$
$game.\text{Actions}(s2) \rightarrow \{2,3,5,7\}$

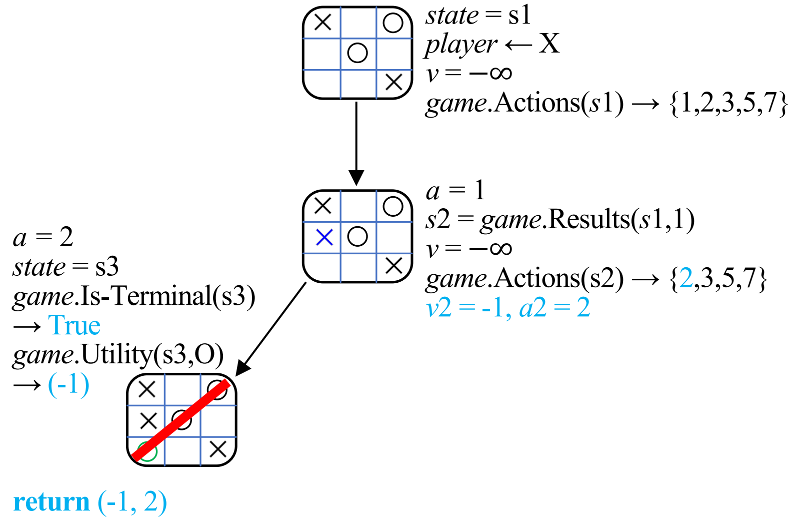
$a = 2$
$s3 = game.\text{Results}(s2,2)$
$(v2,a2) = (-1,null)$
$v2 = -1 < v = +\infty$
return (-1, 2)

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
     if v2 > v then
        v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
     v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
     if v2 < v then
        v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board



