Transpositions & Isomorphisms

CMPUT 355: Games, Puzzles, and Algorithms

Lecture Outline

1. Logistics & Recap
2. Negamax + alpha-beta
3. Transpositions
4. Isomorphisms

Logistics

- **Quiz 2** is being marked
  - Scans of your exam will be attached to the Canvas "assignment"
- **Quiz 1:** deadline for **regrading requests** is this Friday (Feb 13)
- **Practice questions #3** will be released **this Friday** (Feb 13)
- **Reading week: No classes** next week (Feb 16 -- Feb 20)

Recap: Minimax, Negamax, Alpha-beta

```
assume P1 plays at root
assume players alternate turns

def score(s):
    return P1's score at state s

def minimax(s):
    if terminal(s):
        return score(s)
    if player(s) == 1:
        return max{minimax(c) for all c in children(s)}
    if player(s) == 2:
        return min{minimax(c) for all c in children(s)}
```

```
assume P1 plays at root
assume players alternate turns

def ptm_score(s):
    return player-to-move's score at state s

def negamax(s):
    if terminal(s):
        return ptm_score(s)
    return max{-negamax(c) for c in children(s)}
```

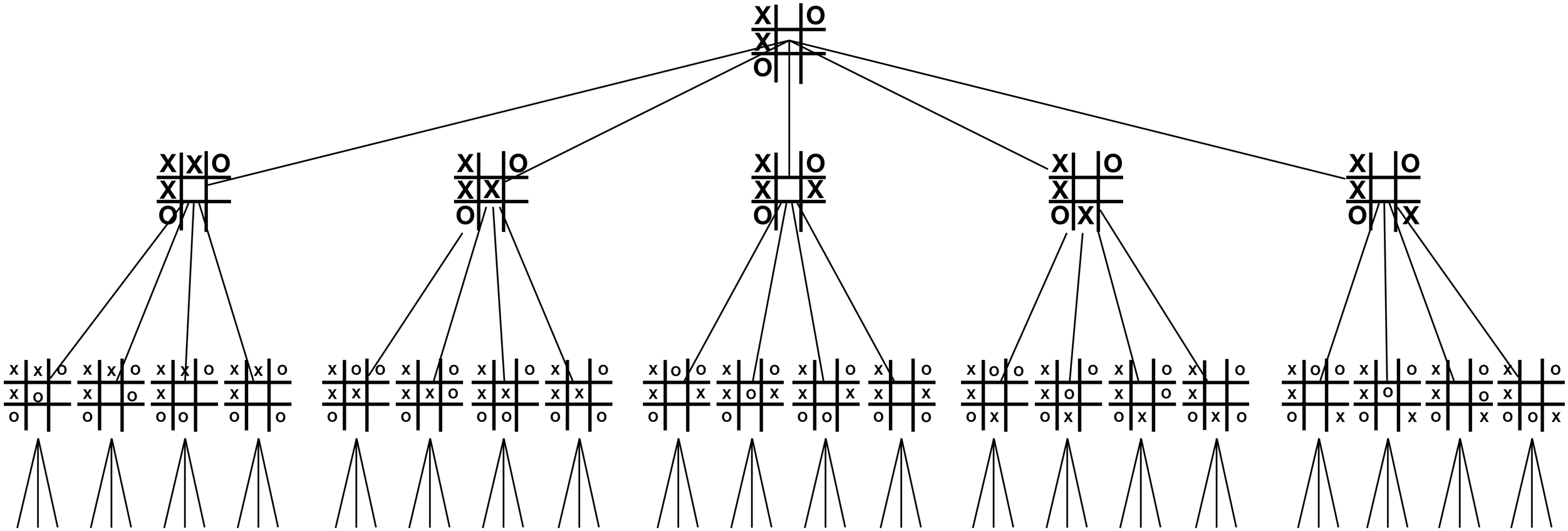
```
def alphabeta(s, alpha, beta):
  if terminal(s):
    return score(s)
  if player(s) = 1: # MAX player
    val = -inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      alpha = max(alpha, ab)
      val = max(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  if player(s) = 2:
    val = inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      beta = min(beta, ab)
      val = min(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  return val
```

