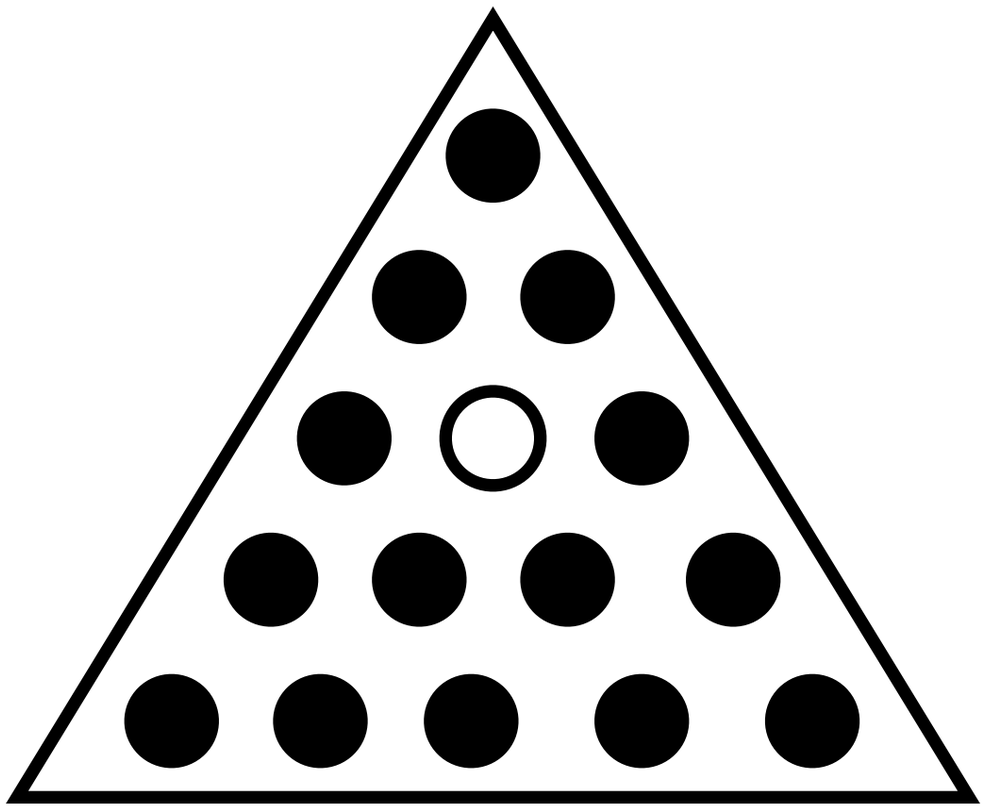
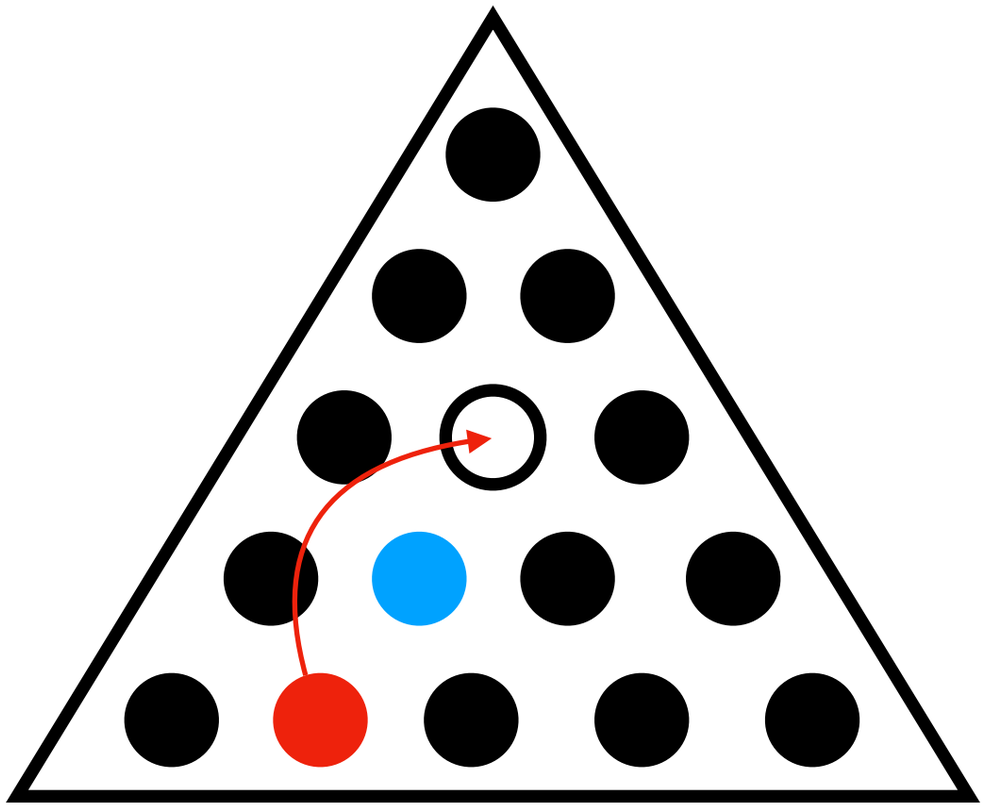
Learning to Win at Peg Solitaire Without Human Knowledge

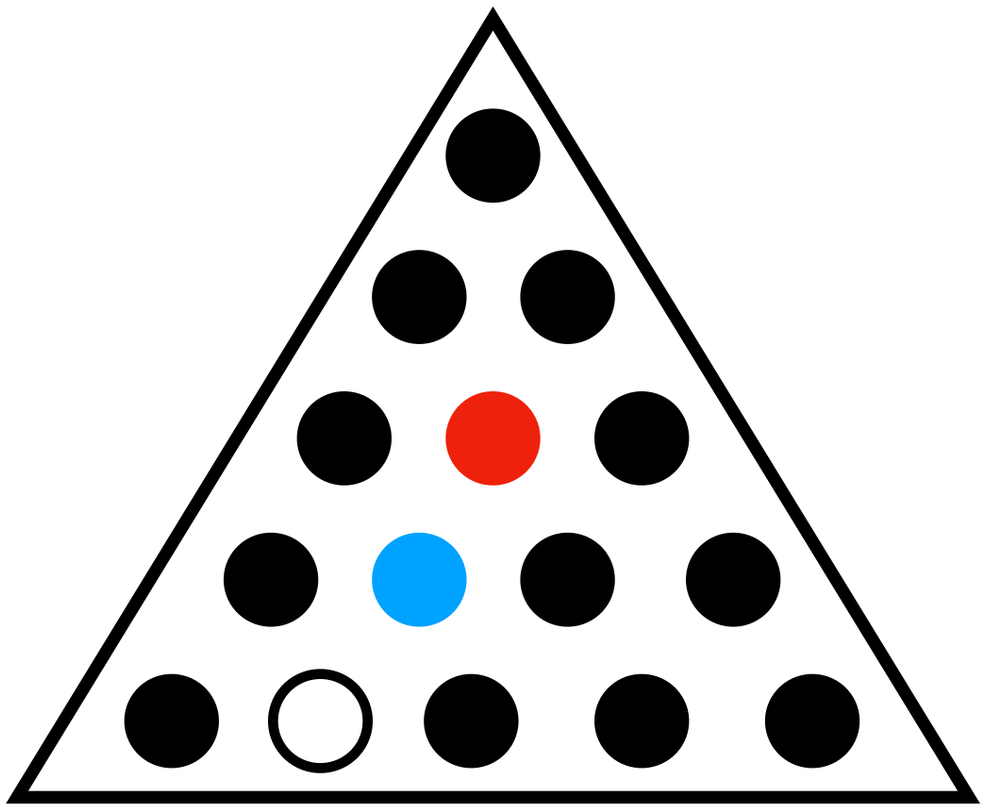
Eric Allen

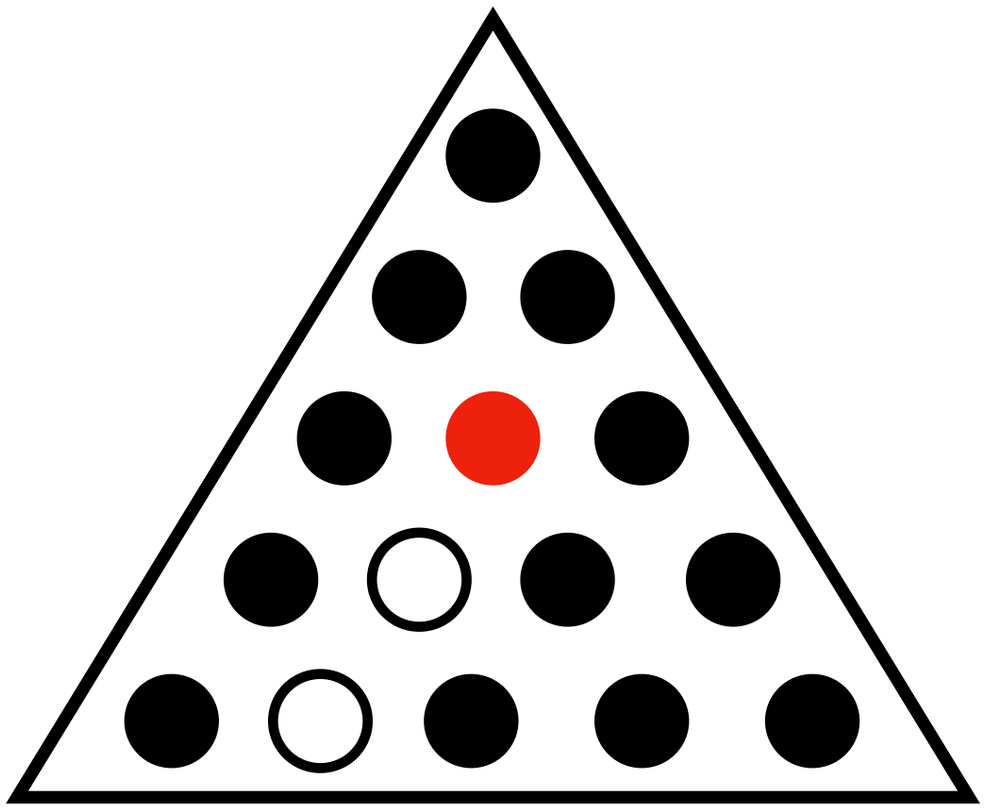
Peg Solitaire

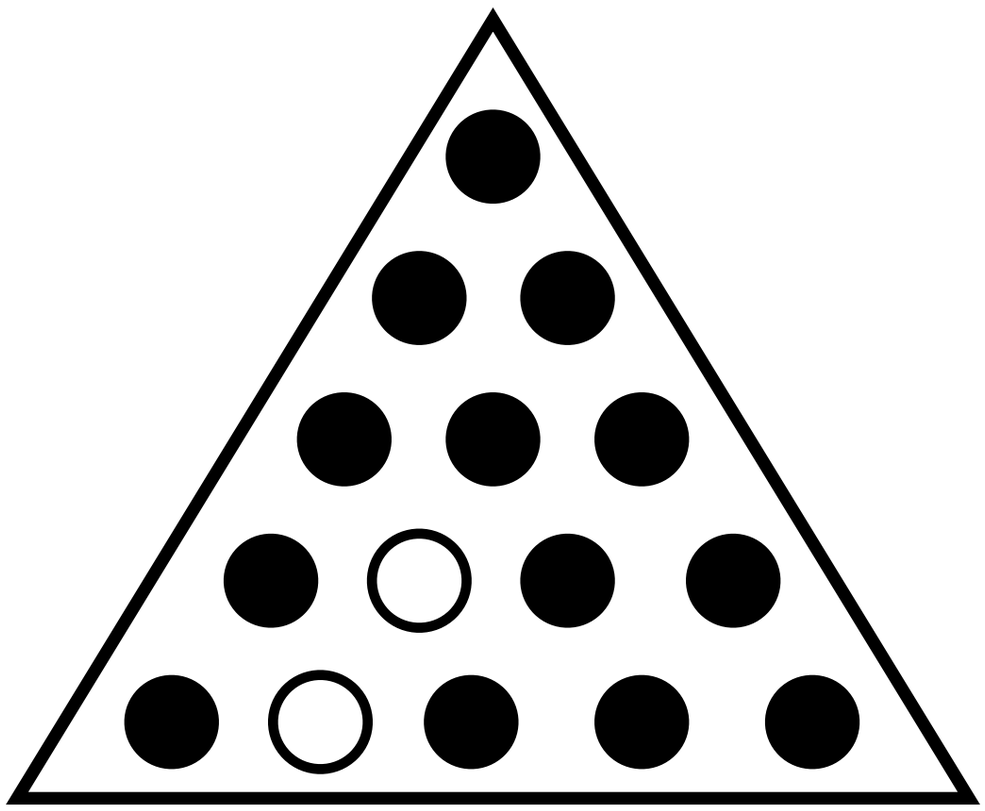
- Pegs are placed in holes
- Pegs can jump over other pegs
- If a peg is jumped over, it is removed it from the board
- The goal is to end with a board with as few pegs as possible