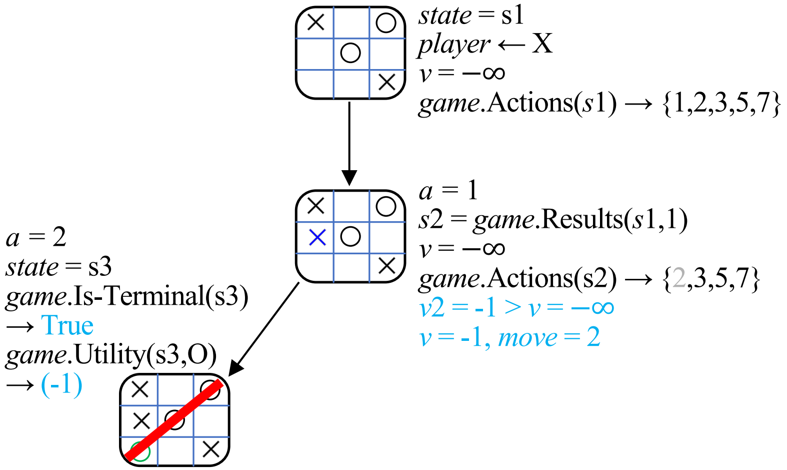

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



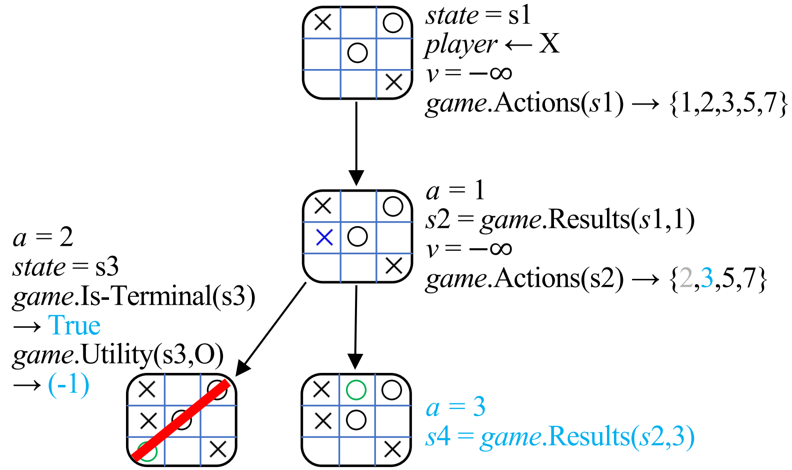

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



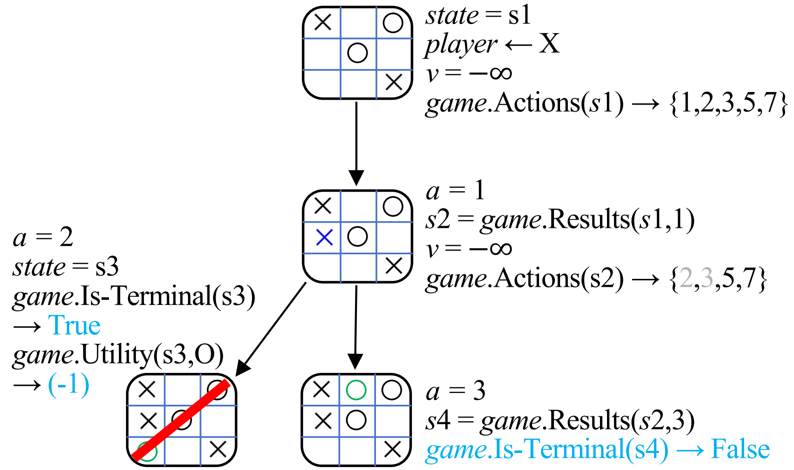

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



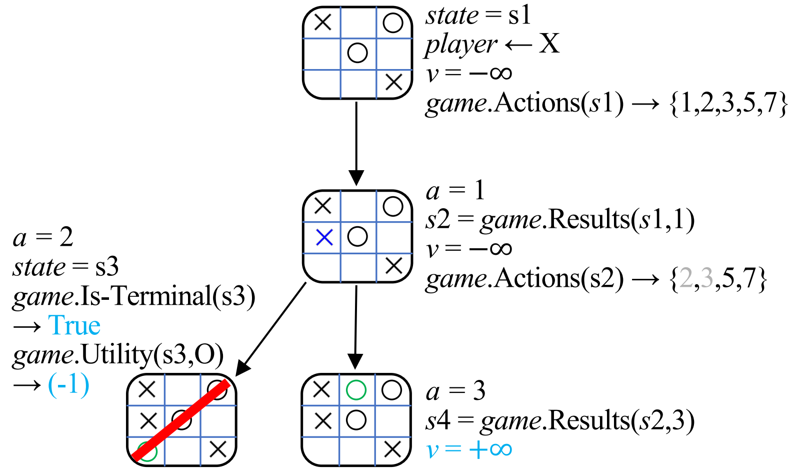

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