Negamax + Alpha-beta

```
def alphabeta(s, alpha, beta):
  if terminal(s):
    return score(s)
  if player(s) = 1: # MAX player
    val = -inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      alpha = max(alpha, ab)
      val = max(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  if player(s) = 2:
    val = inf
    for c in children(s):
      ab = alphabeta(c, alpha, beta)
      beta = min(beta, ab)
      val = min(val, ab)
      if alpha >= beta:
        return val # prune remaining children
  return val
```

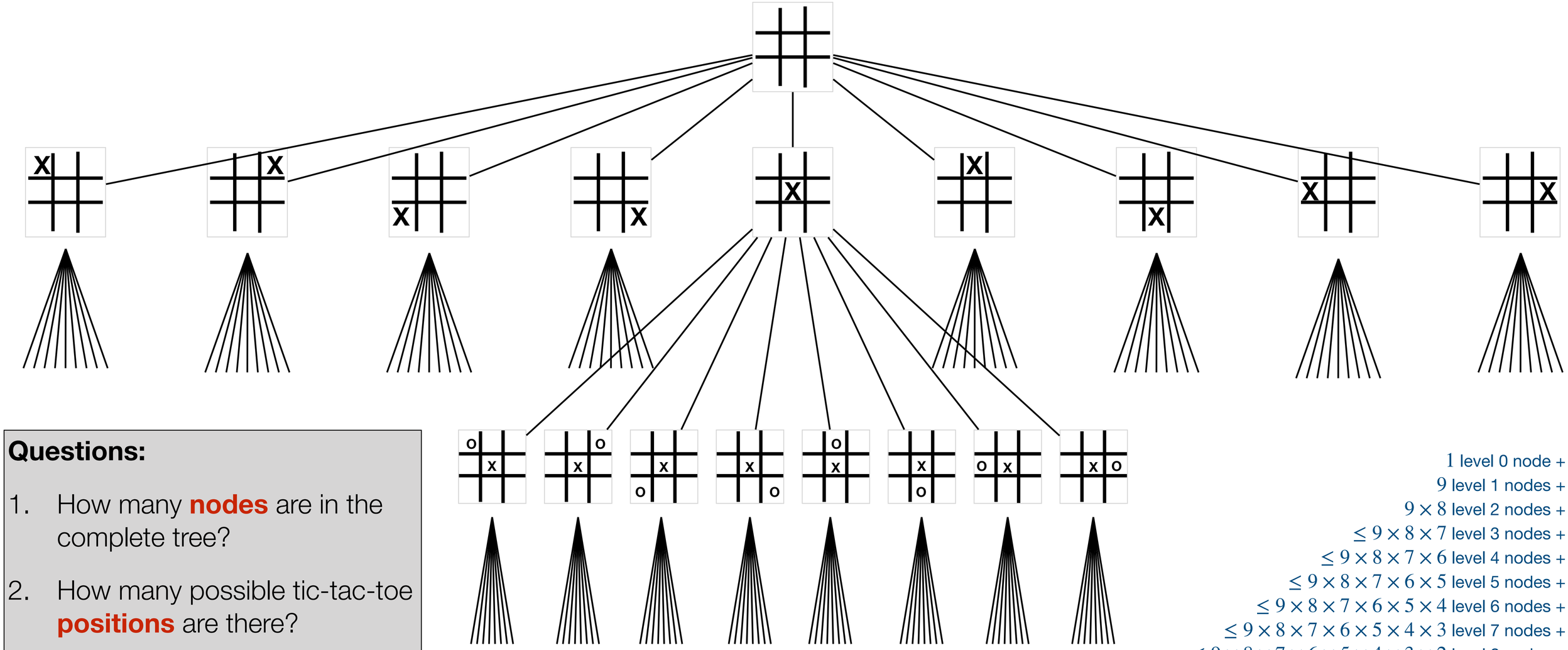
```
def ptm_score(s):
    return player-to-move's score at state s

def abnegamax(s, alpha, beta):
    if terminal(s):
        return ptm_score(s)
    so_far = -inf
    for c in children(s):
        abn = -abnegamax(c, -beta, -alpha)
        so_far = max(so_far, abn)
        alpha = max(alpha, abn)
        if alpha >= beta:
            return so_far # prune
    return so_far
```