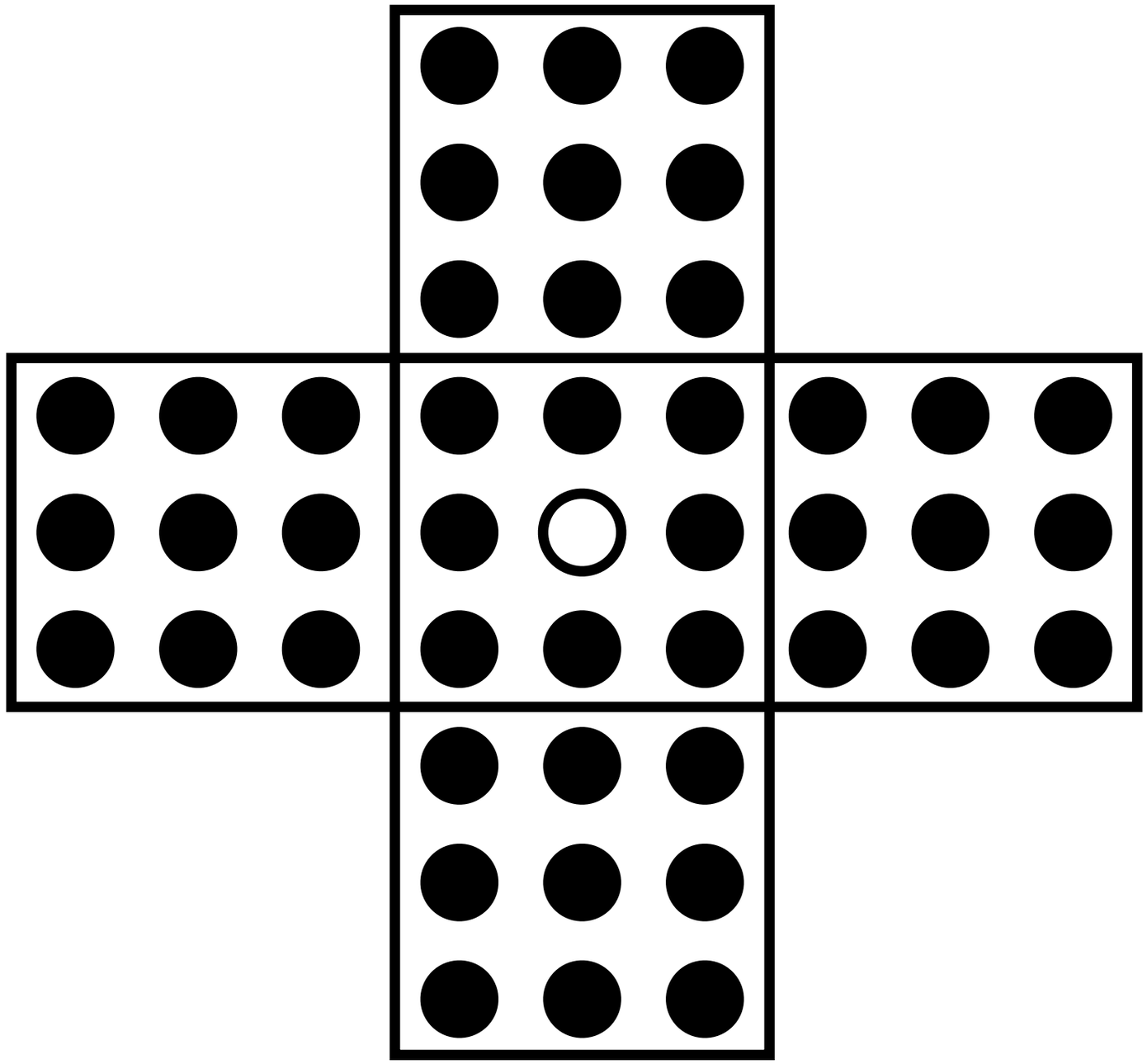






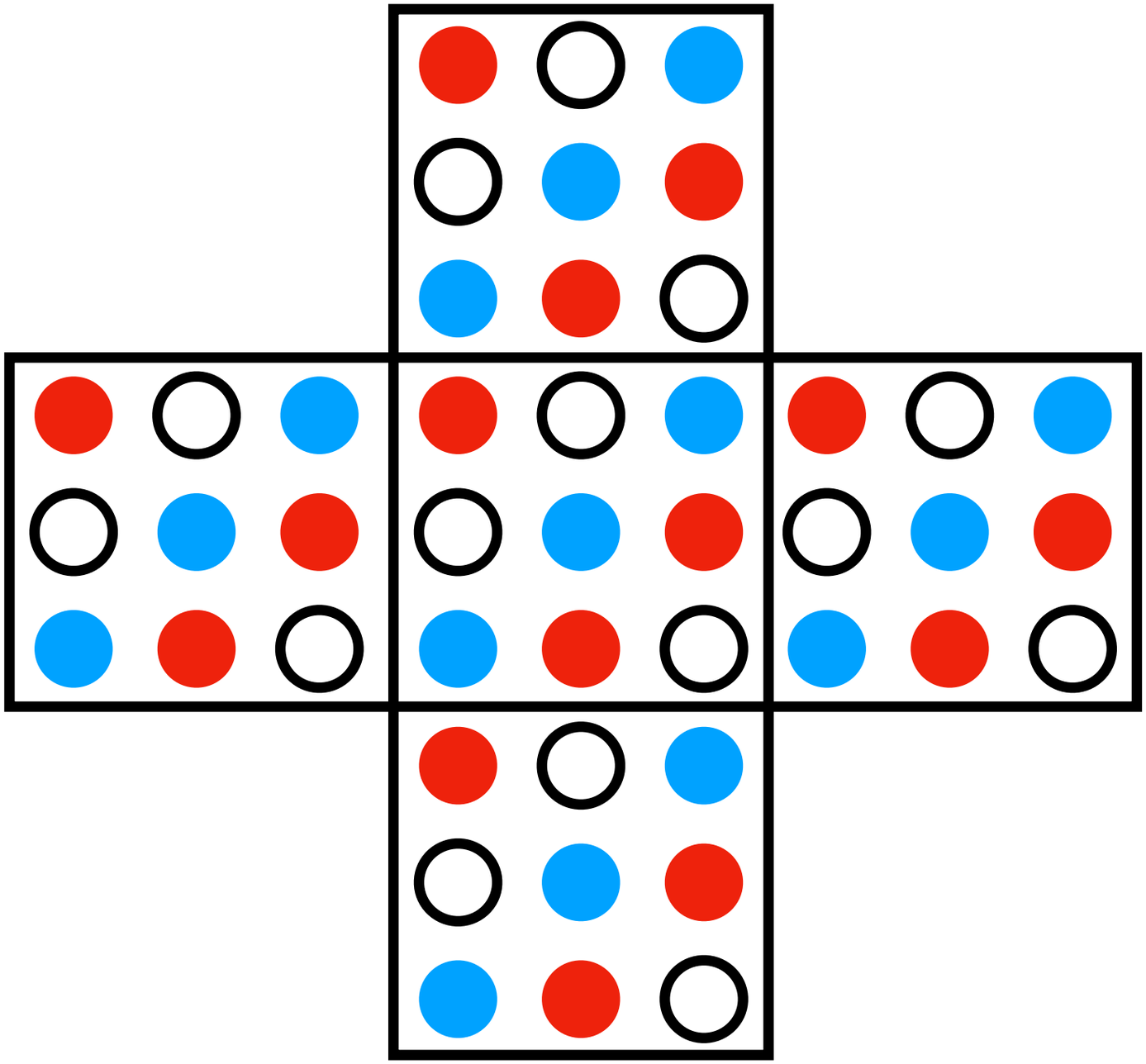


Another Common Board

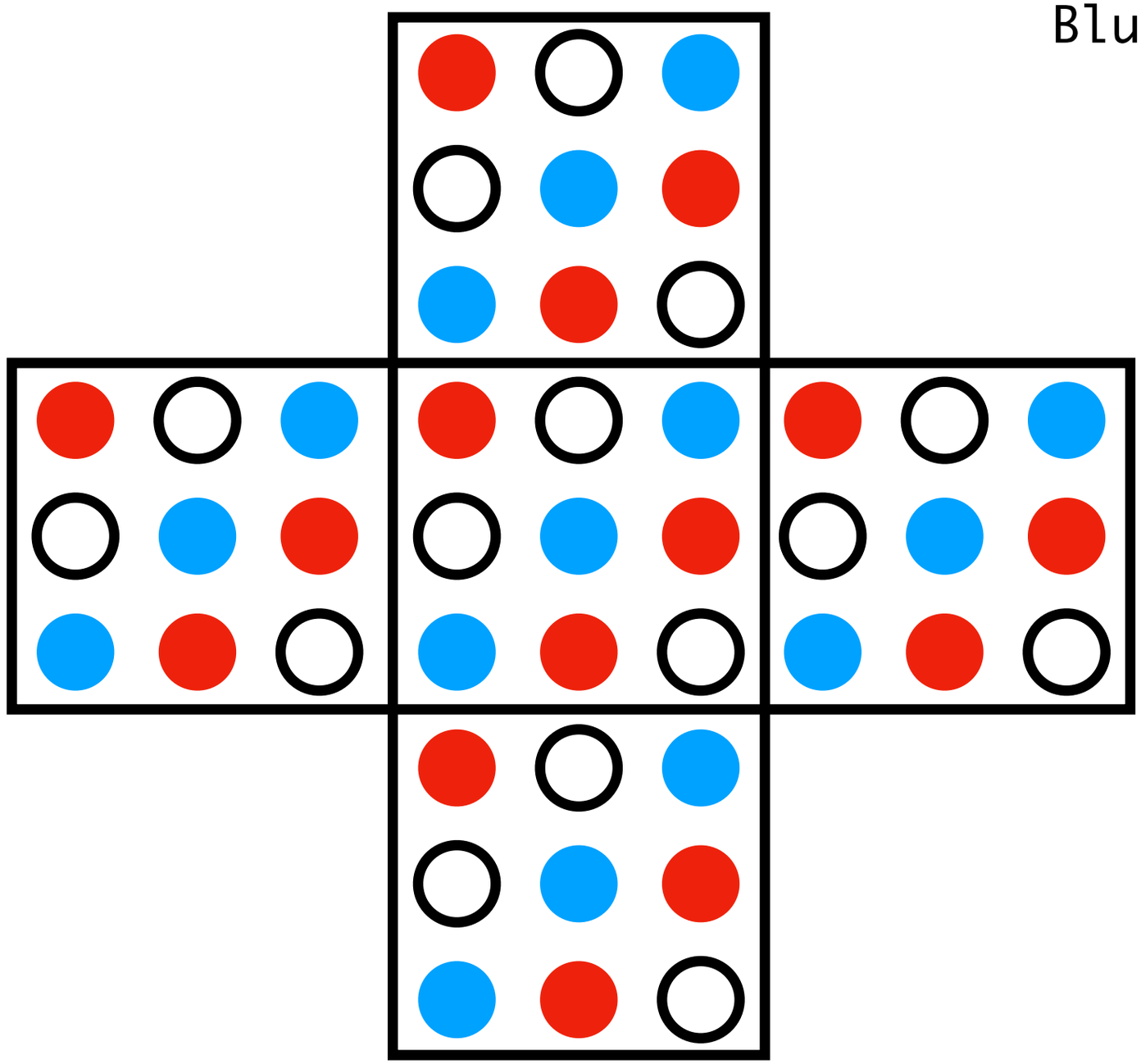


Solution Scoring

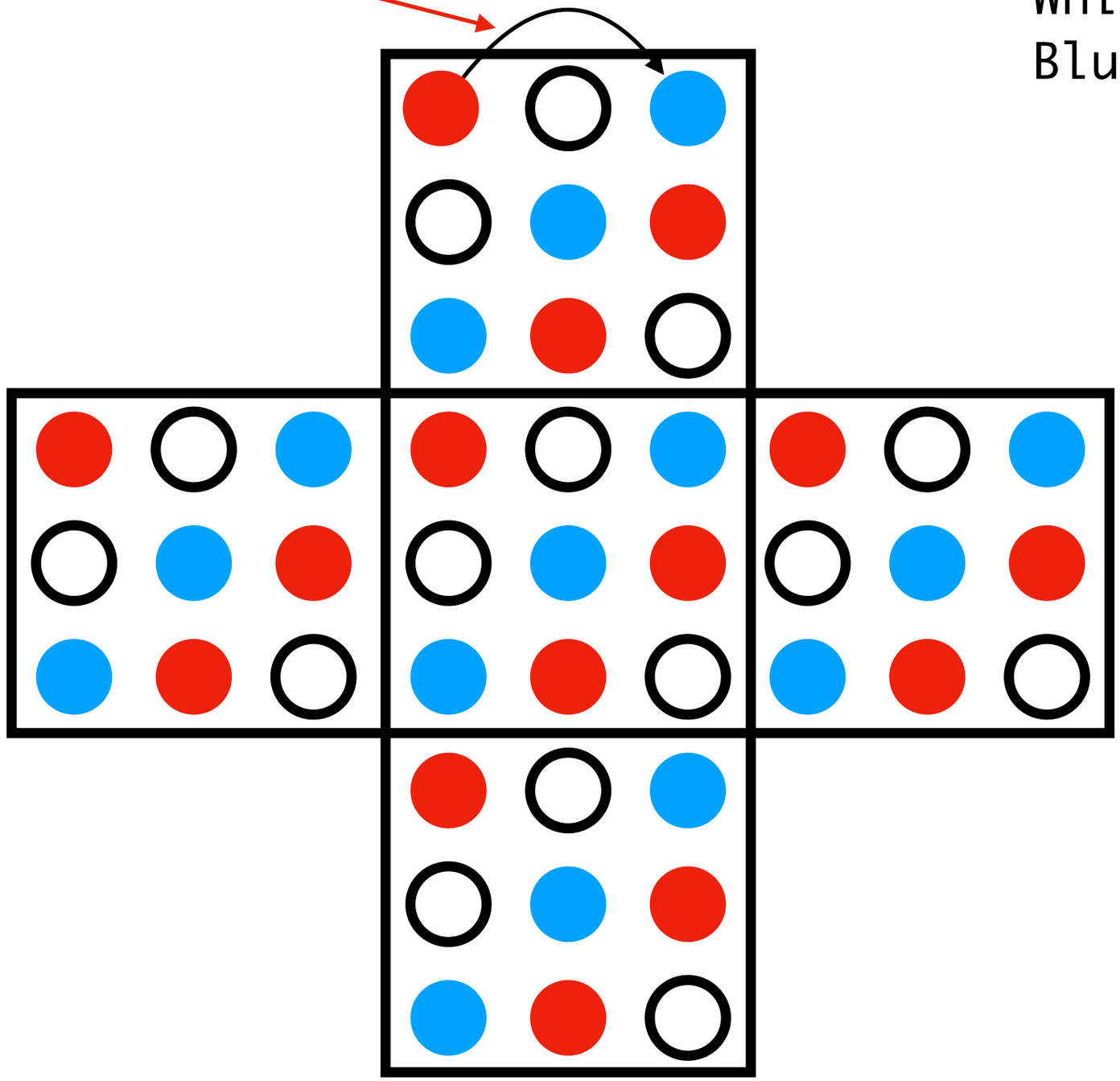
5 Pegs Left	100 IQ
4 Pegs Left	200 IQ
3 Pegs Left	300 IQ
2 Pegs Left	400 IQ
1 Peg Left	500 IQ
1 Peg in Center	1000 IQ