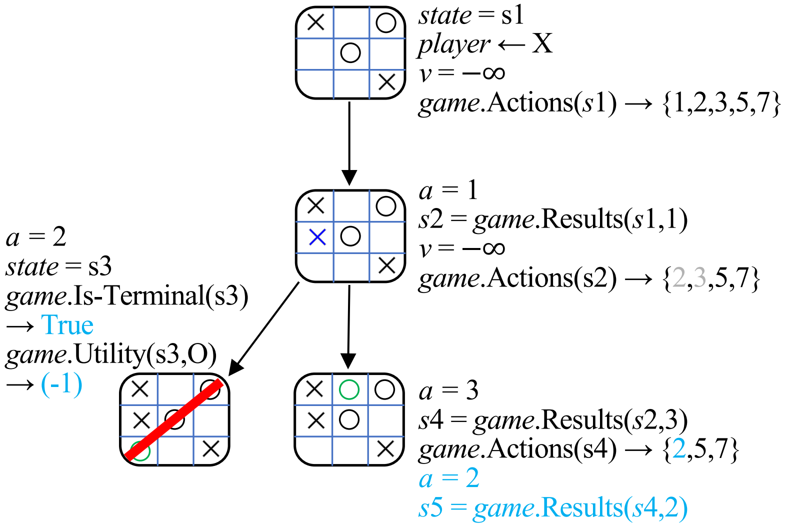

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



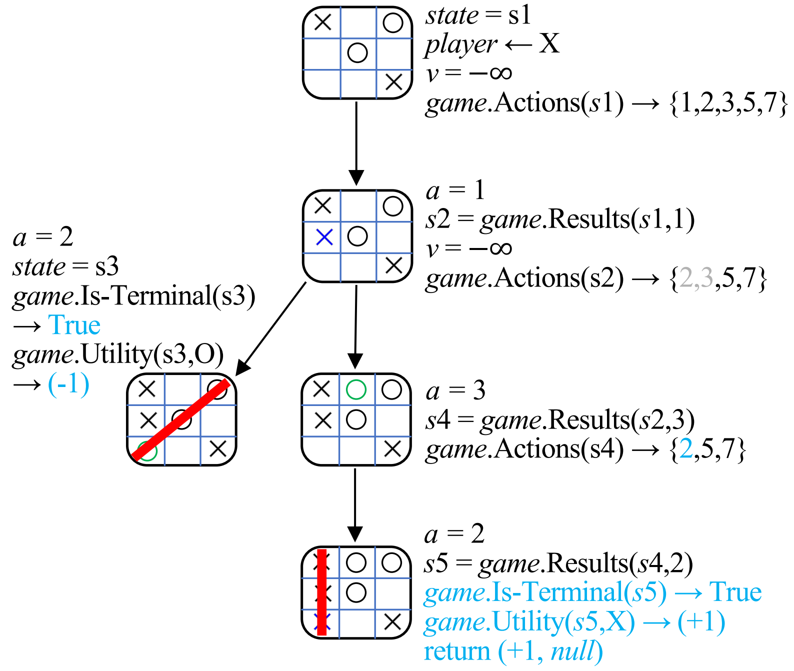

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



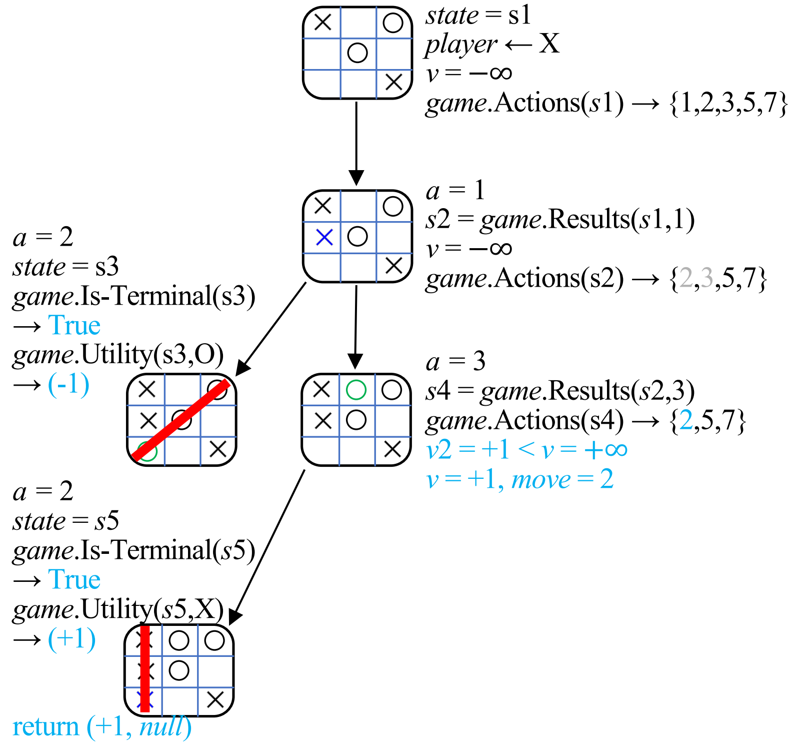

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



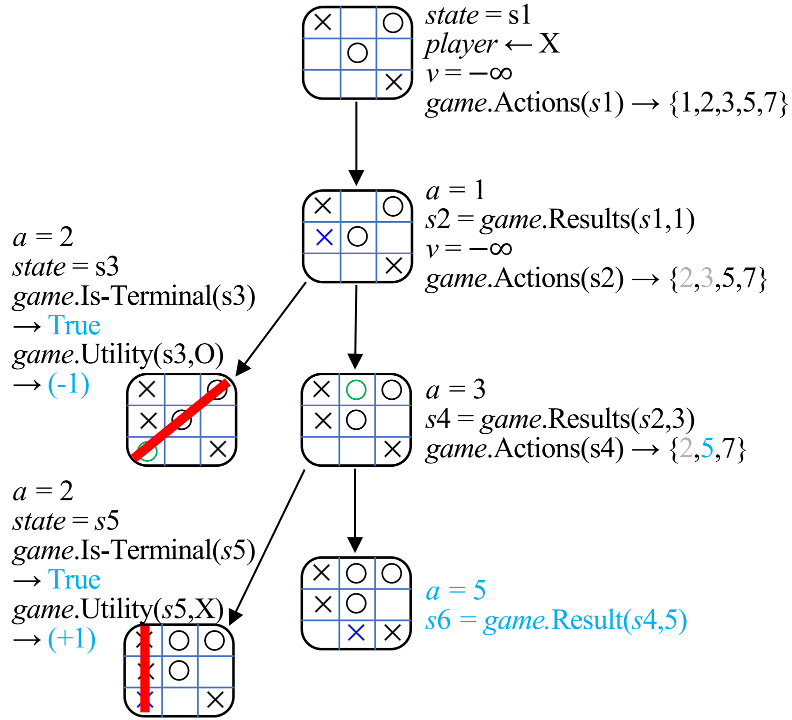

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



