Feature Learning for General Games

Cameron Browne

Search and Parallel Computing Unit
Advanced Intelligence Project (AIP)
RIKEN Institute, Tokyo

Oct 2017 — Mar 2018

Aim

General Game AI

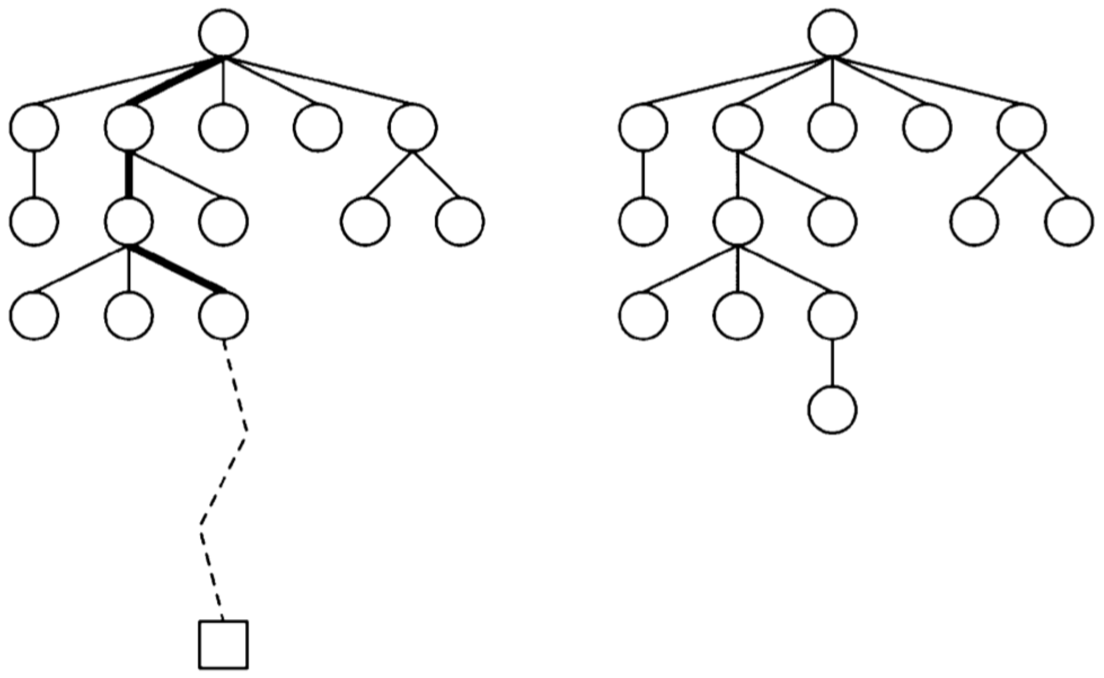
- Play any given game
- Strong human level
- Standard hardware

Approach

- Monte Carlo Tree Search (MCTS)
- Learn relevant features
- Bias playouts

MCTS

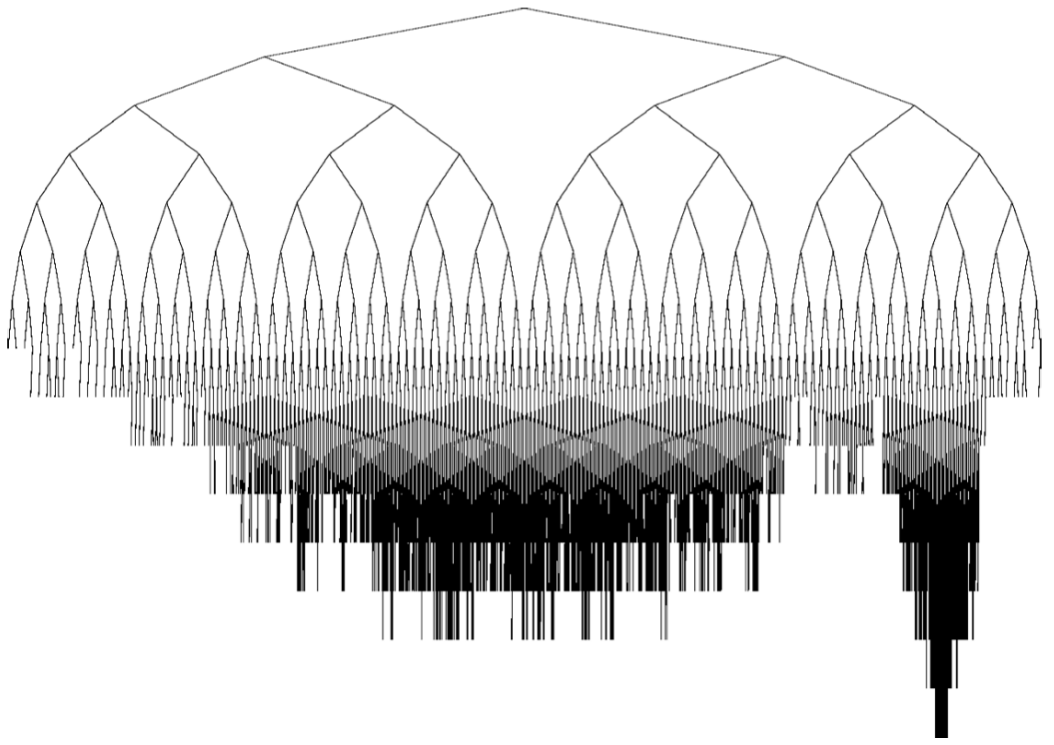
MCTS
- Run *N* simulations
- Build search tree