Recap: Tic-tac-toe Game Tree



≤ 1 **node at root** +
≤ 5 **nodes at level 1** +
$\leq 5 \times 4$ **nodes at level 2** +
$\leq 5 \times 4 \times 3$ **nodes at level 3** +
$\leq 5 \times 4 \times 3 \times 2$ **nodes at level 4** +
$\leq 5 \times 4 \times 3 \times 2 \times 1$ **nodes at level 5** $= 326$

Full Tic-tac-toe Game Tree



Questions:

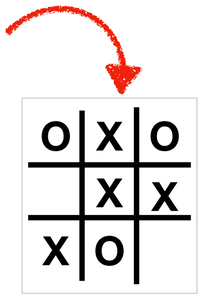
1. How many **nodes** are in the complete tree?
2. How many possible tic-tac-toe **positions** are there?
3. How can these two numbers be **different**?

9 cells in each position
Each cell can be X ,O, or empty
$\leq 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3$ positions
$= 3^9 =$ **19, 683 positions**

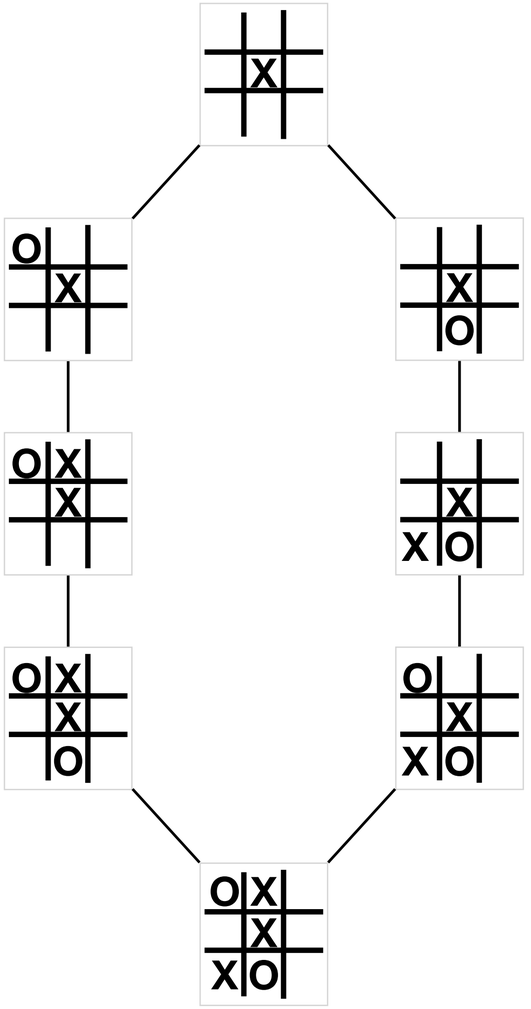
1 level 0 node +
9 level 1 nodes +
9×8 level 2 nodes +
$\leq 9 \times 8 \times 7$ level 3 nodes +
$\leq 9 \times 8 \times 7 \times 6$ level 4 nodes +
$\leq 9 \times 8 \times 7 \times 6 \times 5$ level 5 nodes +
$\leq 9 \times 8 \times 7 \times 6 \times 5 \times 4$ level 6 nodes +
$\leq 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3$ level 7 nodes +
$\leq 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2$ level 8 nodes +
$\leq 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1$ level 9 nodes
= **986, 410 nodes**

Tic-tac-toe Game Graph

- The game tree for tic-tac-toe isn't really a tree
- Many positions can be reached by **multiple paths**
  - These are called **transpositions**
- **Question:** **how many** paths from root to this position?
  - How many orders could O have moved in?
  - How many orders could X have move in?
  - $3! \times 4! = 144$ paths to the **same position**



- Once we've computed the value of a transposition, why compute it again?
- **Idea:** Cache the values of each position in a **transposition table**
  - Check the table before computing values