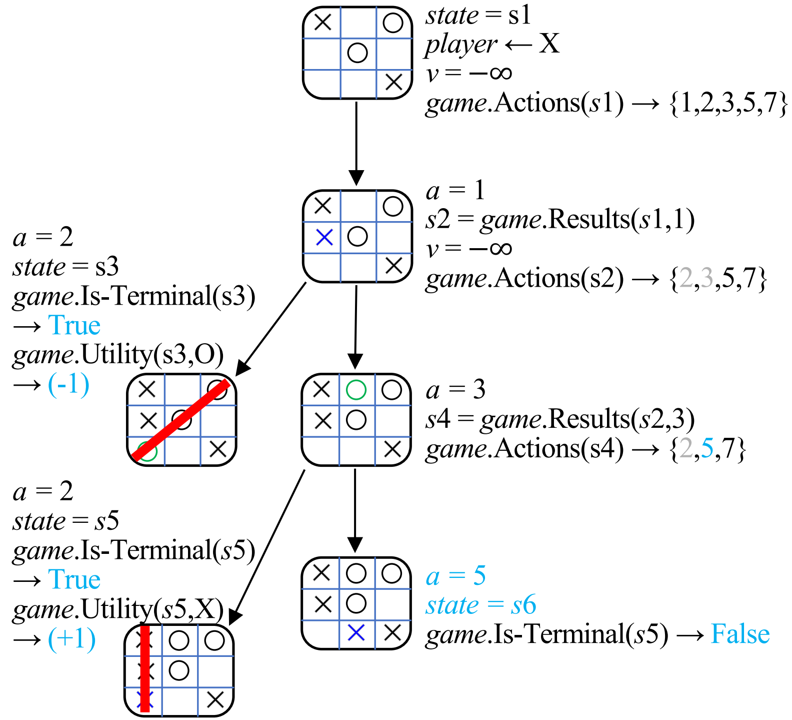

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



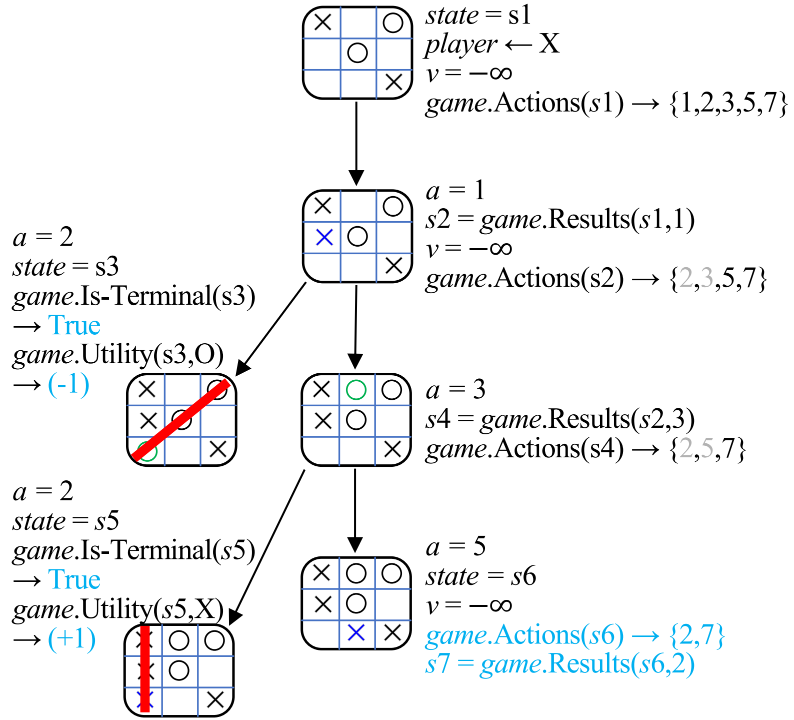

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

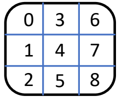
Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



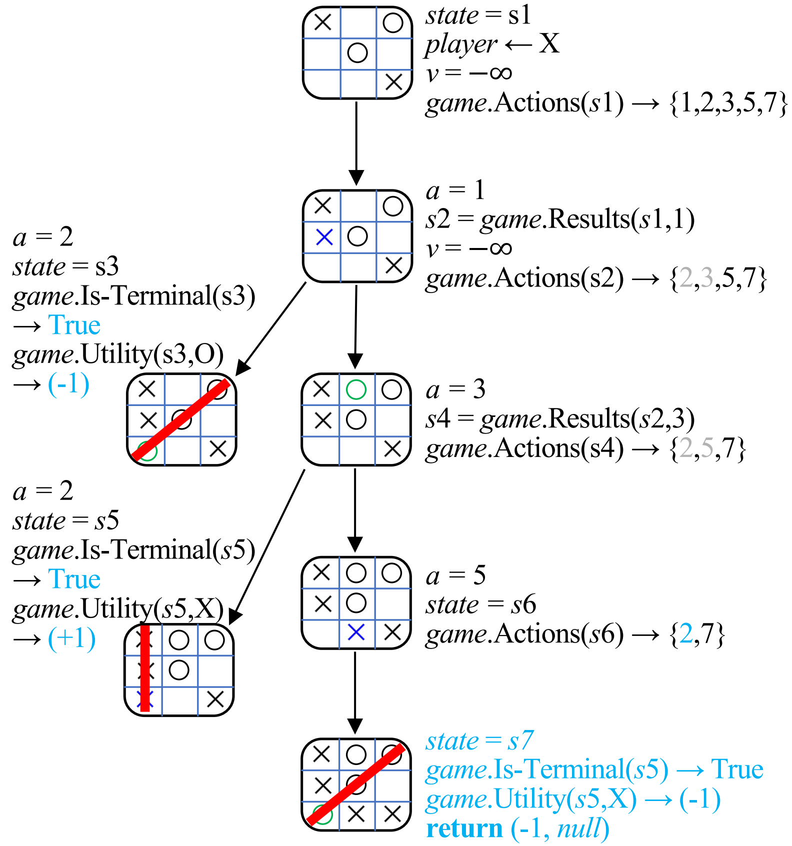

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.



Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



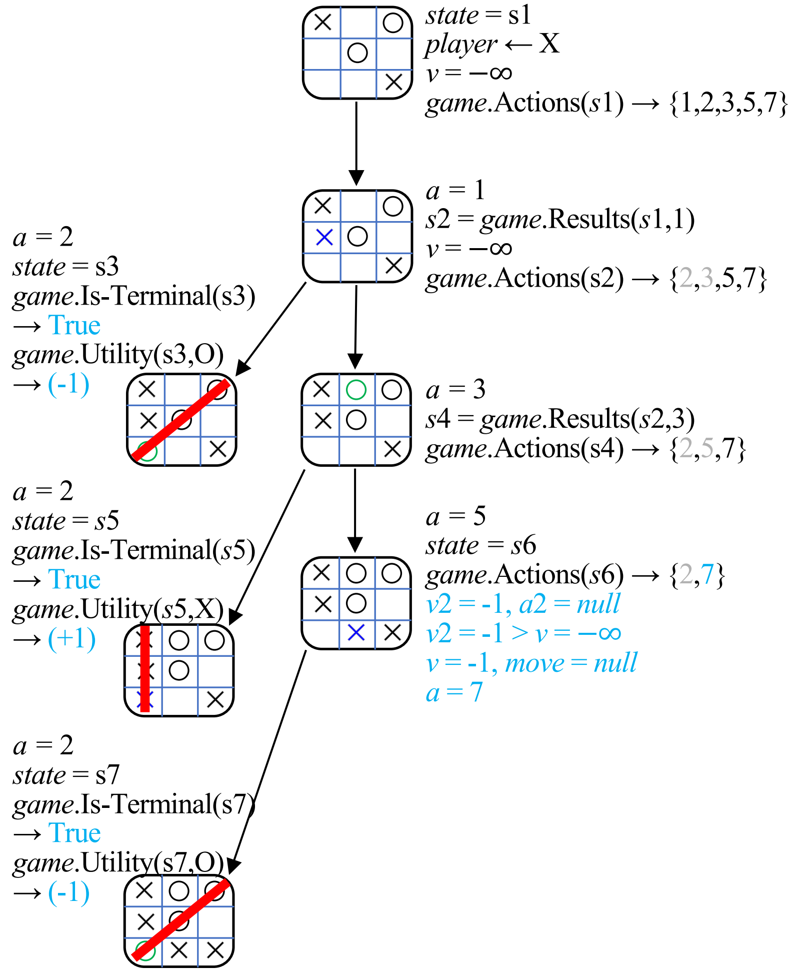

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We can see the **recursion** nature in Min-Max Search so far



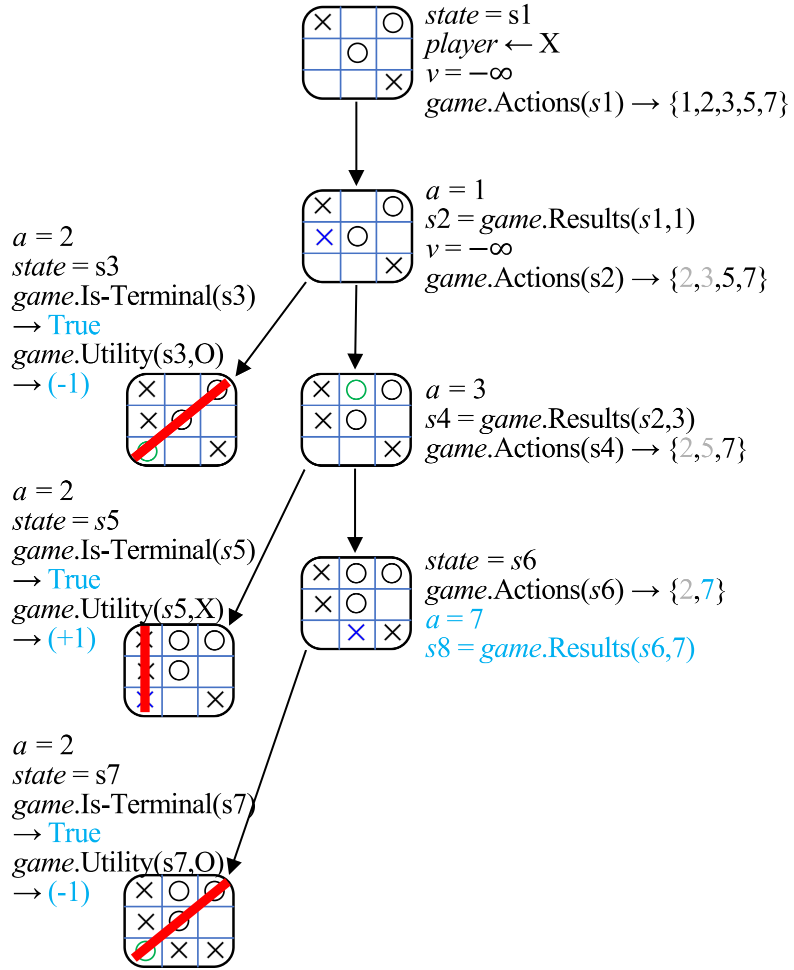

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s7* since it is similar case to the *s3*