General Game Playing
- MCTS very successful
- World champion AIs for last 10 years
- Still weak w/o domain knowledge

Improvement
- Bias playouts
- More realistic results
- Better estimates

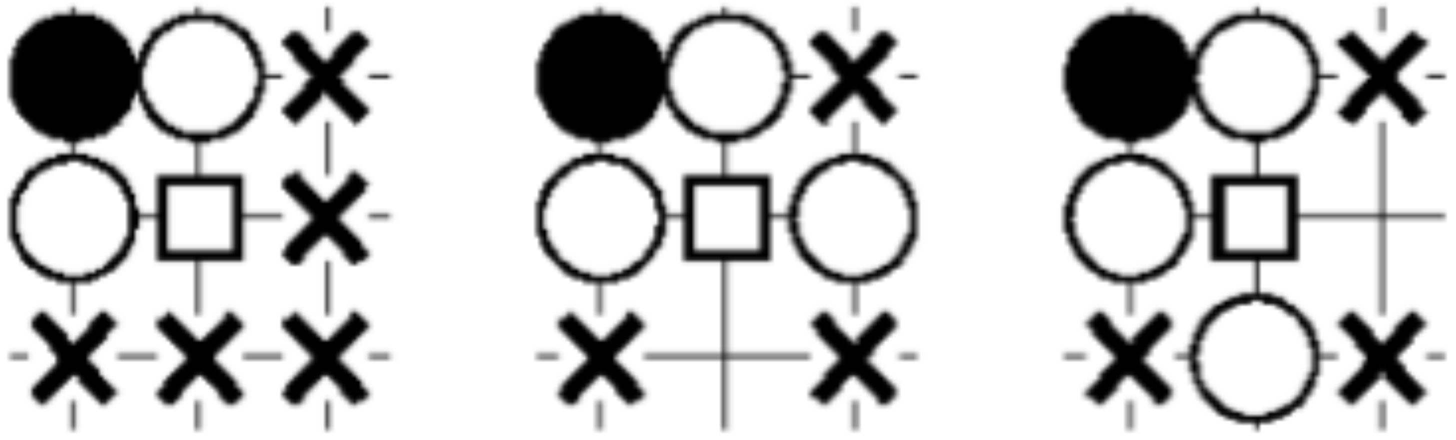
- Explore promising areas



Features

Computer Go

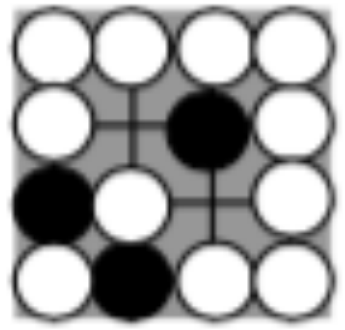
- Geometric piece patterns
- Handcrafted
- e.g. "Cut" pattern:
  – Gelly *et al.* (2006)
- Bias MCTS playouts:
  – Win rate: 41% ⇒ 80%