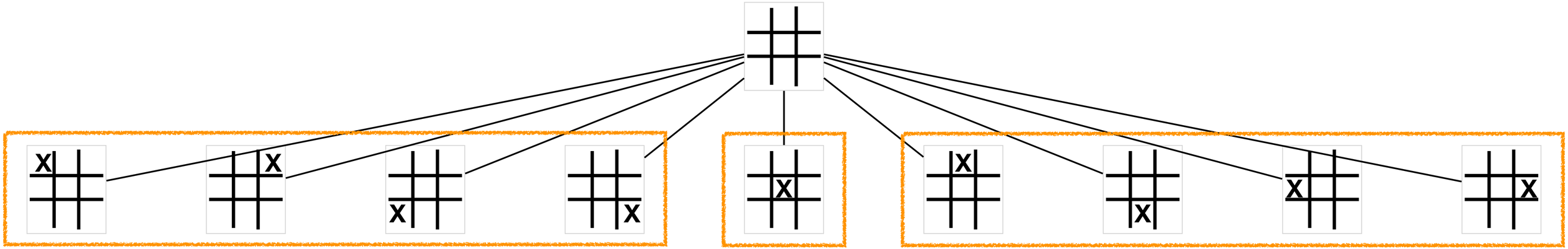


Isomorphisms

- Two positions are **isomorphic** to each other if one can be transformed into the other in a reversible way that preserves structure (e.g., all children are also isomorphic)
- E.g.:  can be transformed to  by rotating 90°:
  - The child  is isomorphic to the child  by the **same transformation**
  - The transformation is reversible by rotating 270°

Questions:

1. How many **children** does the empty position have?
2. How many **non-isomorphic** children does the empty position have?
3. How can we **exploit** isomorphisms?

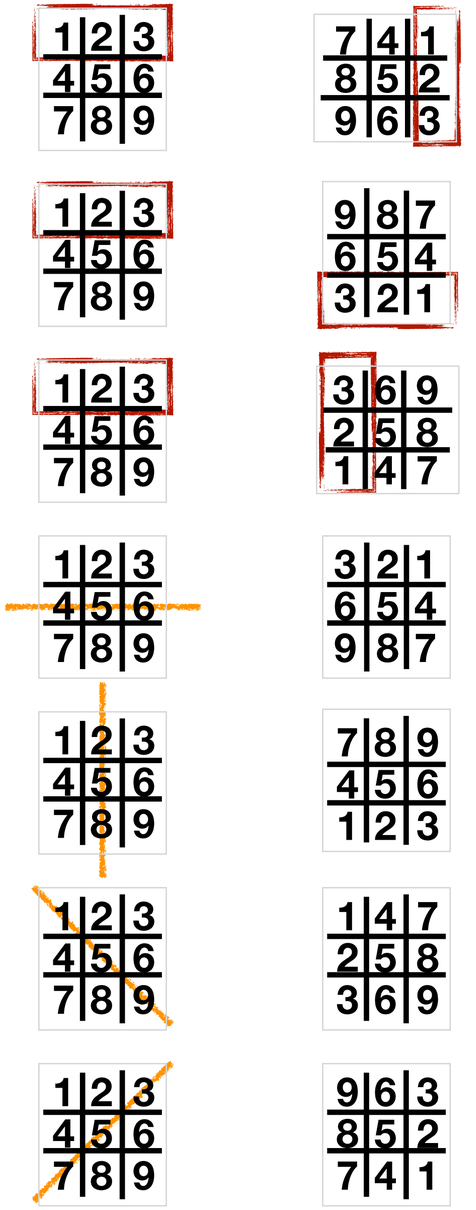


Tic-tac-toe Isomorphisms

There are **7 ways** to **isomorphically** transform a tic-tac-toe position:

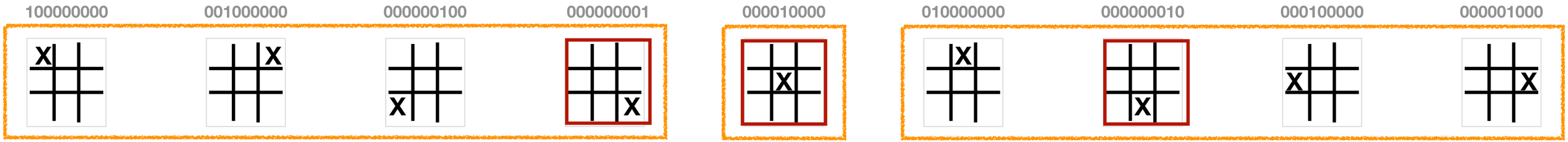
1. 90° **rotation**
2. 180° **rotation**
3. 270° **rotation**
4. Vertical **reflection**
5. Horizontal **reflection**
6. Major-diagonal **reflection**
7. Minor-diagonal **reflection**

Question: How would you **reverse** each of these transformations?