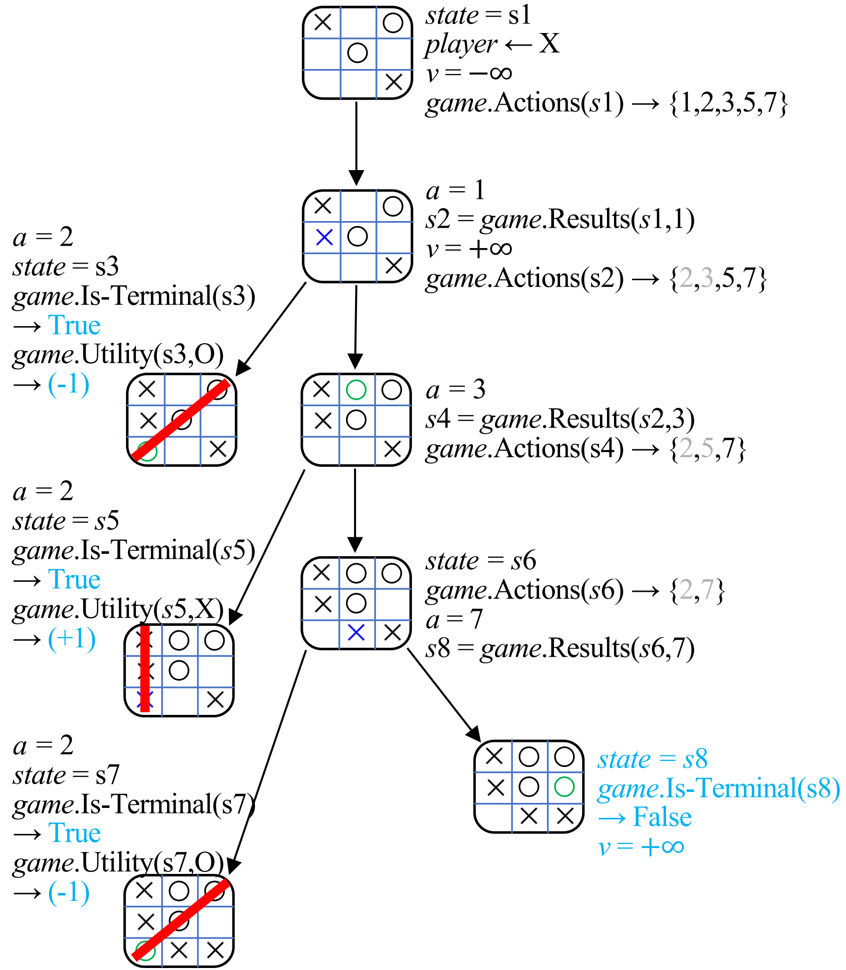

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s*7 since it is similar case to the *s*3