Patterns

Automated Learning

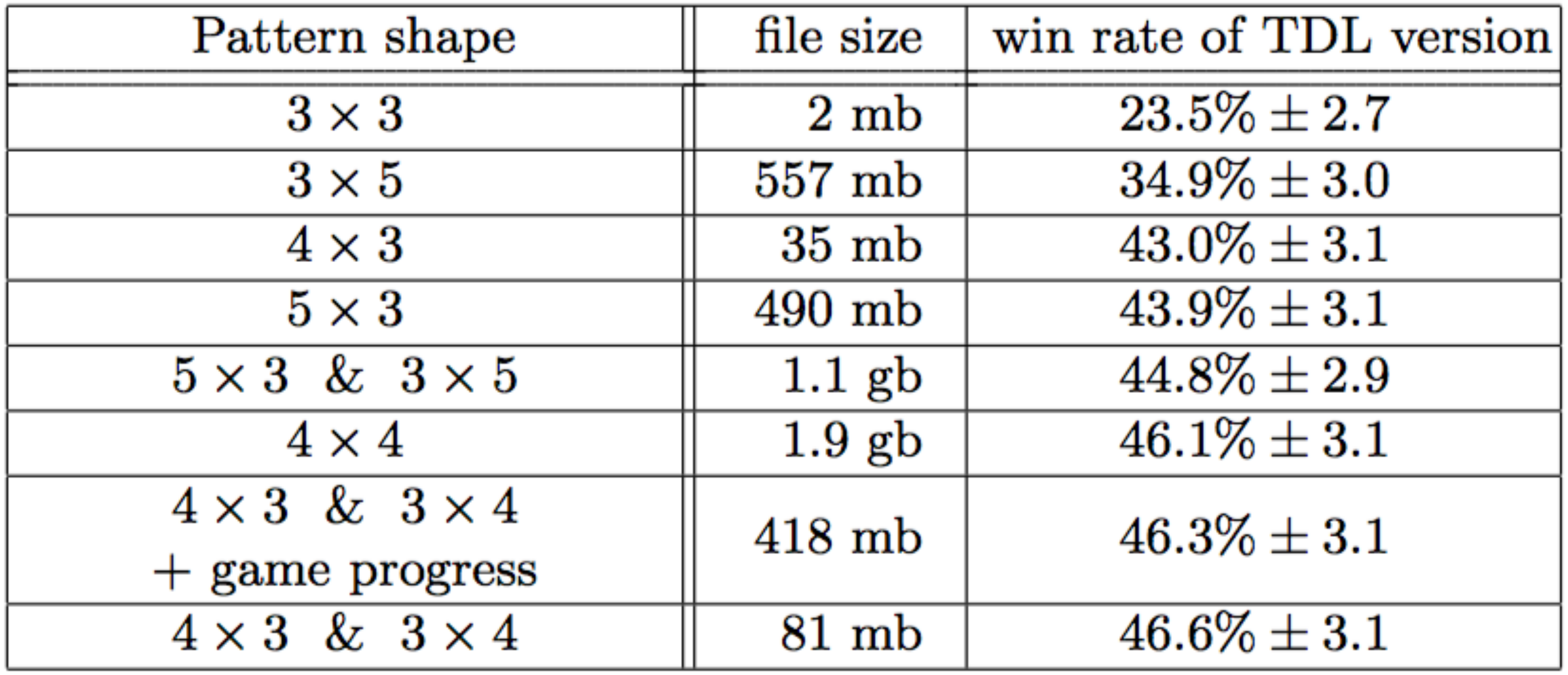
- Bouzy (2001):
  – Go, Retrograde analysis, MC



D = {H | K }

- Stern *et al.* (2006):
  – Go, Bayesian (harvested from expert games), MCTS

- Lorentz (2017):
  – Breakthrough
  – TDL(ג)
  – MCTS
  – Okay results
  – Big file sizes!