Canonical Form

- We only want to explore **non-isomorphic** children of each node
  - I.e., **one member** from each **isomorphic set**
- It's important that we always map an isomorphic set to the **same member** (**why?**)
  - This same member is the **canonical form** for a given isomorphic set
- **Question:** How can we choose a canonical form?
- For **tic-tac-toe**, to find the canonical form for a position:
  1. Compute **all isomorphic positions** (including **self**)
  2. Write each position as a **9-digit number** (each digit is 0 for empty, 1 for X, 2 for O)
  3. Choose the position with the **smallest number**
- **Question:** Why does each isomorphic set have only **4** (or 1!) **elements** when there are **7 transformations**?



Exploiting Isomorphisms & Transpositions

```
### isomorphism permutations
Isos = ( (0,1,2,3,4,5,6,7,8),
         (0,3,6,1,4,7,2,5,8),
         (2,1,0,5,4,3,8,7,6),
         (2,5,8,1,4,7,0,3,6),
         (8,7,6,5,4,3,2,1,0),
         (8,5,2,7,4,1,6,3,0),
         (6,7,8,3,4,5,0,1,2),
         (6,3,0,7,4,1,8,5,2)  )
```

```
# L is list of possible next moves
def non_iso_moves(self, L, ptm):
  assert(len(L)>0)
  H, X = [], []  # hash values, moves
  for j in range(len(L)):
    p = L[j]
    self.brd[p] = ptm
    h = min_iso(self.brd)
    if h not in H:
      H.append(h)
      X.append(p)
    self.brd[p] = Cell.e
  return X
```

```
def negamax(use_tt, use_iso, MMX, calls, d, psn, ptm): # 1/0/-1 win/draw/loss
  calls += 1
  psn_int = board_to_int(psn.brd)
  if use_tt and (psn_int in MMX[ptm - 1]):
    return MMX[ptm - 1][psn_int], calls
  if psn.has_win(opponent(ptm)):
    return -1, calls  # previous move created win
  G = psn.legal_moves()
  if len(G) == 0:
    return 0, calls  # board full, no winner
  L = psn.non_iso_moves(G, ptm) if use_iso else G
  so_far = -1  # best score so far
  for cell in L:
    psn.brd[cell] = ptm
    nmx, c = negamax(use_tt, use_iso, MMX, 0, d+1, psn, opponent(ptm))
    so_far, calls = max(so_far, -nmx), calls + c
    psn.brd[cell] = Cell.e  # reset brd to original
    # if so_far == 1: break   # improvement: return once win found
  if use_tt: MMX[ptm - 1][psn_int] = so_far
  if d == 0 and use_tt:
    xsize, osize = len(MMX[0]), len(MMX[1])
    print('\n  TT size', xsize, osize, xsize+osize)
  return so_far, calls
```

```
def min_iso(L): # L is a position; return isomorphic position with smallest board_to_int
  return min([board_to_int([L[Isos[j][k]] for k in range(Cell.n)]) for j in range(8)])
```

Demo:

% cd ttt
% python3 tt.py

Summary

- Incorporating **alpha-beta pruning** into negamax search is straightforward
- **Transpositions:**
  - Many positions can be reached by **multiple paths**
  - When we encounter one, we can simply **return the previously-computed value**
- **Isomorphisms:**
  - Many positions are **structurally equivalent** to each other (**isomorphic**)
  - E.g., playing in any corner is in some sense the same move
  - So we only need to evaluate **non-isomorphic children**
    - since any two isomorphic children will have exactly the same value
  - Convert each position to an isomorphic **canonical form**
  - Only explore the canonical forms
- Tic-tac-toe: transpositions and isomorphisms reduce exploration from **549,946** nodes to **8,307**!