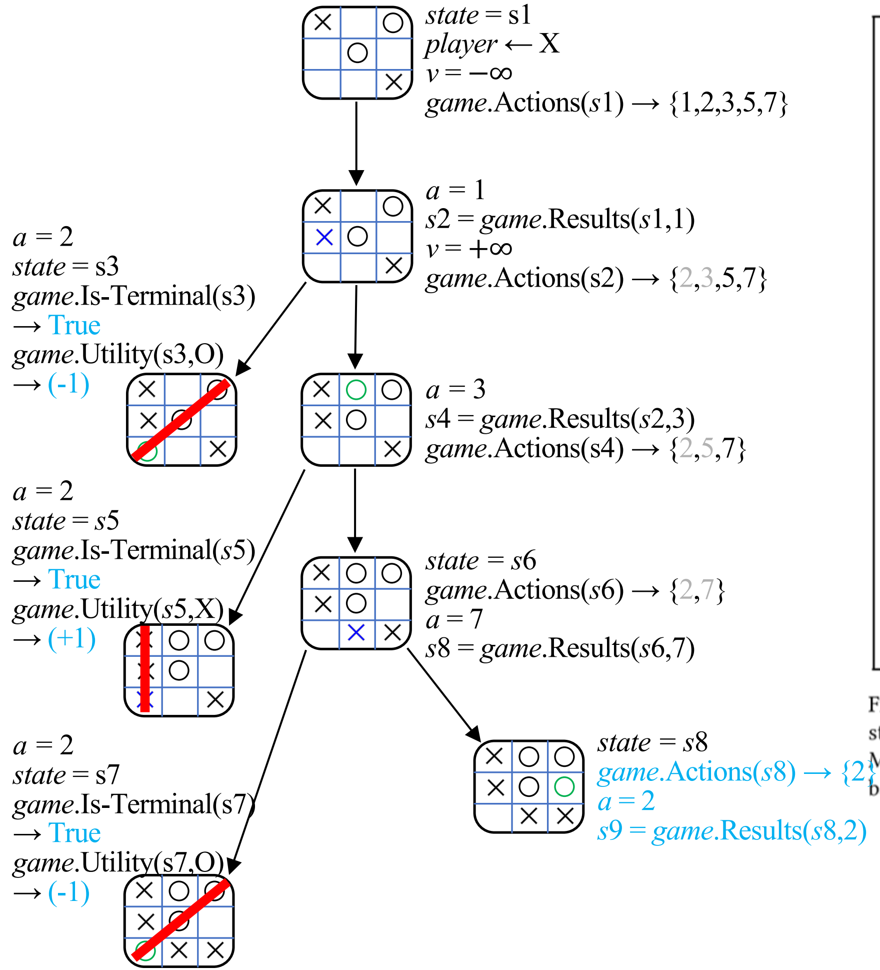

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s*7 since it is similar case to the *s*3



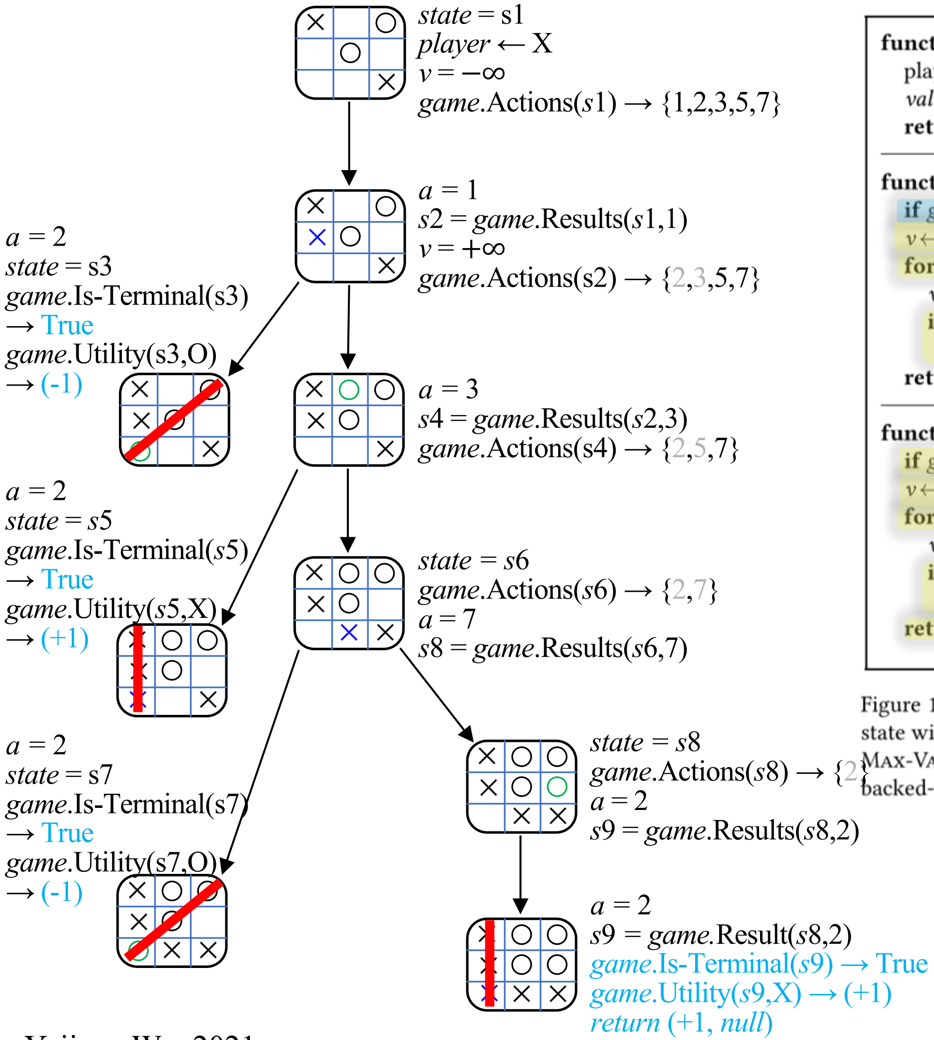

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s7* since it is similar case to the *s3*