Red pegs: 15
White pegs: 15
Blue pegs: 14

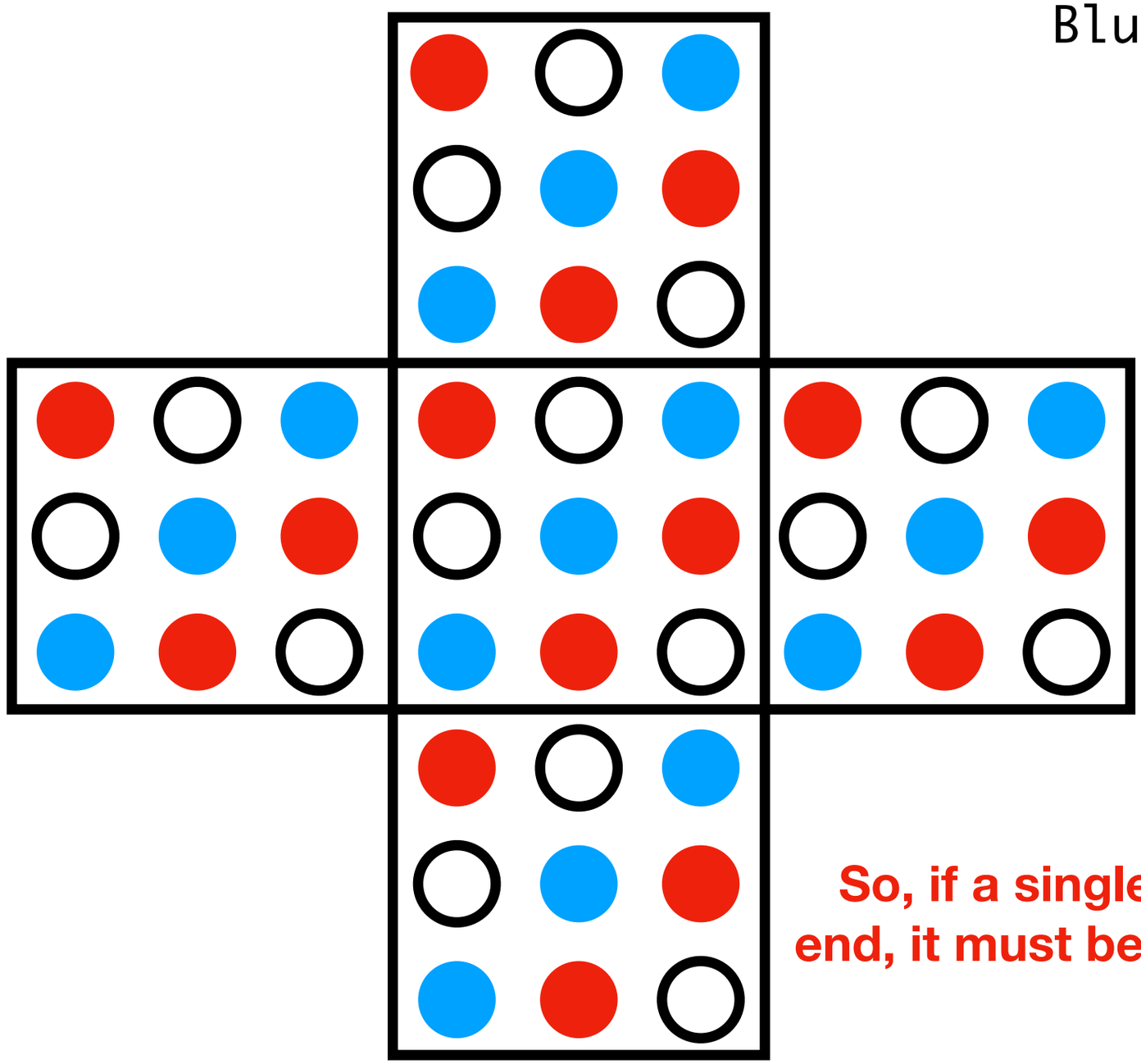


Every jump switches the parity of the number of pegs on each color.

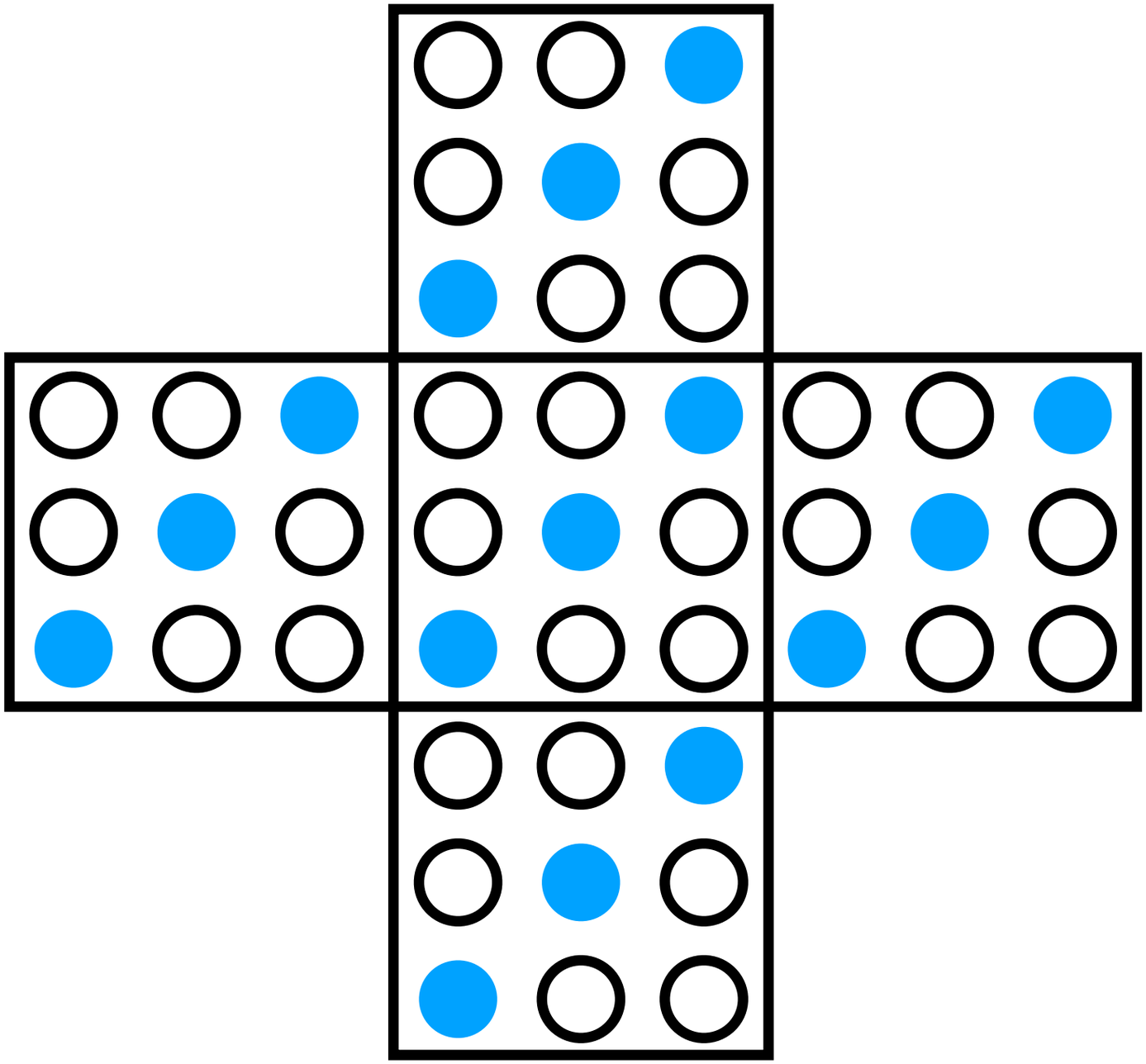


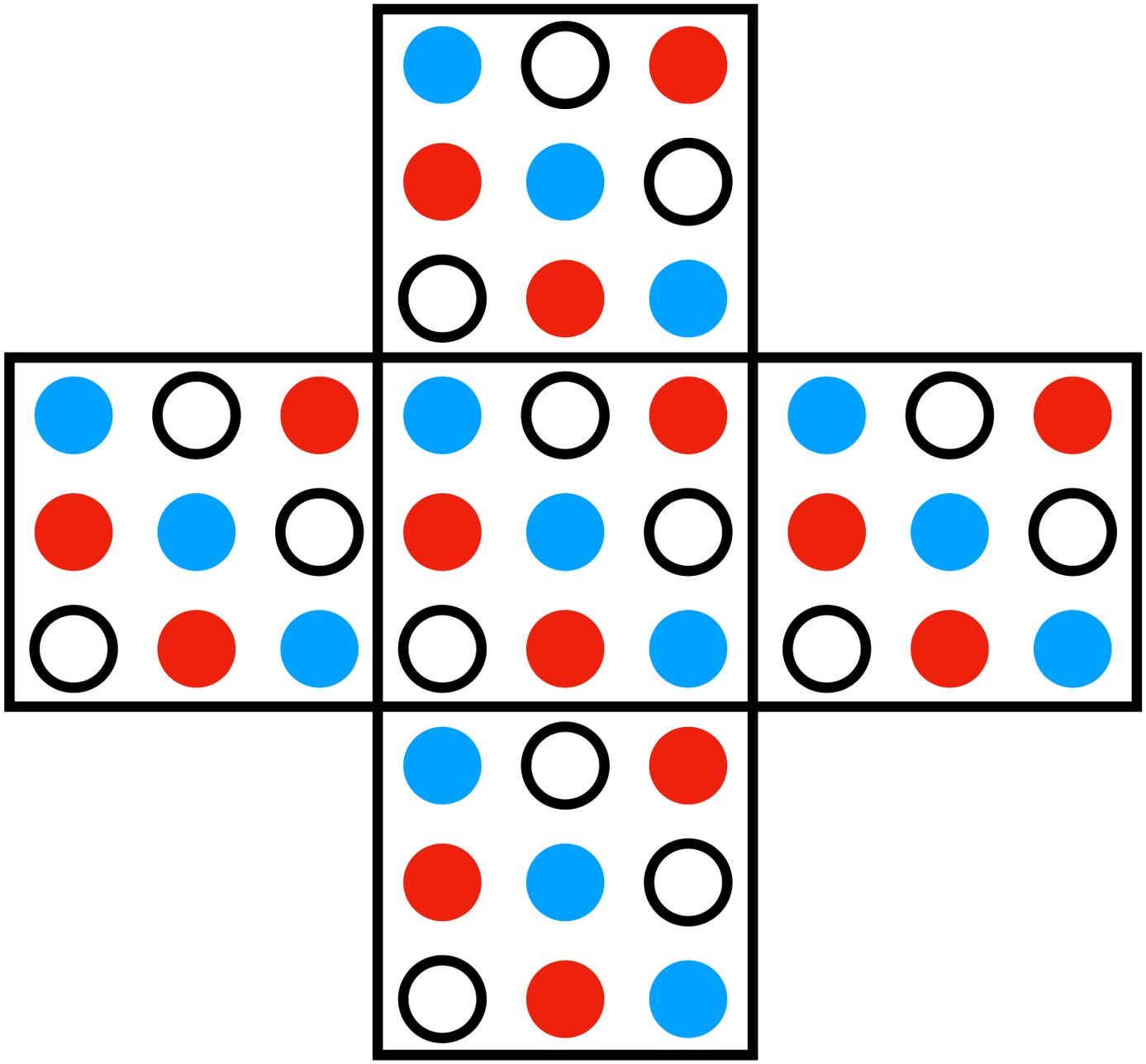
```
Red pegs:    15
White pegs:  15
Blue pegs:   14
```

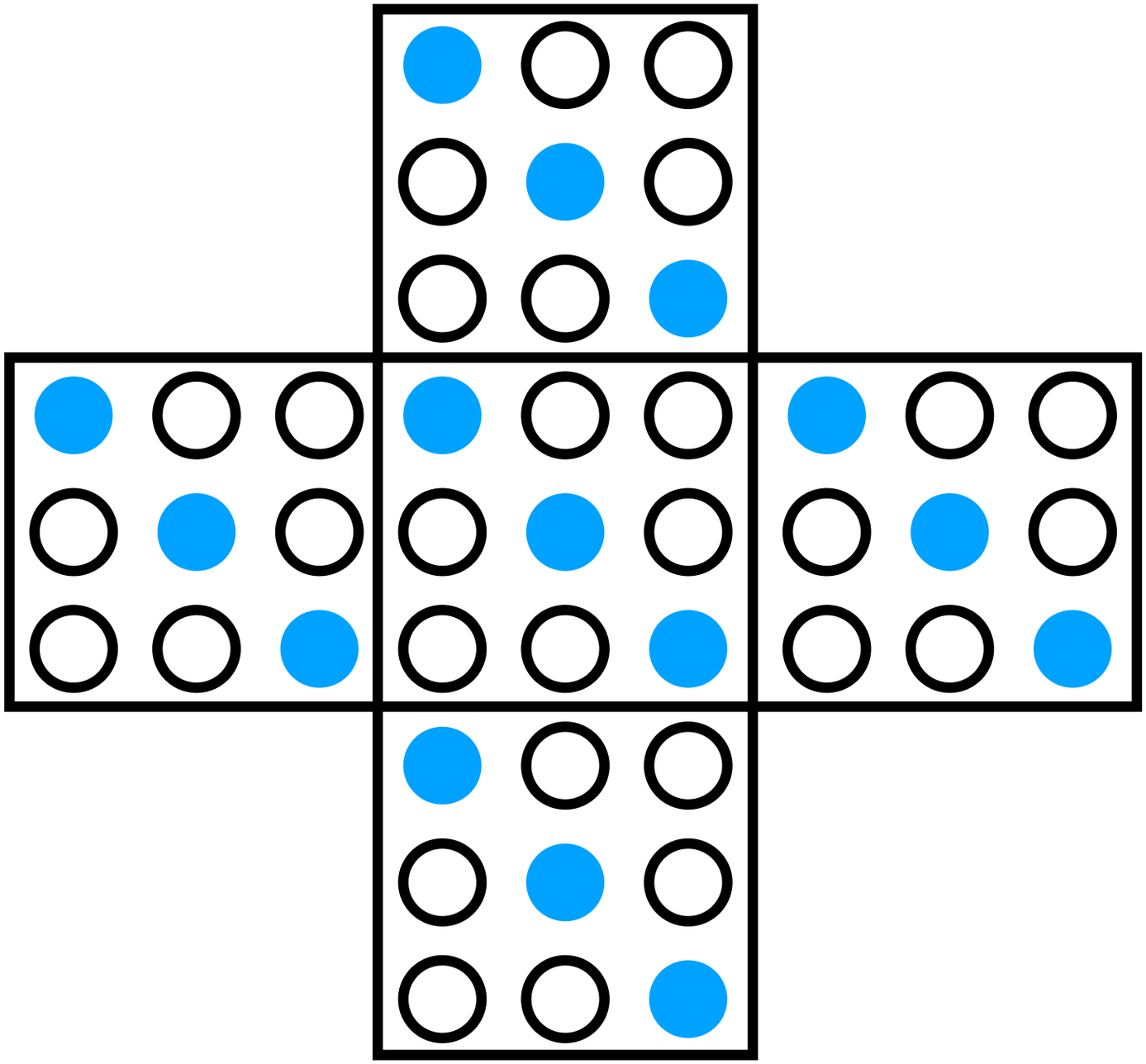
Red pegs: 15
White pegs: 15
Blue pegs: 14