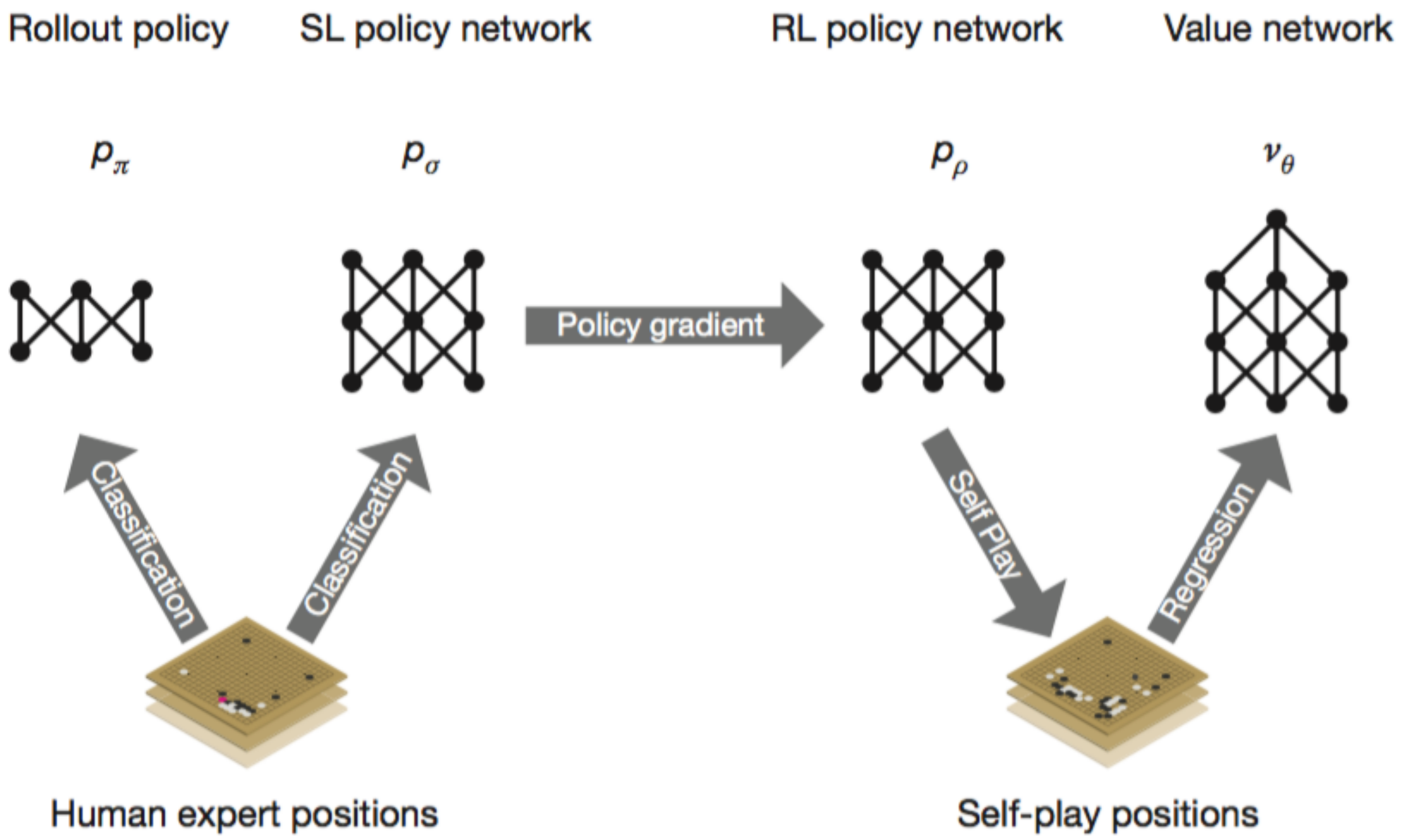


Pattern shape	file size	win rate of TDL version
3×3	2 mb	$23.5\% \pm 2.7$
3×5	557 mb	$34.9\% \pm 3.0$
4×3	35 mb	$43.0\% \pm 3.1$
5×3	490 mb	$43.9\% \pm 3.1$
5×3 & 3×5	1.1 gb	$44.8\% \pm 2.9$
4×4	1.9 gb	$46.1\% \pm 3.1$
4×3 & 3×4 + game progress	418 mb	$46.3\% \pm 3.1$
4×3 & 3×4	81 mb	$46.6\% \pm 3.1$

Google DeepMind (I)

AlphaGo Lee

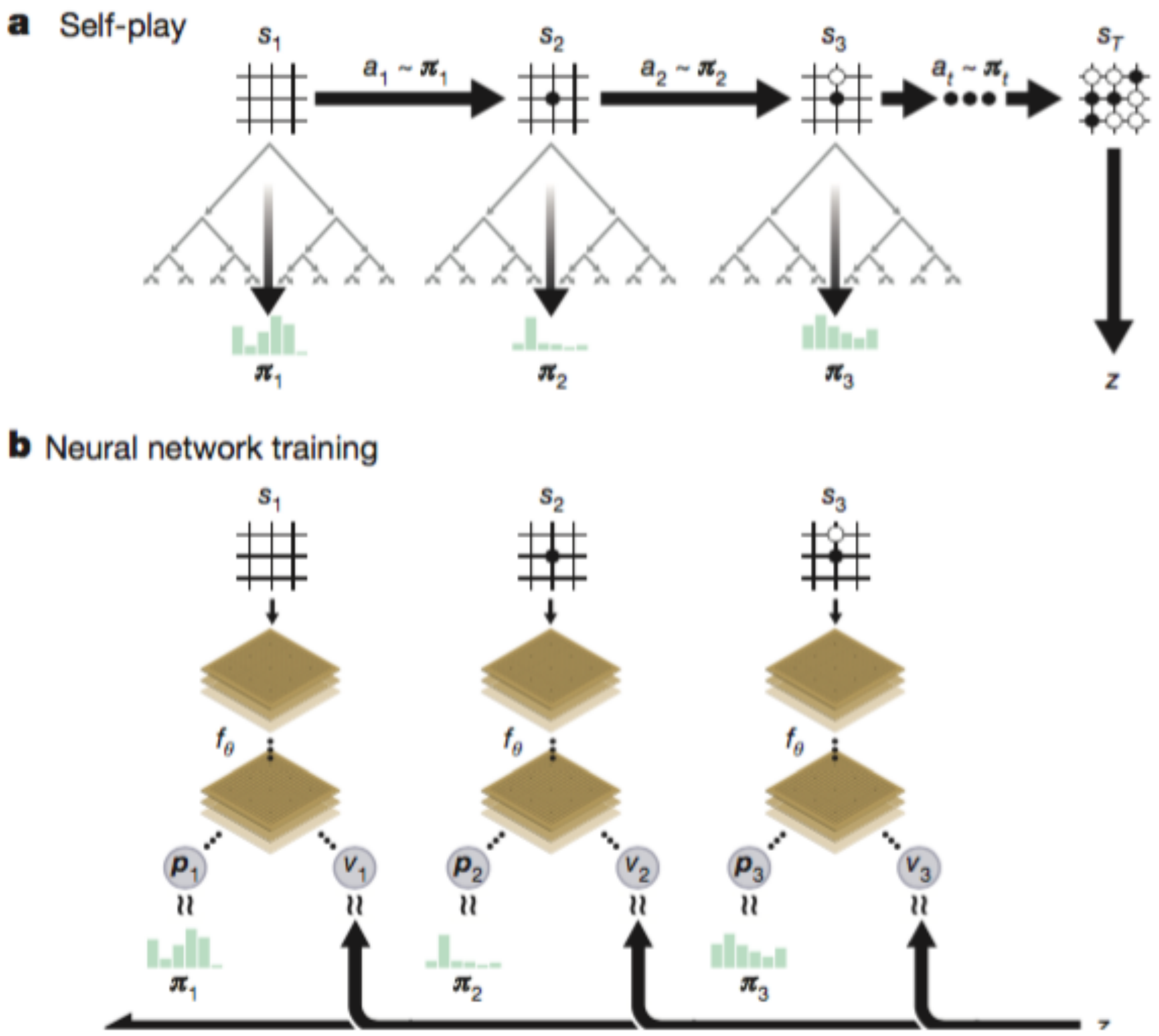
- Silver *et al.*, *Nature* (2016)
- Fast rollout policy
- Trained on expert games + self-play
- Geometric piece patterns:
  - 3x3 for "non-response"
  - 12-cell diamond for "response" moves