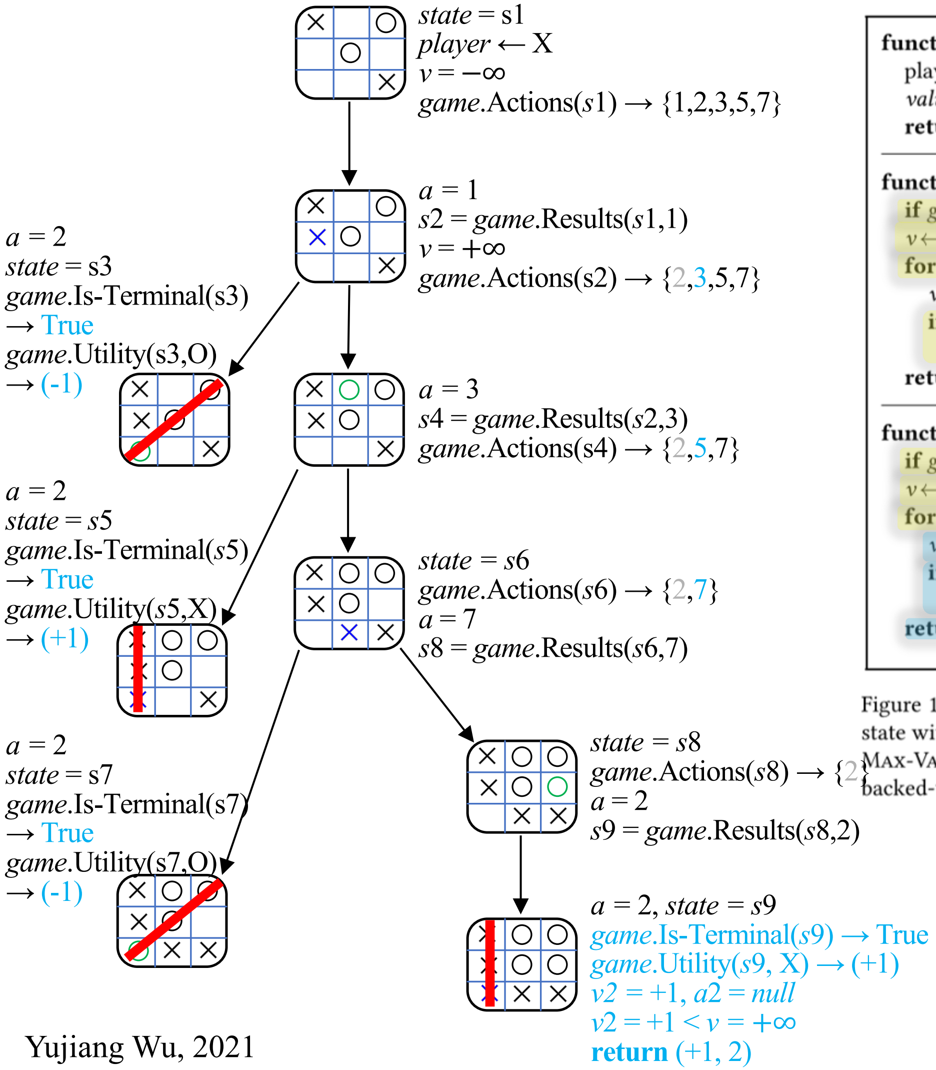

```
function MIN-MAX-SEARCH(game, state) returns an action
  player ← game.TO-MOVE(state)
  value, move ← MAX-VALUE(game, state)
  return move

function MAX-VALUE(game, state) returns a (utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← −∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MIN-VALUE(game, game.RESULT(state, a))
    if v2 > v then
      v, move ← v2, a
  return v, move

function MIN-VALUE(game, state) returns returns a(utility, move) pair
  if game.IS-TERMINAL(state) then return game.UTILITY(state, player), null
  v ← +∞
  for each a in game.ACTIONS(state) do
    v2, a2 ← MAX-VALUE(game, game.RESULT(state, a))
    if v2 < v then
      v, move ← v2, a
  return v, move
```

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s*9 since it is similar case to the *s*5



function MIN-MAX-SEARCH(*game, state*) **returns** *an action*
  player ← *game*.TO-MOVE(*state*)
  value, move ← MAX-VALUE(*game, state*)
  return *move*

---

function MAX-VALUE(*game, state*) **returns** a (*utility, move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*
  $v \leftarrow -\infty$
  for each *a* in *game*.ACTIONS(*state*) **do**
    v2, a2 ← MIN-VALUE(*game, game*.RESULT(*state, a*))
    if *v2 > v* **then**
      v, move ← *v2, a*
  return *v, move*

---

function MIN-VALUE(*game, state*) **returns returns** a(*utility, move*) pair
  if *game*.IS-TERMINAL(*state*) **then return** *game*.UTILITY(*state, player*), *null*
  $v \leftarrow +\infty$
  for each *a* **in** *game*.ACTIONS(*state*) **do**
    v2, a2 ← MAX-VALUE(*game, game*.RESULT(*state, a*))
    if *v2 < v* **then**
      v, move ← *v2, a*
  return *v, move*

Figure 1: An algorithm for calculating the optimal move using min-max - the move that leads to a terminal state with maximum utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state and the move to get there.

0	3	6
1	4	7
2	5	8

Numbers indicates unique location on the board

We will skip examining the recursion call for *s9* since it is similar case to the *s5*

Need to show the backing out of the recursion