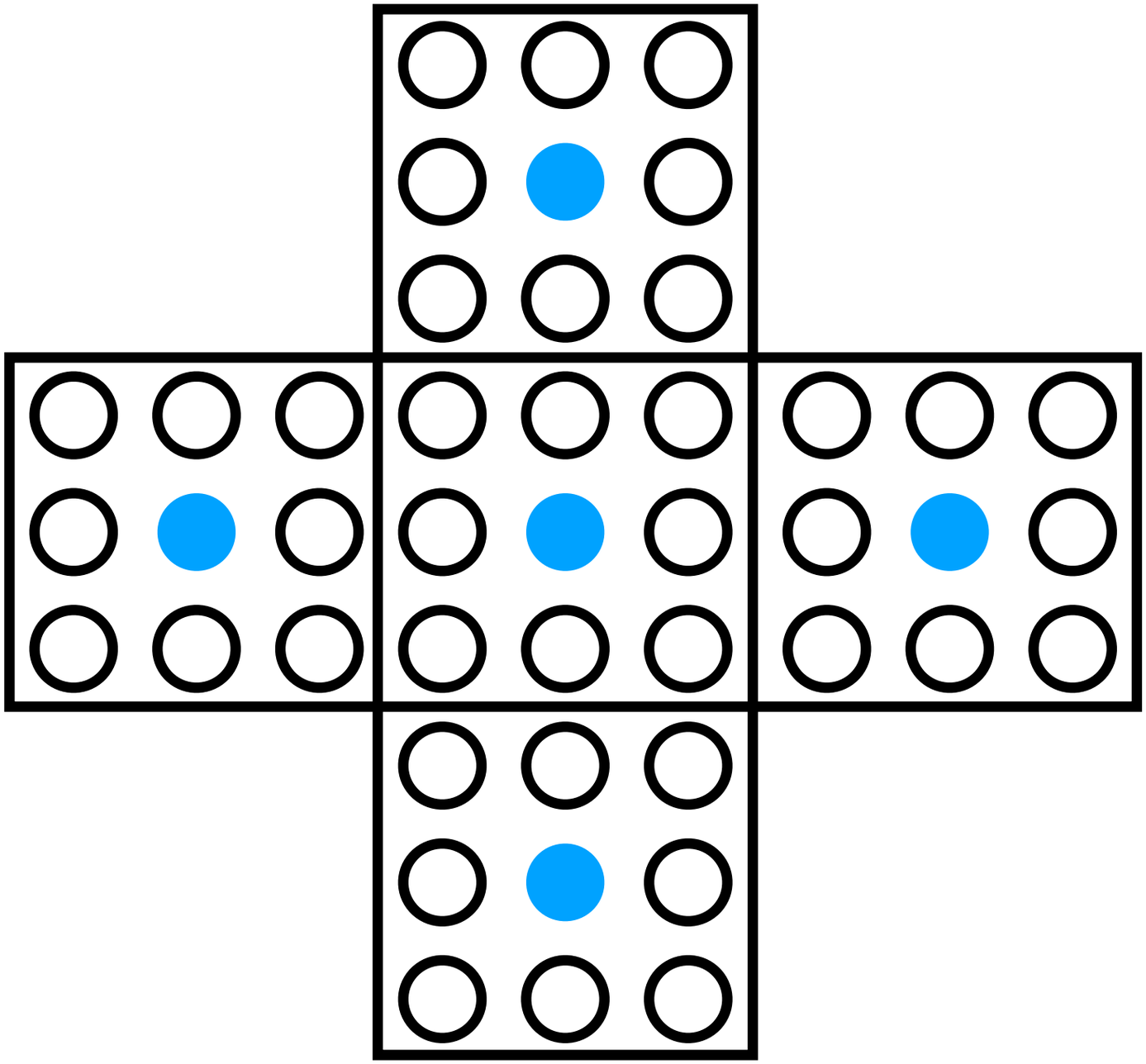


So, if a single peg is left at the end, it must be on a blue position.

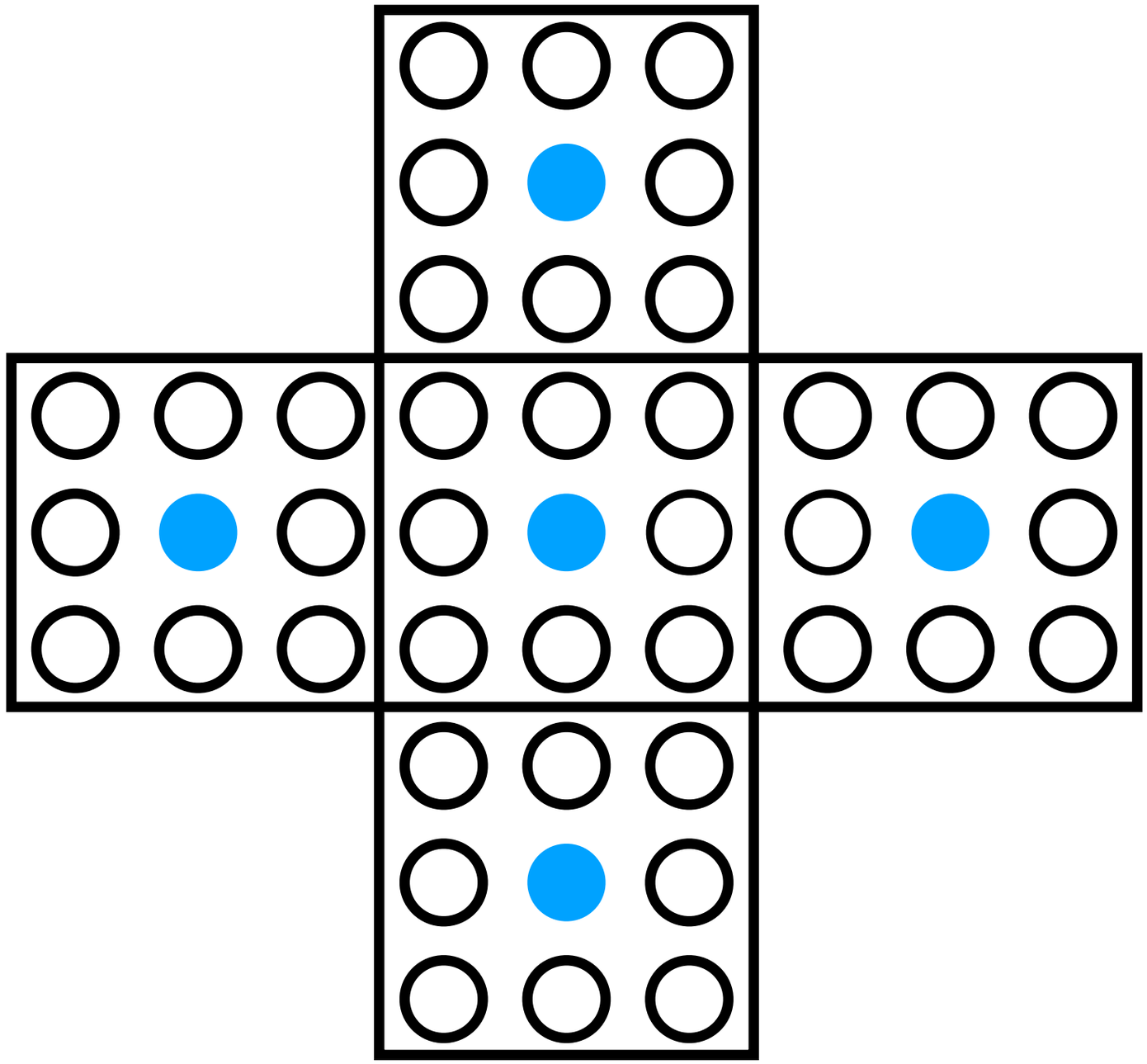




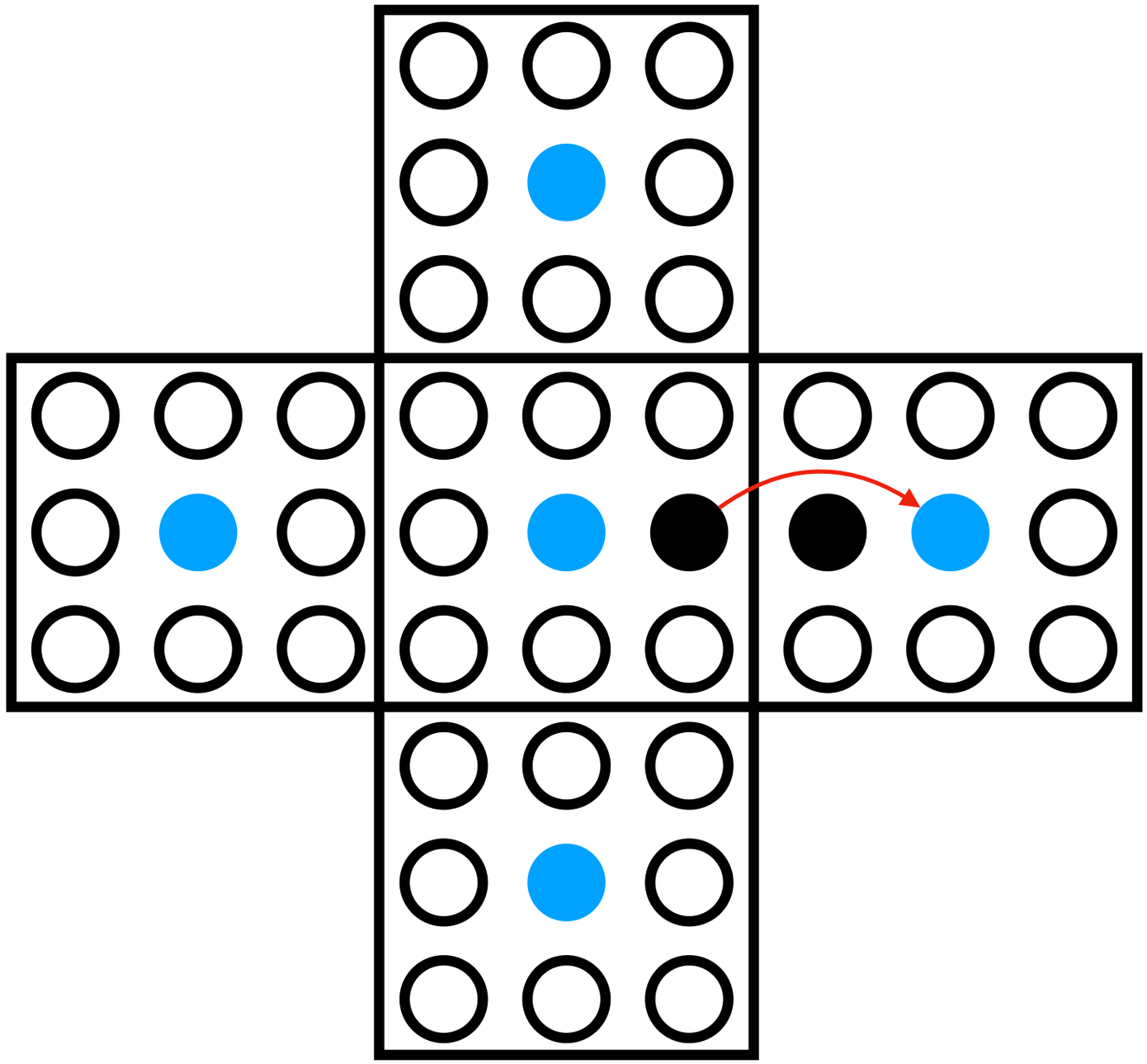




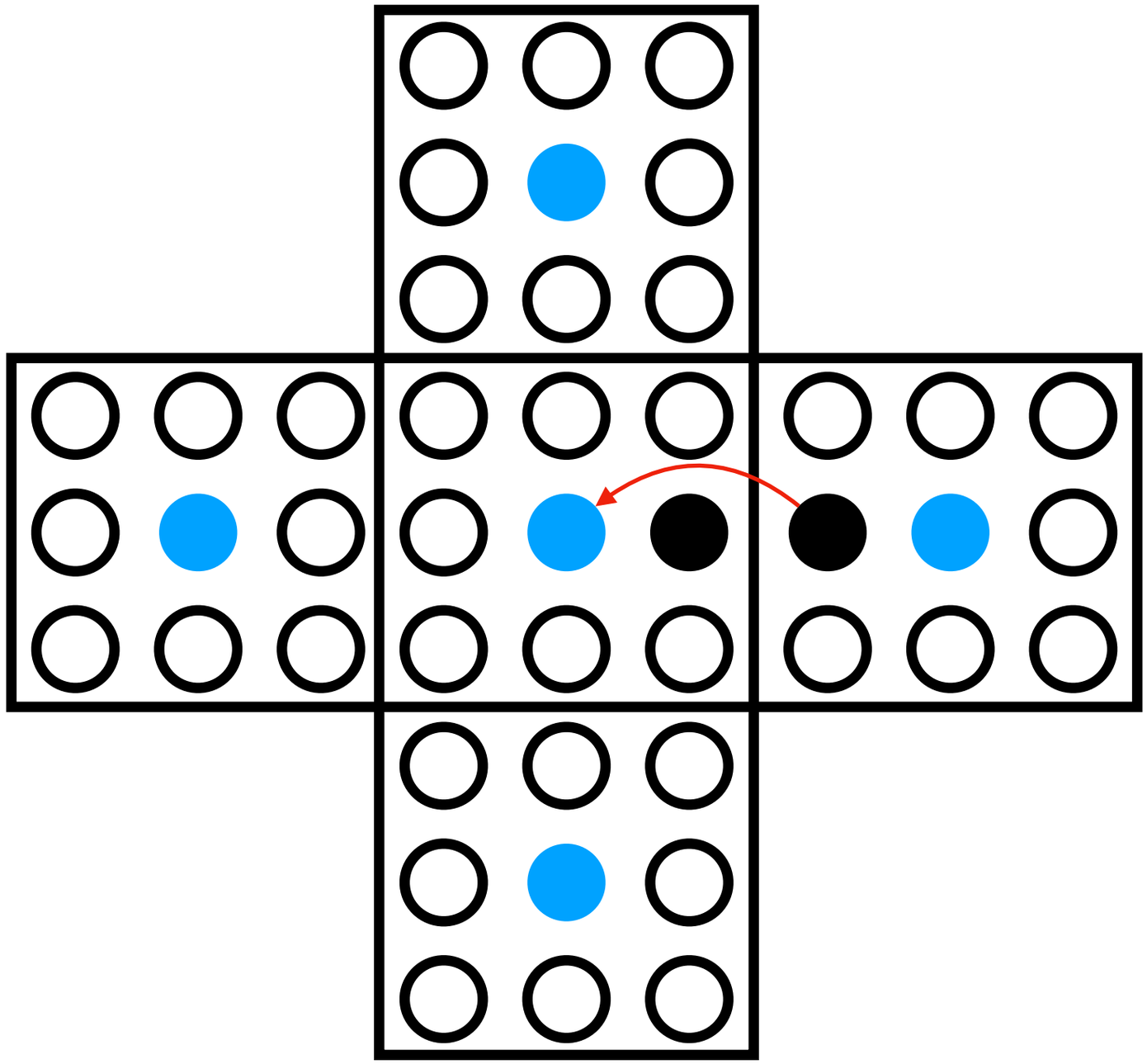
Thus, if we are left with a single peg, in must be in one of these blue holes.