Google DeepMind (II)

AlphaGo Zero

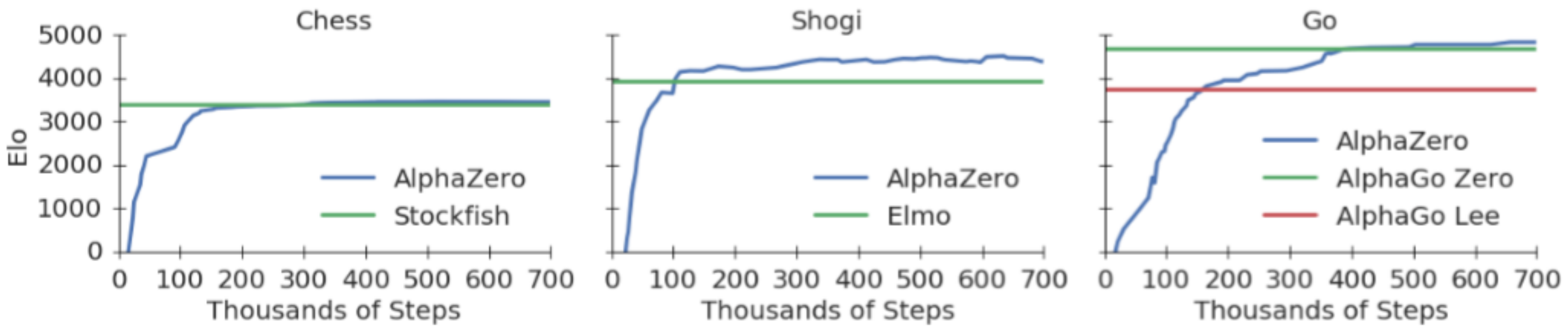
- Silver *et al.*, *Nature* (2017)
- Trained through self-play
- MCTS but no playouts!
- 3x3 convolution layer



Google DeepMind (III)

AlphaZero

- Silver *et al.*, *ArXiv* (2017)
- AlphaGo Zero approach:
  – Chess, Shogi, Go
- Superhuman level of play



AlphaZero

Good

- Superhuman results in difficult games
- Self-play (no expert database)
- Static and dynamic games
- Learns in good time
- General solution?

Bad

- Resources
  - Training, saving, playing
- Regular grid
- Case-by-case:
  - Architecture for each game
  - Trained from scratch (no transfer)

AlphaZero Resources

Training
- 5,000 x GPUs
- ~$25,000,000 hardware
- Several weeks
- On standard machine with GPU:
  – 1,700 years (Pascutto, *Computer Go* list, 2017)

Saving
- ANN with up to 2,000,000 parameters:
  – >1gb per game

Play
- Virtual machine (cloud)
  – 4 x TPUs

AlphaZero Geometry

Regular Square Grid
- Go, Chess, Shogi
- Small images
- Ideal for CNNs

General Games
- Other geometries
- Irregular bases

My Approach

Geometric Pattern Learning

- Bias MCTS playouts
- Invariant under geometry
- Fast application
- Small memory footprint

Aim

- Improve MCTS to strong human level (not superhuman!)
- Trainable on standard equipment
- Playable on standard equipment