But consider what the last move must be in order to end with a peg in one of the blue holes not in the center…



We could have ended with one peg in the center square just as easily.

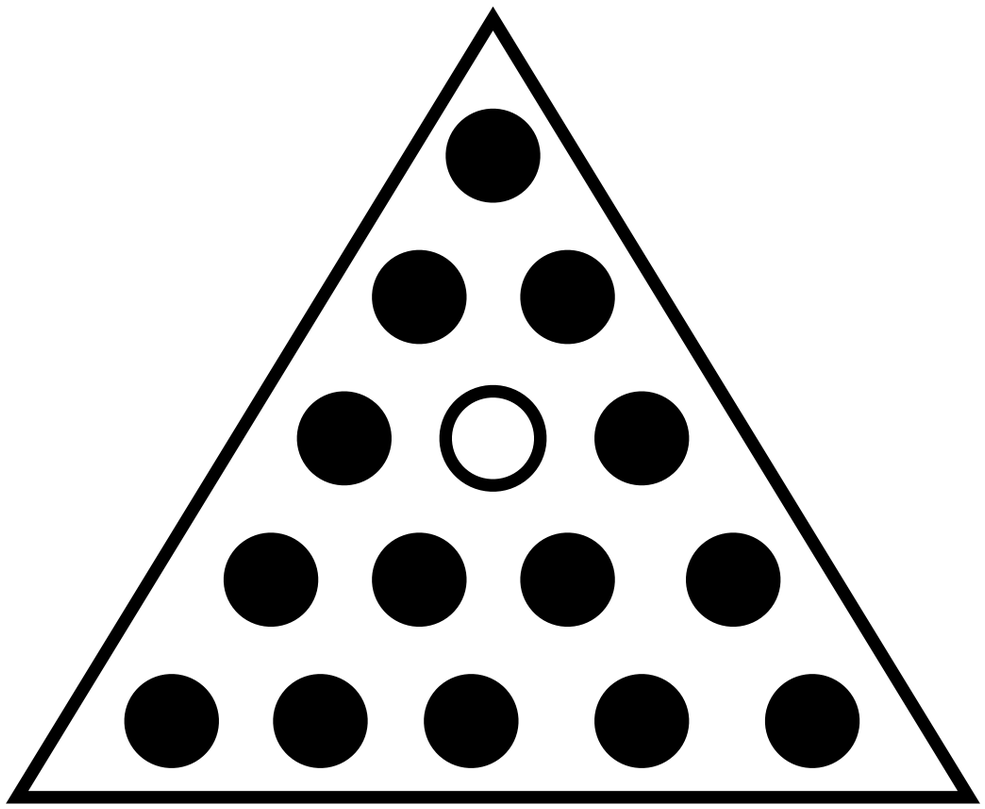


Therefore, at least one of these scores is incorrect.

5 Pegs Left	100 IQ
4 Pegs Left	200 IQ
3 Pegs Left	300 IQ
2 Pegs Left	400 IQ
1 Peg Left	500 IQ
1 Peg in Center	1000 IQ

Automating Peg Solitaire

We would like to automate finding solutions to various sizes of the triangular board



Framing the Problem

- We would like a function that takes:
  - The size of a board
  - The position of an initial hole
- And returns:
  - A sequence of moves that reduces the board to a single peg
- We allow the peg to be in any position
  - A 500 IQ for our AI is acceptable

Depth-First Search

```
case class DepthFirstSolver(size: Int) extends Solver {
  import Moves._

  def solve(hole: (Int,Int)): Option[List[Move]] = {
    def inner(b: Board, moves: List[Move], bms: List[(Board, List[Move])]): Option[List[Move]] = {
      if (b.isSolution) {
        Some(moves.reverse)
      }
      else {
        val nextBoardsAndMoves = {
          for {
            i <- (0 until size);
            j <- (0 to i);
            Move(from,to) <- Moves.moves(b, (i,j))
          }
          yield (b.move(from,to), Move(from,to) :: moves)
        }.toList

        nextBoardsAndMoves match {
          case Nil => bms match {
            case Nil => None
            case (board, moves) :: more => inner(board, moves, more)
          }
          case (board, moves) :: more => inner(board, moves, more ++ bms)
        }
      }
    }
    inner(BoardFactory(size, Set(hole)), Nil, Nil)
  }
}
```

Depth-First Search

- Scales well on this laptop up to boards of size six

- Able to solve a size nine board in roughly 3.2 hours

AlphaGo Zero

AlphaGo

- First program to achieve superhuman performance at the game of Go

AlphaGo Fan

- Played against European champion Fan Hui in October 2015
- Distributed over many machines and 176 GPUs

AlphaGo Lee

- Defeated international champion Lee Sodol 4-1 in October 2016

- Value network trained from outcomes of fast games of self-play

- Distributed over many machines, using 48 TPUs

AlphaGo Master

- An intermediate (unpublished) upgrade
- Defeated top human players 60-0 in January 2016
- Same neural network architecture, reinforcement learning algorithm, and MCTS algorithm as AlphaGo Zero
- Uses the same handcrafted features and rollouts as AlphaGo Lee
- Training initialized by supervised learning from human data

AlphaGo Zero

- Learned entirely from self-play reinforcement learning, starting from random weights for the neurons
- No human supervision
- Only the raw board (history) was used as input features
- Used a single machine on the Google Cloud, with 4 TPUs (the smallest and simplest approach that worked)

Domain Knowledge Incorporated in AlphaGo Zero

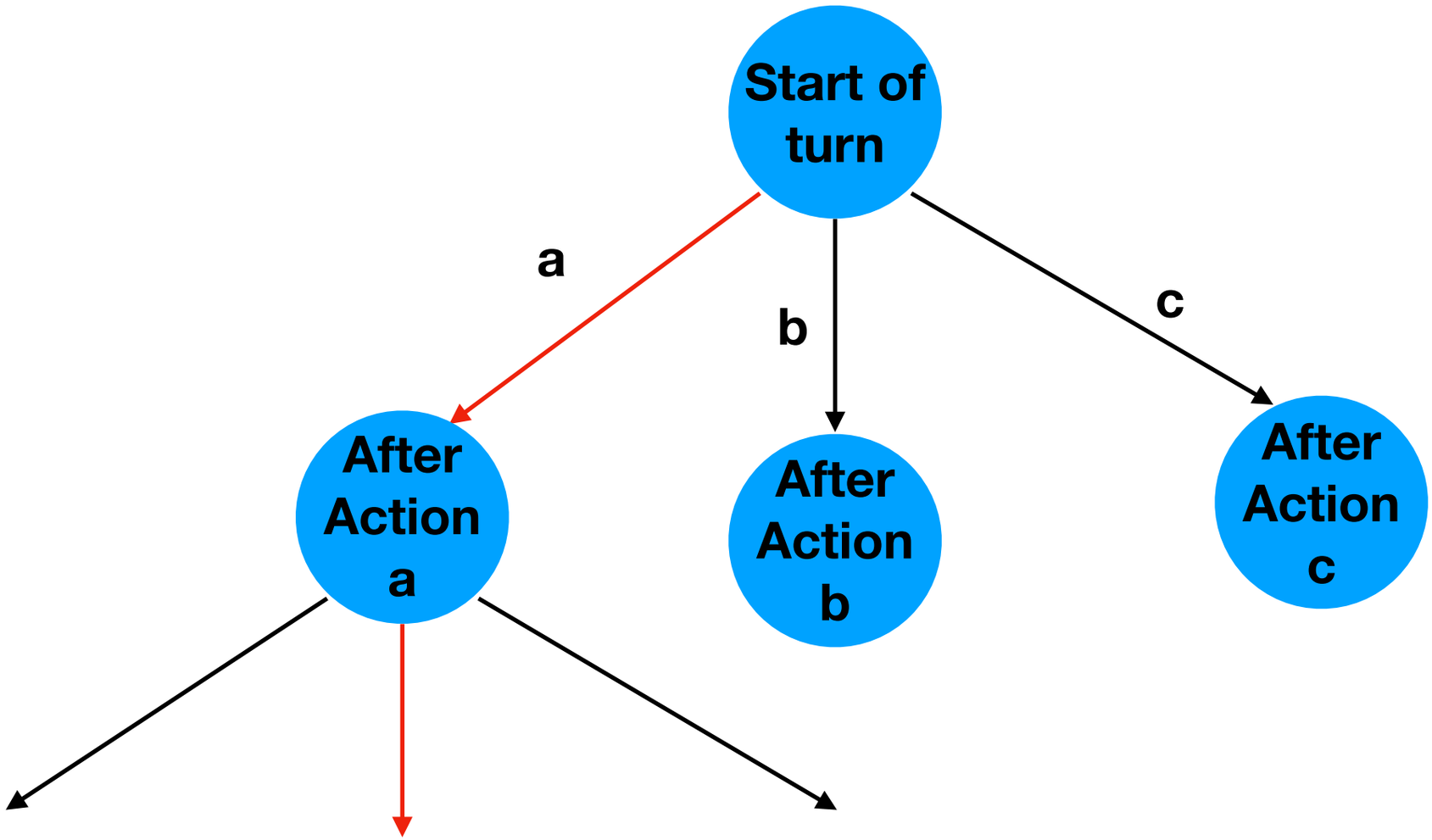
- Perfect knowledge of the rules of Go
- Uses “Tromp-Taylor” scoring during MCTS
    - Avoids some ambiguities in Chinese, Japanese, and Korean rules
- Board evaluation is invariant with respect to reflections and rotations, and virtually symmetric with respect to color
    - These properties are used to augment the dataset

How AlphaGo Zero Chooses Moves to Play During a Game

AlphaGo Zero Uses Monte Carlo Tree Search

- Consider a game tree for Go:
  - Every game state / board state is a node in the tree
  - The root of the tree is the starting game state
  - There is an edge from vertex *s* to vertex *t* iff there is a legal action that transforms the game state from *s* to *t*

AlphaGo Zero Uses Monte Carlo Tree Search



AlphaGo Zero Uses Monte Carlo Tree Search

- Each search begins at a root node s_0 (the current actual game state)

- The search selects a single action at each step for L time steps, arriving at a state s_L

- Many games are simulated in parallel in this fashion from root node s_0

- Statistics on the potential actions are collected by these simulations

- The “best” potential action is actually played

AlphaGo Zero Uses Monte Carlo Tree Search

- Each action in the search tree is annotated with:
  - *N(s,a)* the number of times that *a* has been visited
  - *W(s,a)* the sum of values that have resulted from performing action *a*
  - *Q(s,a)* the mean action value
  - *P(s,a)* the prior probability of selecting edge *a*

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Exploitation **Exploration**

Search Algorithm

- Each potential action from a game state is *scored* using a variant of the *Predictor + Upper Confidence bound applied to Trees (PUCT) algorithm*:

$$Q(s,a) + U(s,a)$$

$$U(s,a) = cP(s,a)\sqrt{\frac{\sum_b N(s,b)}{1+N(s,a)}}$$

Informed by a neural network (the *value network*)

Informed by a neural network (the *policy network)*

Search Algorithm

- At each step in a simulation:
    - The next action is selected in proportion to the scores of the potential actions

Value and Policy Networks

- The value network estimates the value (desirability) of an action
- The policy network estimates the probability of an action being taken
- We will see that AlphaGo Zero uses two heads on the same network for value and policy
- Thus, a single network is trained to output both quantities

Board Evaluation

- When reaching the end of a path:
  - The value v of board s_L is computed using our value network applied to a random reflection or rotation of s_L
  - Each new edge a from our end state s_L is initialized as:

$$N(s_L, a) = 0, W(s_L, a) = 0, Q(s_L, a) = 0, P(s_L, a) = p_a$$

- The value p_a is computed using our policy network
- The network(s) are shared resources:
  - They evaluate actions for eight boards at a time
  - All threads waiting on those board evaluations are locked until they complete

Backup

- The statistics of each board s_t in the path from s_0 to s_L is now updated:

$$
\begin{aligned}
N(s_t, a_t) &\quad += &&\quad 1 \\
W(s_t, a_t) &\quad += &&\quad v \\
Q(s_t, a_t) &\quad = &&\quad \frac{W(s_t, a_t)}{N(s_t, a_t)}
\end{aligned}
$$

Search Algorithm

- The tree nodes along the selected path are kept
- All other tree nodes are discarded

- If the root value and best child value are below a threshold, AlphaGo Zero resigns

Rollouts

- When game states are encountered for the first time, how do we decide which action to take?

- Rollouts: Select a move based on some default policy (e.g., random play, or something smarter)

- AlphaGo Zero does not use rollouts; it simply evaluates the game state based on the current value and policy networks

Action Selection

- At the end of the search, an action is selected for actual play

- Action is chosen probabilistically, in proportion to its "exponentiated visit count":

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

Action Selection

- At the end of the search, an action is selected for actual play

- Action is chosen probabilistically, in proportional to its "exponentiated visit count":

$$\pi(a|s) = \frac{N(s,a)^{1/\tau}}{\sum_b N(s,b)^{1/\tau}}$$

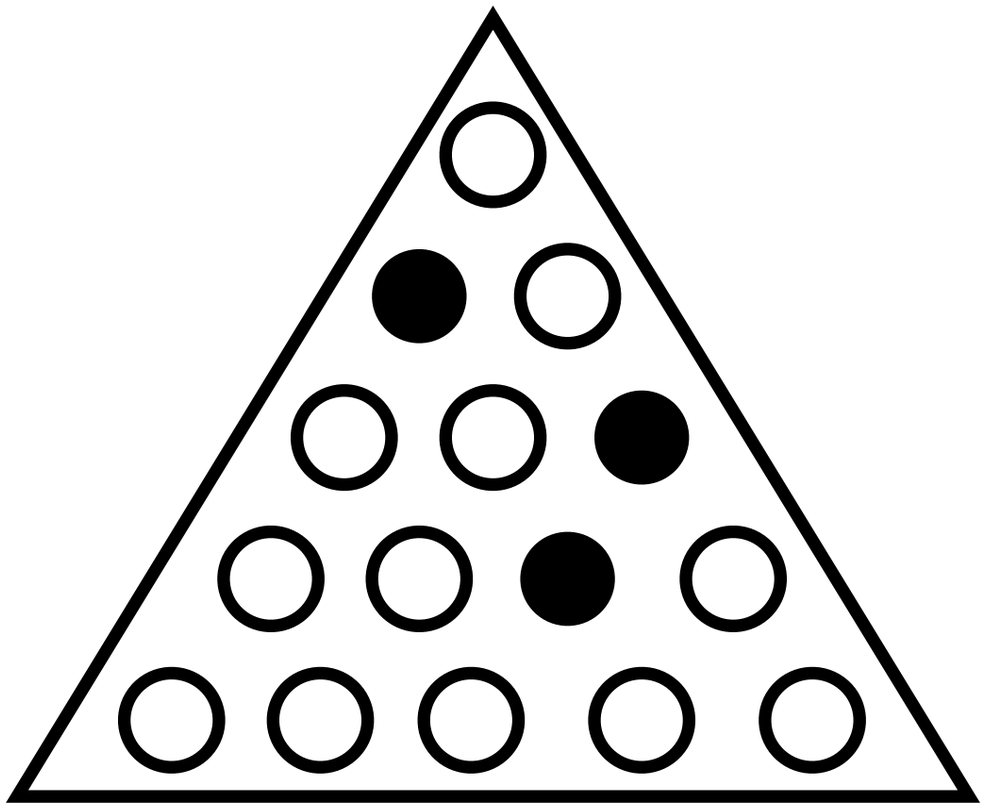
Temperature.

Higher values result in more exploration.

Applying Monte Carlo Tree Search to Peg Solitaire

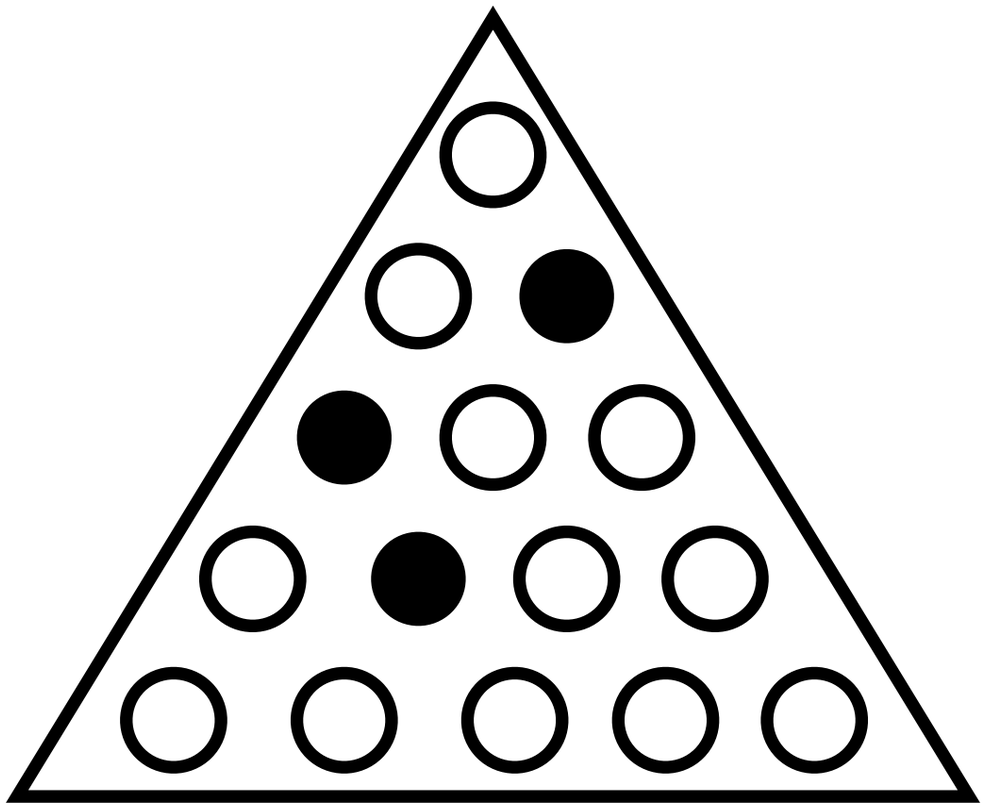
Applying Monte Carlo Tree Search to Peg Solitaire

- Note that we can exploit spatial symmetries, just as with Go:



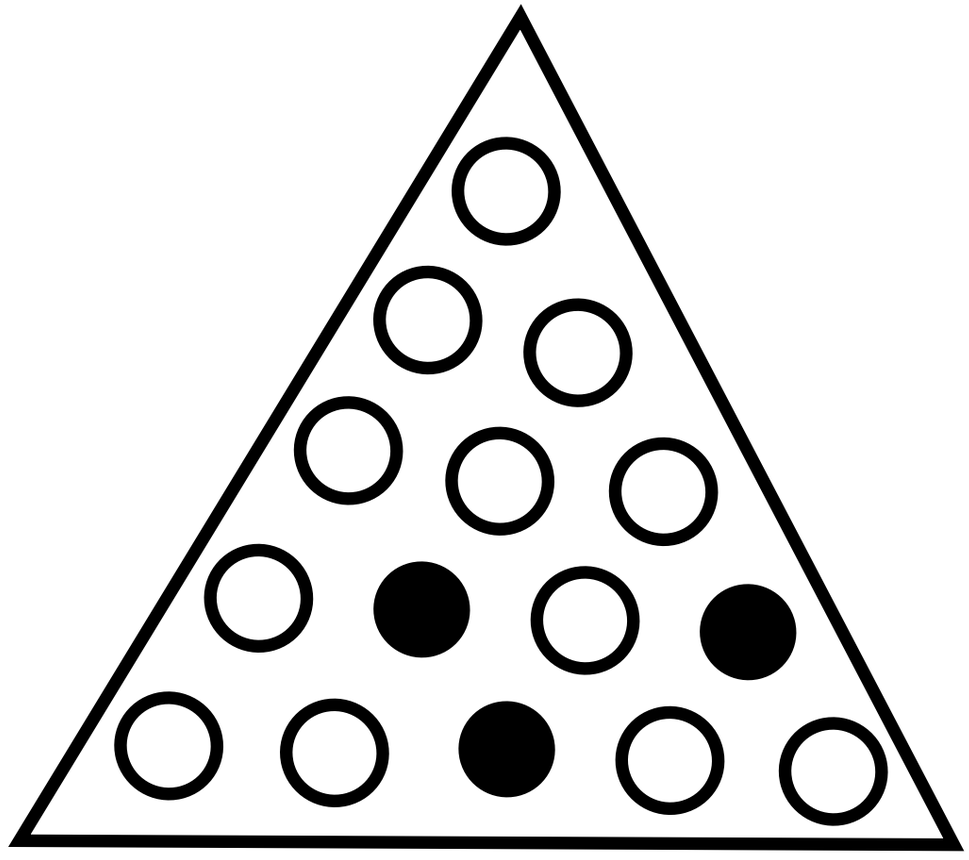
Applying Monte Carlo Tree Search to Peg Solitaire

- Note that we can exploit spatial symmetries, just as with Go:



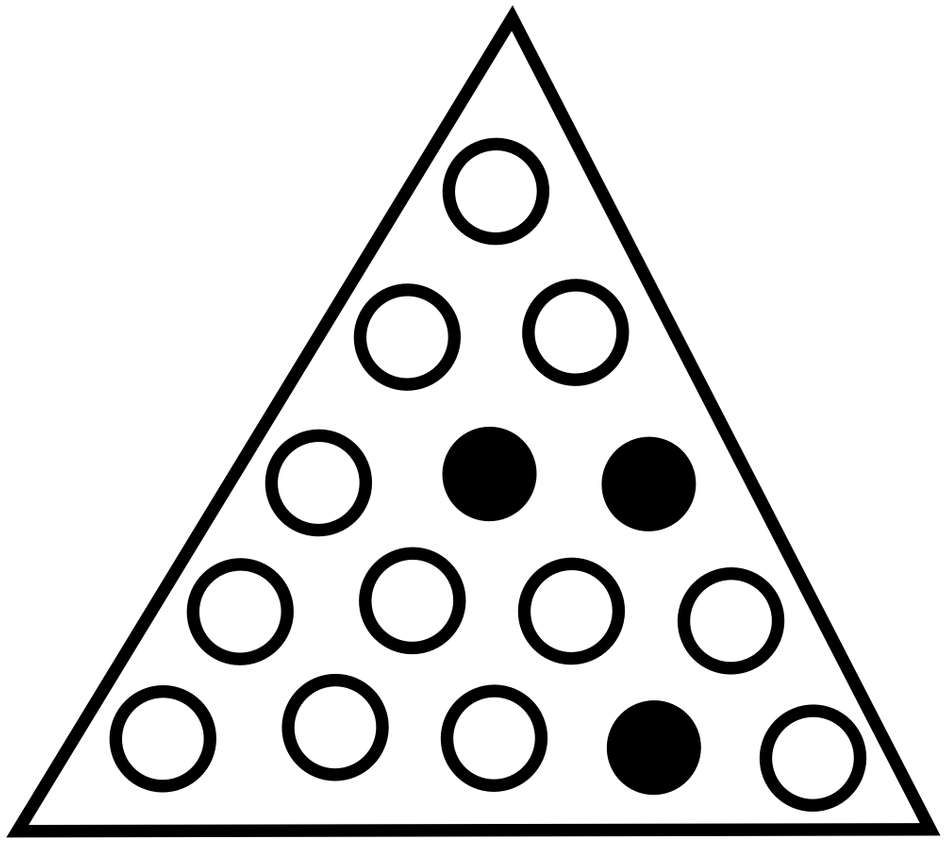
Applying Monte Carlo Tree Search to Peg Solitaire

- Note that we can exploit spatial symmetries, just as with Go:



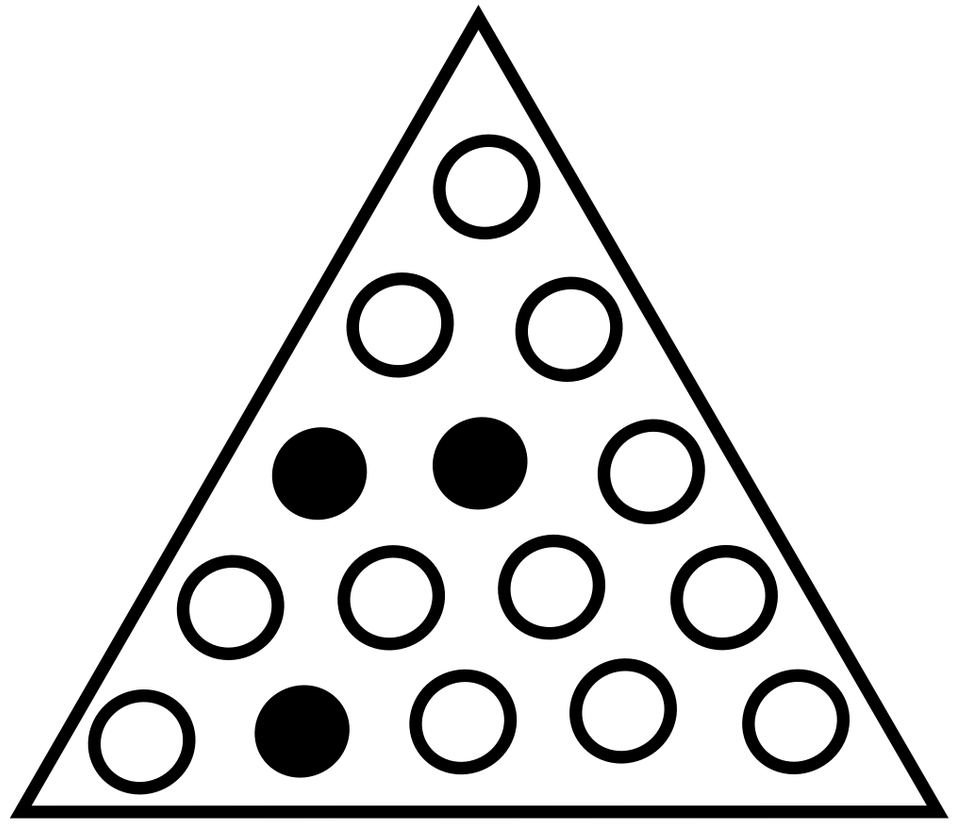
Applying Monte Carlo Tree Search to Peg Solitaire

- Note that we can exploit spatial symmetries, just as with Go:



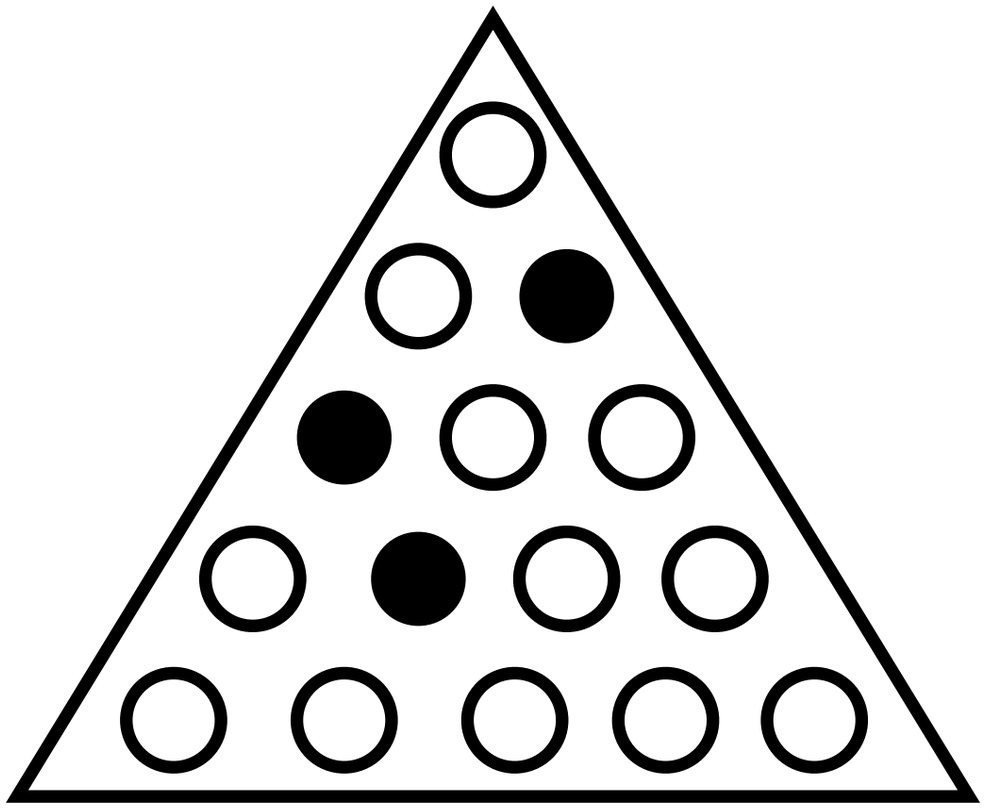
Applying Monte Carlo Tree Search to Peg Solitaire

- Note that we can exploit spatial symmetries, just as with Go:



Applying Monte Carlo Tree Search to Peg Solitaire

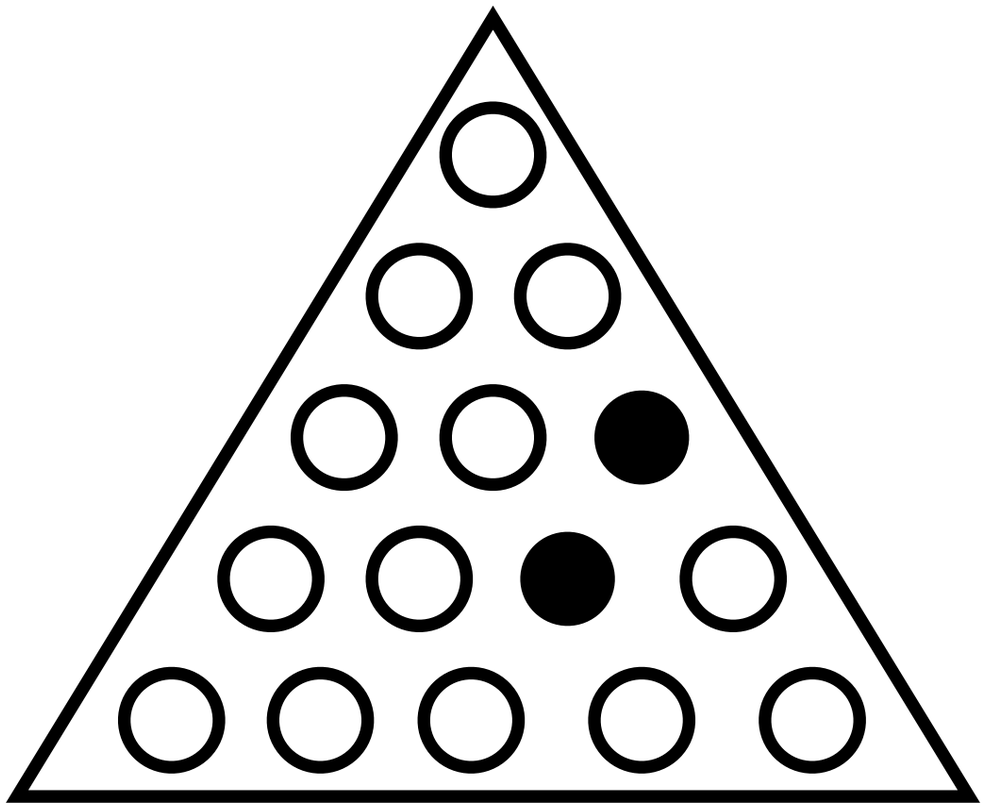
- Note that we can exploit spatial symmetries, just as with Go:



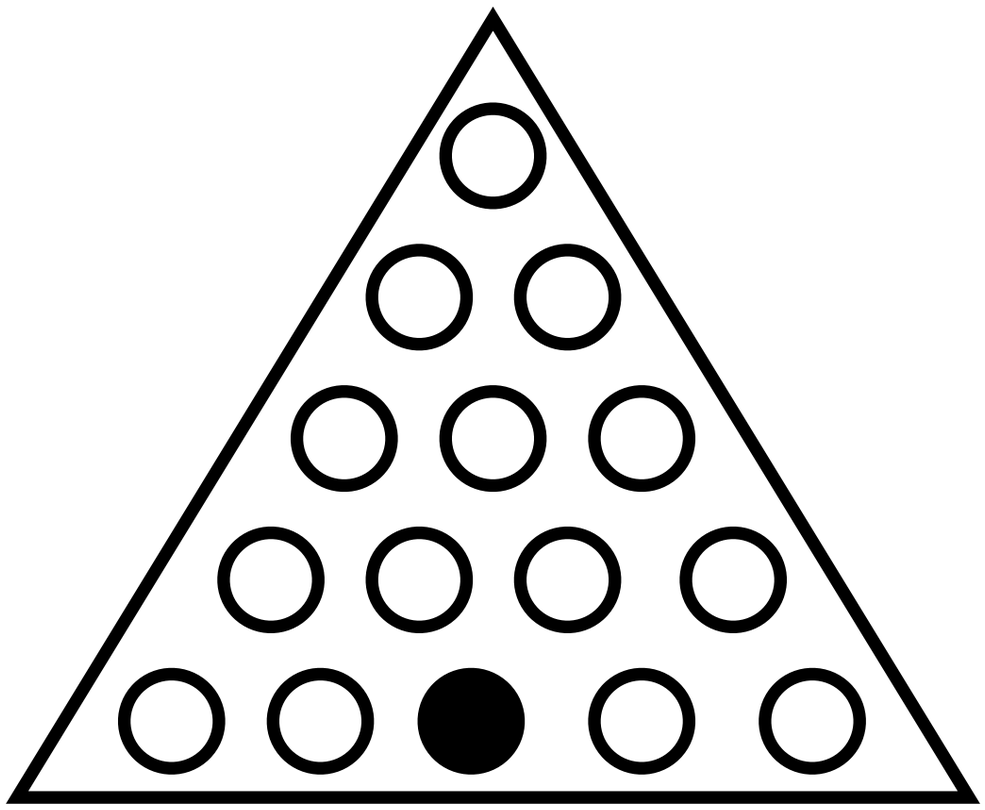
Temporal Symmetry

- Consider any winning sequence of moves:
  - It starts with a single hole
  - It ends with a single peg
- We could imagine playing the same sequence of moves backward in time, switching the roles of holes and pegs

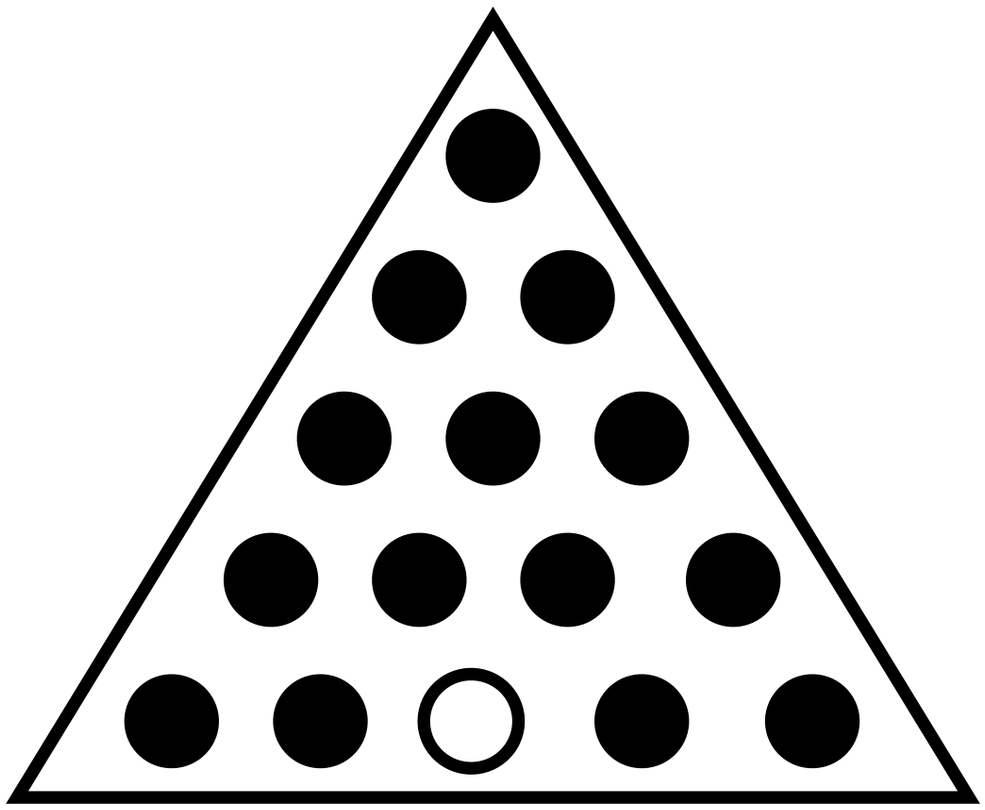
Temporal Symmetry



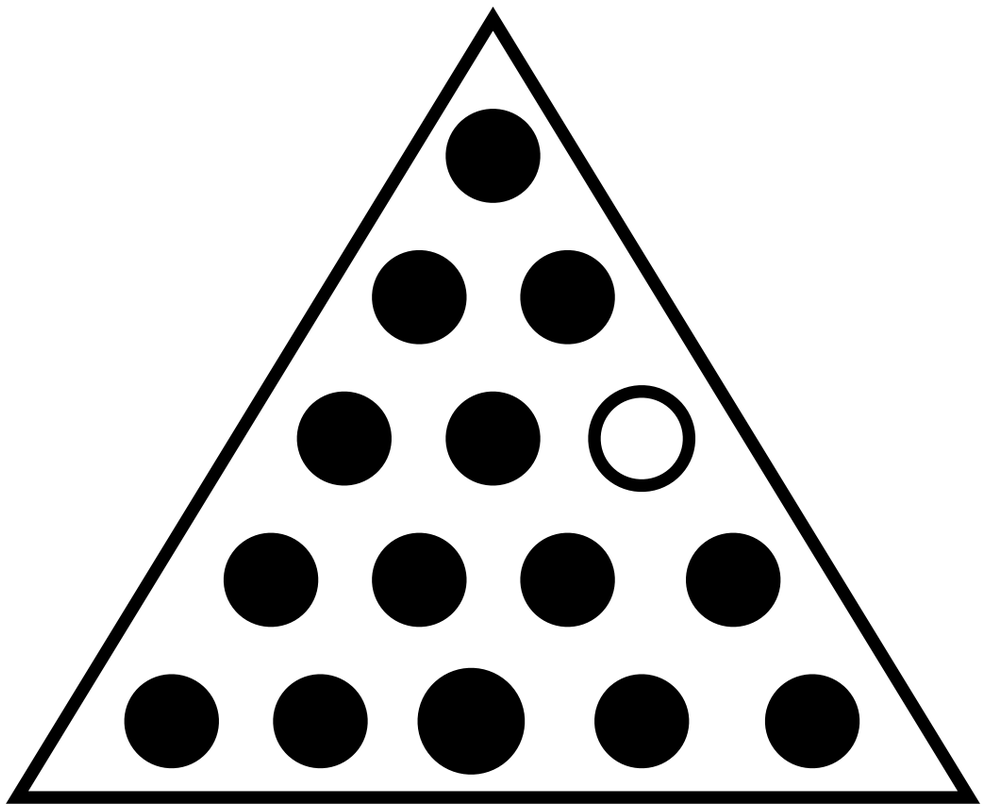
This could be the last move of a winning game…



Or the first move of the reverse (winning) game…



Or the first move of the reverse (winning) game…



“…This game can more elegantly be played backwards, after one stone has been put at will on an empty board, by placing the rest with it, but the same rule being observed for the addition of stones as was stated just above for their removal…”

–Gottfried Wilhelm Leibnitz
Miscellenea Berolinensia **1** (1710) 24.