Features

Patterns

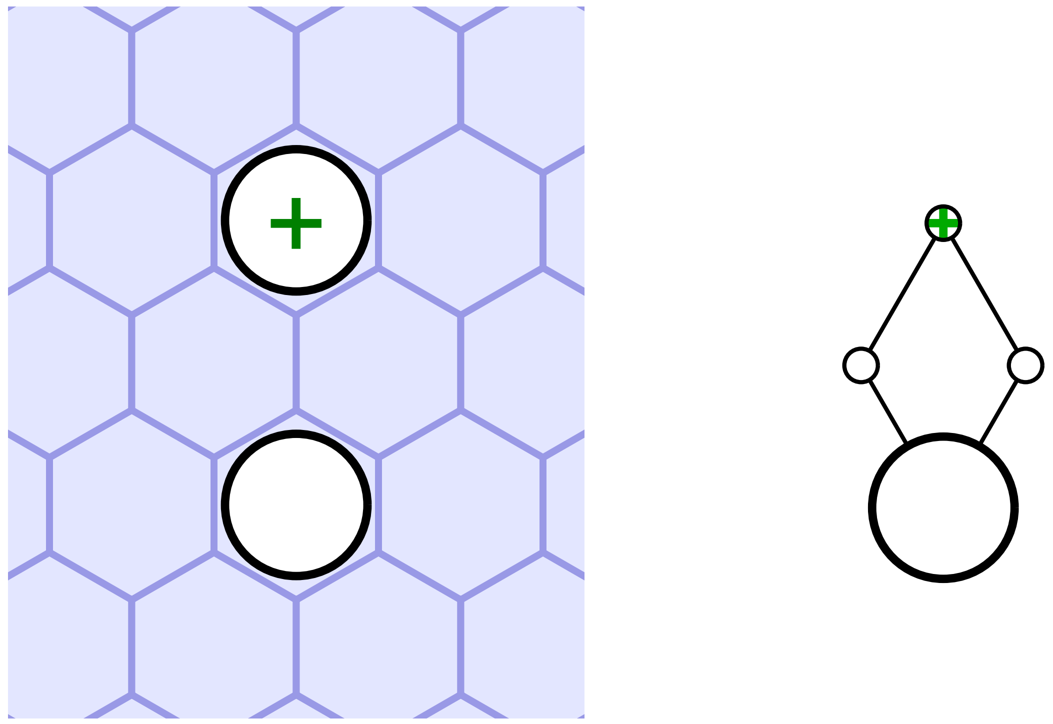
- Geometric piece patterns
- Indicate good/bad moves
- Use to bias MCTS playouts

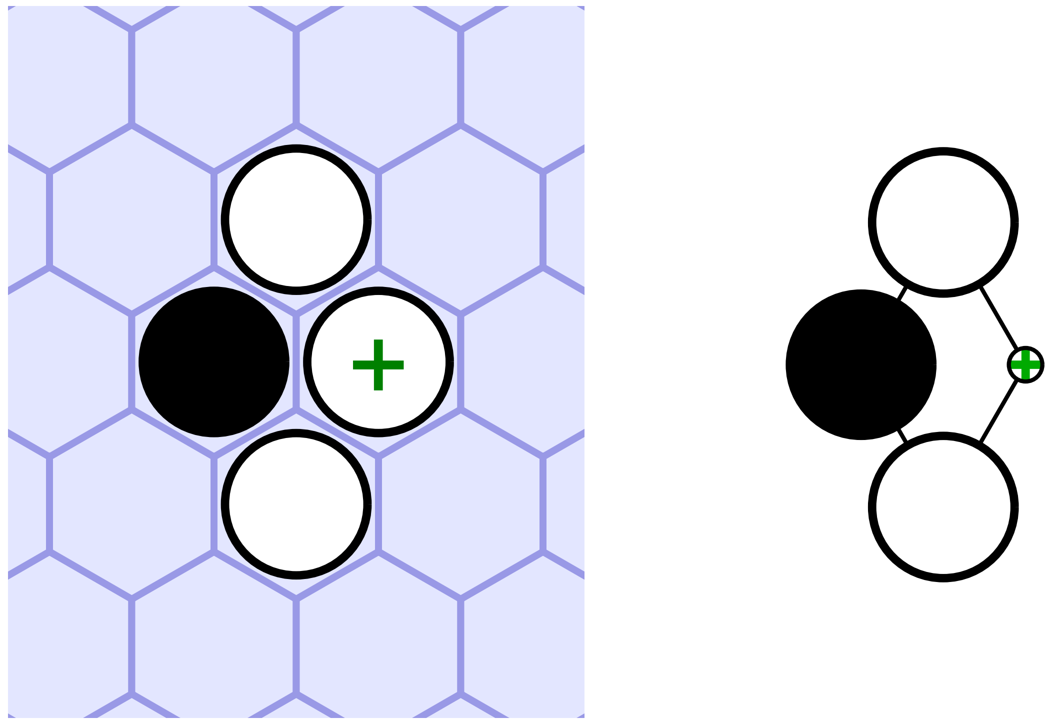
Examples

- Bridge extension/completion

Types

- Proactive (non-response):
  – Predict good move
- Reactive (response):
  – Reply to opponent's last move