Problem

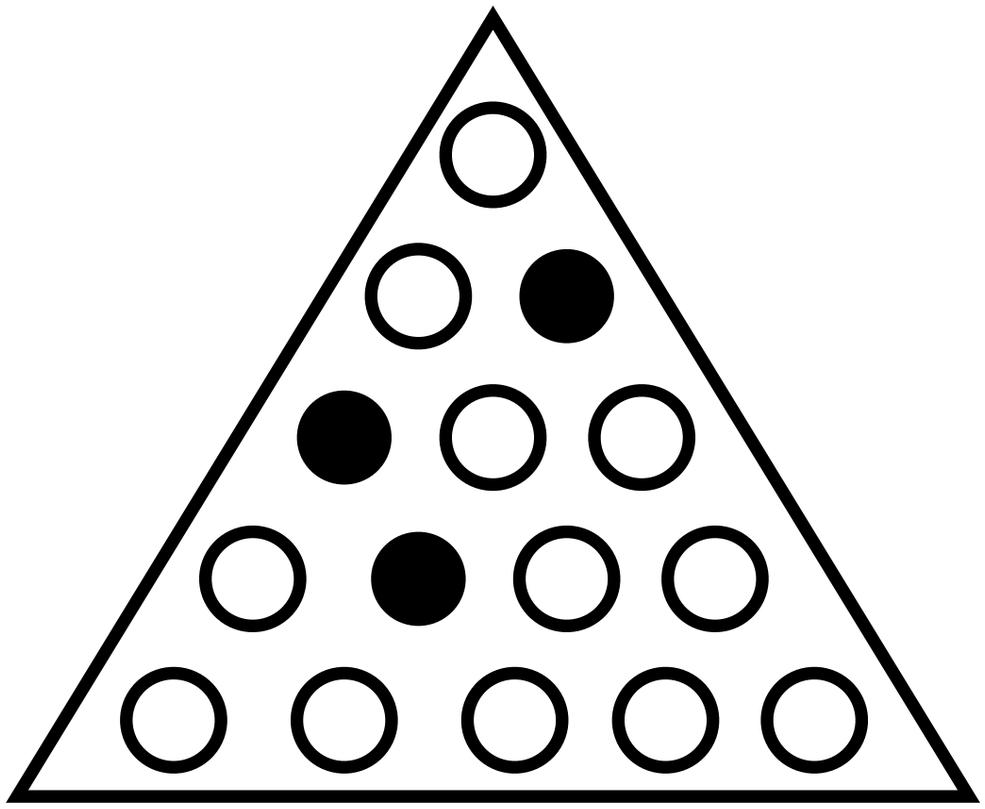
- To apply an AlphaGo Zero strategy to peg solitaire, we need to construct positive and negative examples of winning boards to train it with
- But almost all sequences of moves are losses!
    - Contrast with Go, or any two-player zero-sum game
- Also, if we have a winning sequence to train the neural network with, we no longer need the neural network to find it!

Solution: Bootstrapping

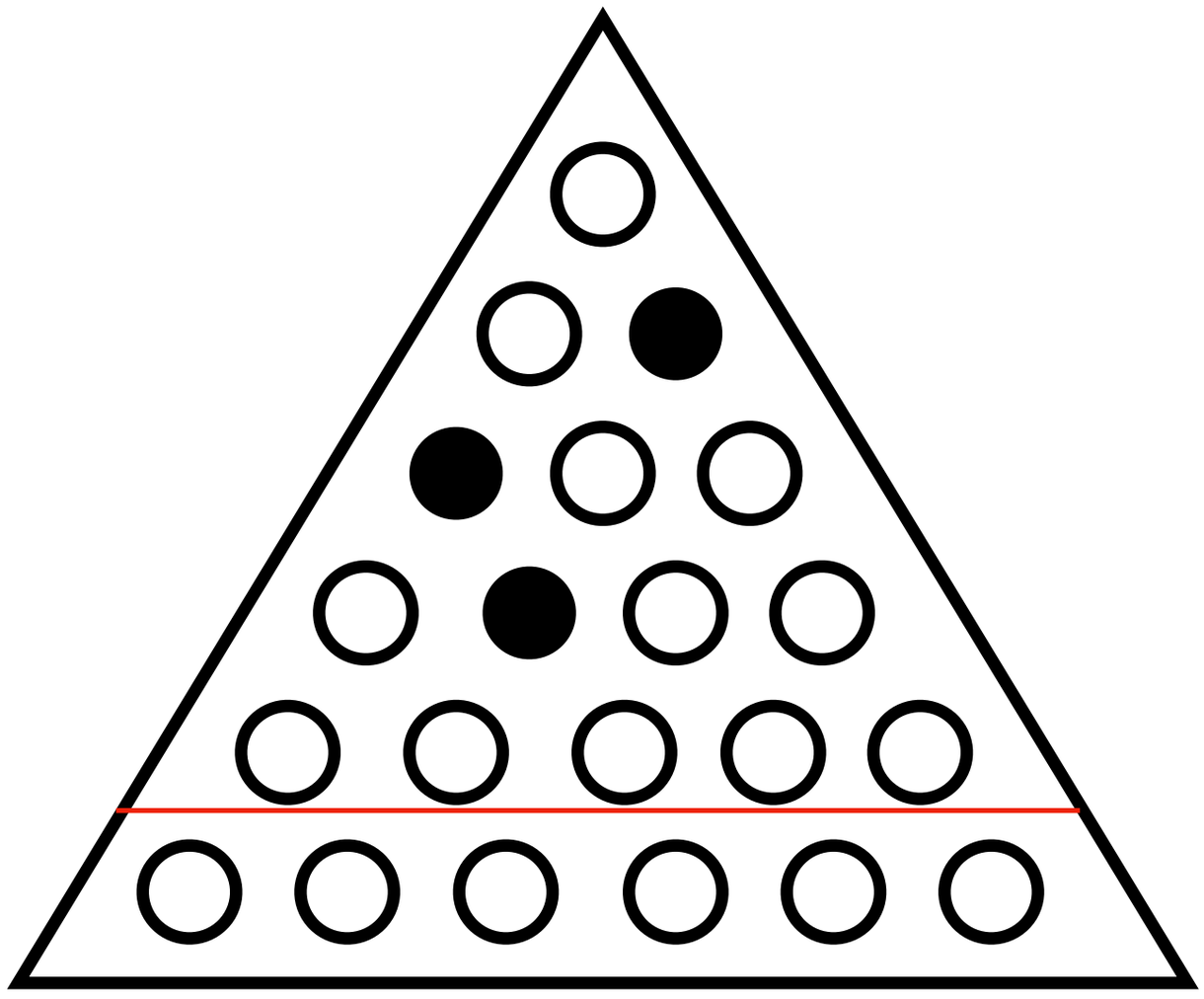
- We can incrementally solve boards of increasing size

- When winning boards are found for a board of size *n-1*, we can tile that board into a board of size *n*

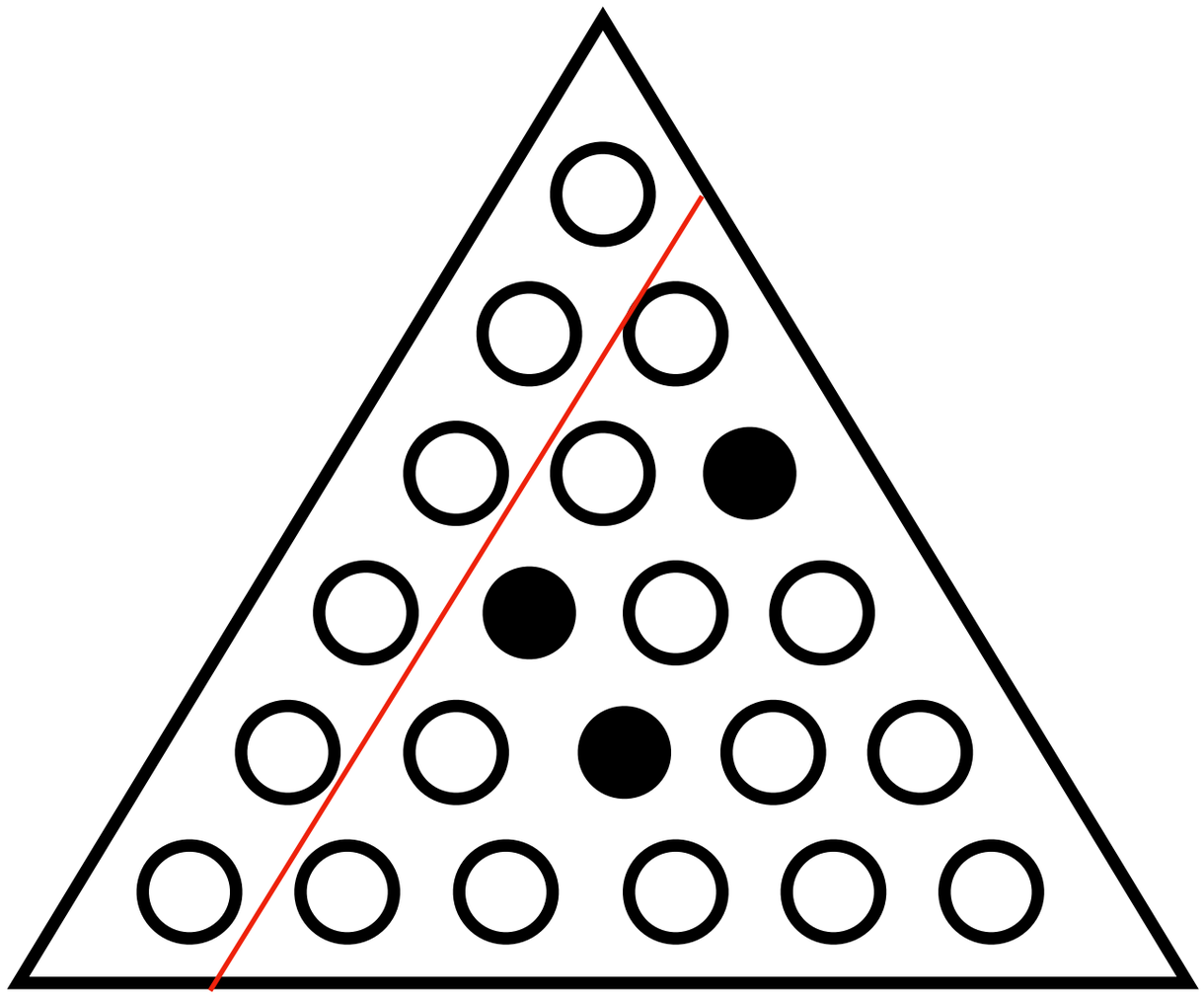
Tiling into a Larger Board



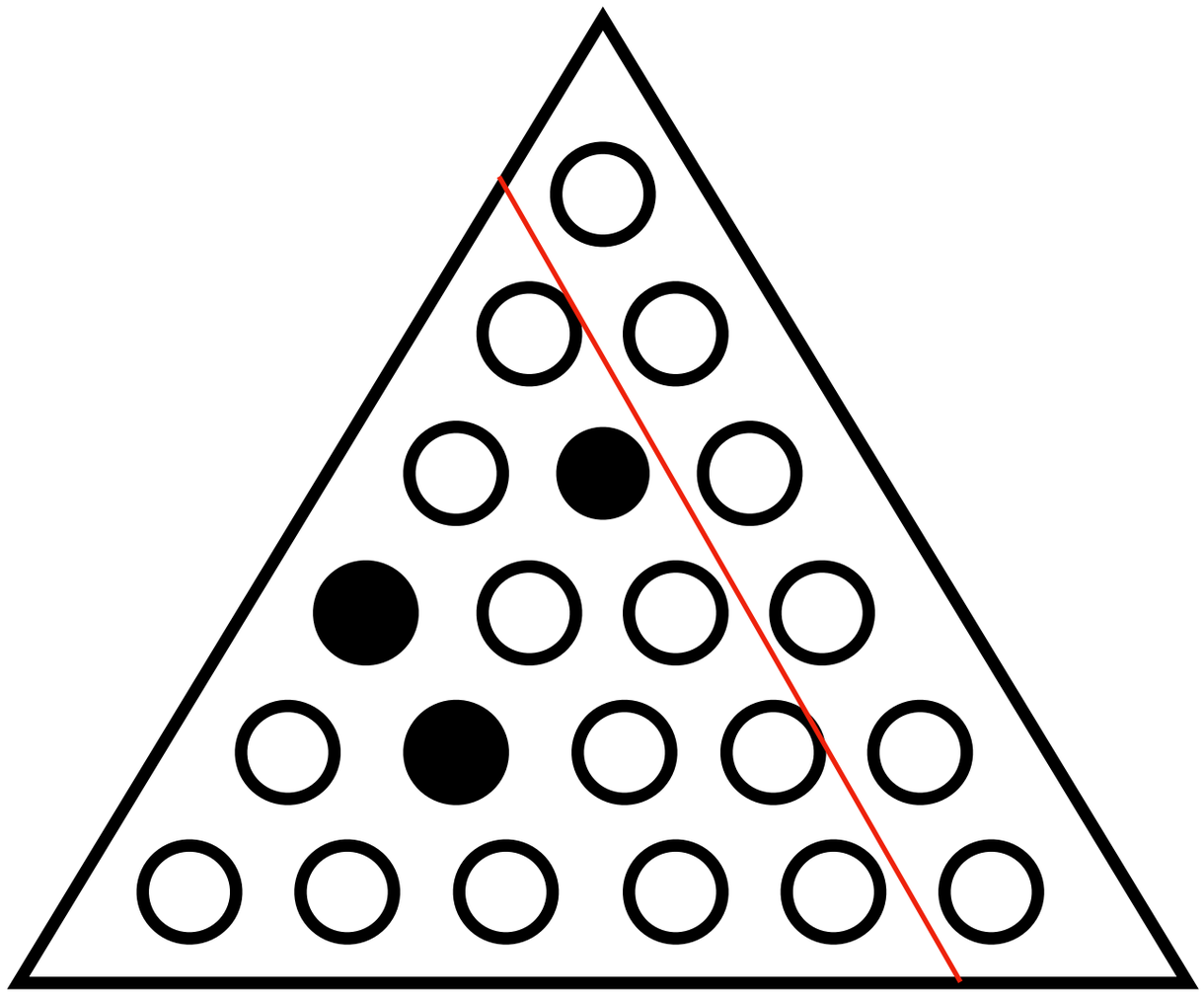
Tiling into a Larger Board



Tiling into a Larger Board



Tiling into a Larger Board



Applying Monte Carlo Tree Search to Peg Solitaire

- For each board size (starting from five), twenty solutions are found

- Every spatial and temporal symmetry of each of the twenty winning board sequences is recorded in the score for those boards

- The cache is the tiled up to the next sized board

- By size nine, this results in roughly 200,000 known boards that the system can win from

Optimizing for Peg Solitaire

- Up to boards of at least size nine, neural network training does not pay for itself
- Memory pressure caused by remembering losing board sequences does not pay for itself
    - Forget losing sequences and turn the exploration coefficient to zero

Conclusions

- When applying an algorithm to a new problem, be sure to consider each part of the algorithm and check whether it's needed in your application

  - Neural networks, exploration

- Think about the structure of the problem you are trying to solve

- Bootstrapping data from smaller versions of the same problem is a powerful technique

- Monte Carlo Tree Search is awesome