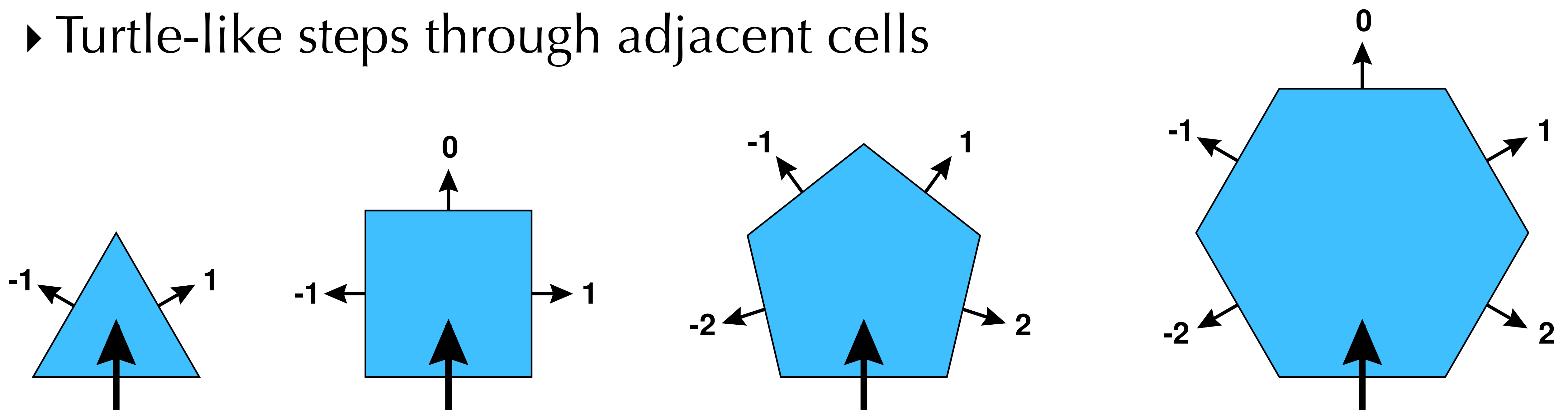
Geometry Invariant

Game Graph

- Based on adjacency
- Underlying board geometry

Cell Relations

- Not coordinates
- Relative locations
- Turtle-like steps through adjacent cells



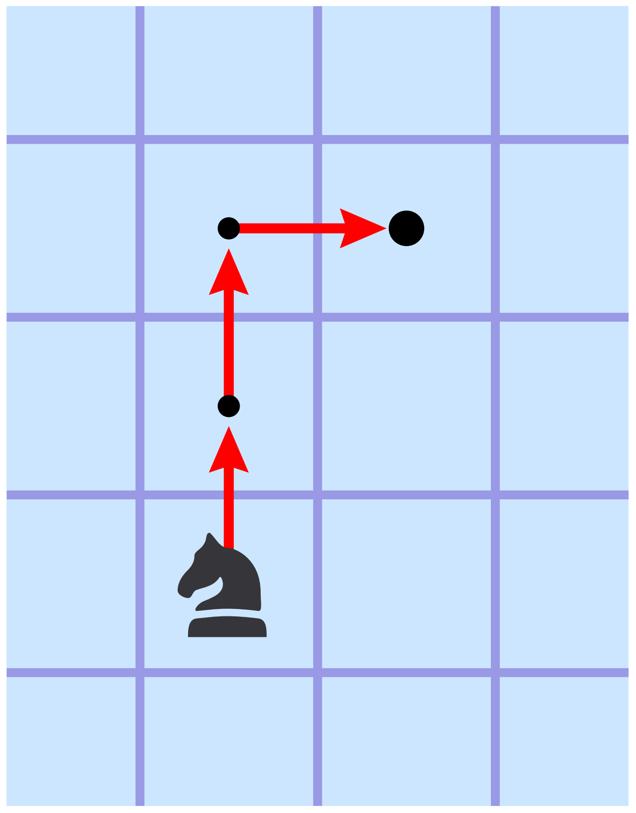
Example: Knight Move

Knight

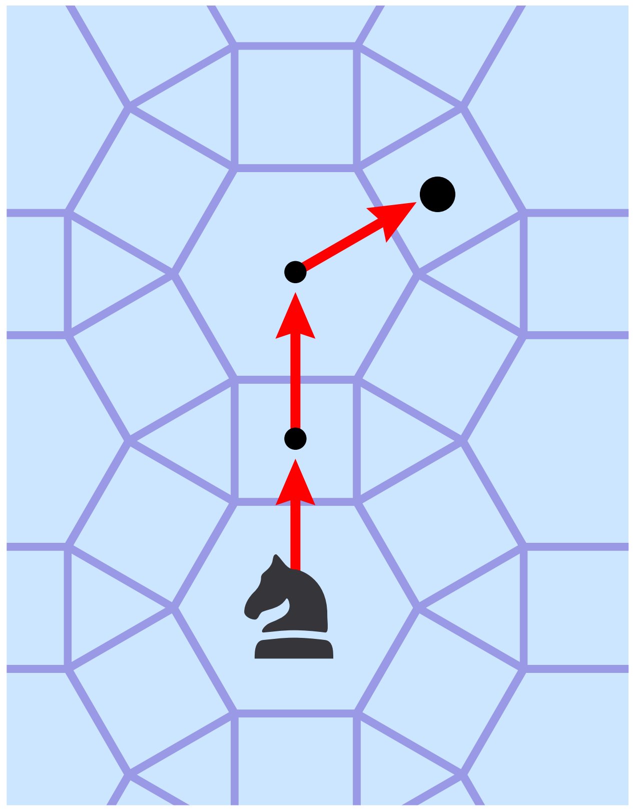
- Hippogonal
- Square grid: [1, 2]
- Arbitrary graph: {0, 0, 1}

Invariant

- Apply to other geometries
- Transfer to other games



$P_k = \{0,0,1\}$



Implementation (I)

Game Features 1.1

- Java 8 app
- Five games so far:
  - Override `Game` class
  - Dozen expected

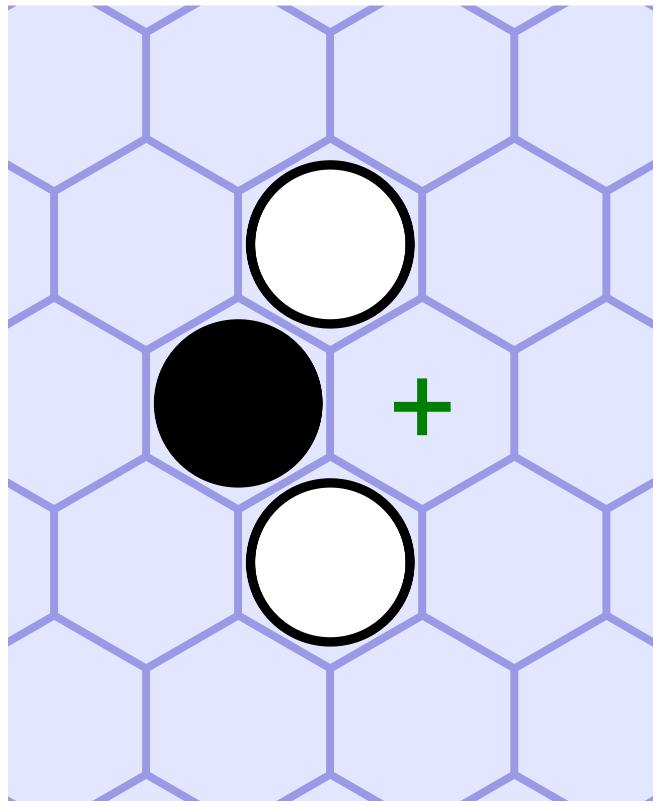
Game State

- Flat bitset (derived from standard `BitSet` class)
  - *n* bits per board cell (where *n* is a power of 2)

Patterns

- Each pattern contains *m* instances
- Each instance corresponds to a bitset
- Pre-generated for all possible reflections, rotations, translations
- Efficient pattern matching

Implementation (II)

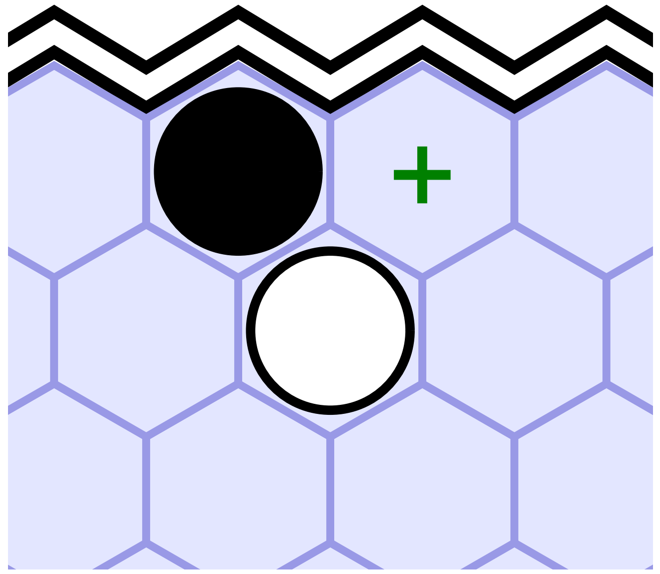


Example

- Hex patterns

```
    //    + f
    //   f e
  "Reactive bridge repair:All:act=<{-1}>:lst=<{}>:
   rot=D:val=0.5:pat=<e{},f{0},f{-2},-{-1}>”

  //  #
  // + e
  //  f
  "Reactive edge bridge repair (1):1:act=<{}>:ref:
   lst=<{1}>:rot=2:val=0.5:pat=<e{1},-{},f{2},#{0}>”
```



Results

- Efficient: Speed loss ~1-2% per pattern
- Effective: 55% ⇒ 85% win rate vs MCTS
- Small: <100 bytes per pattern

Benefits

Improve AI Strength

- Strong human level play (not superhuman!)

Reveal Strategies

- Patterns encode strategies
- Explain in human-comprehensible terms
- Transfer to other games
- Reveal depth of game?

Reason

Game Quality

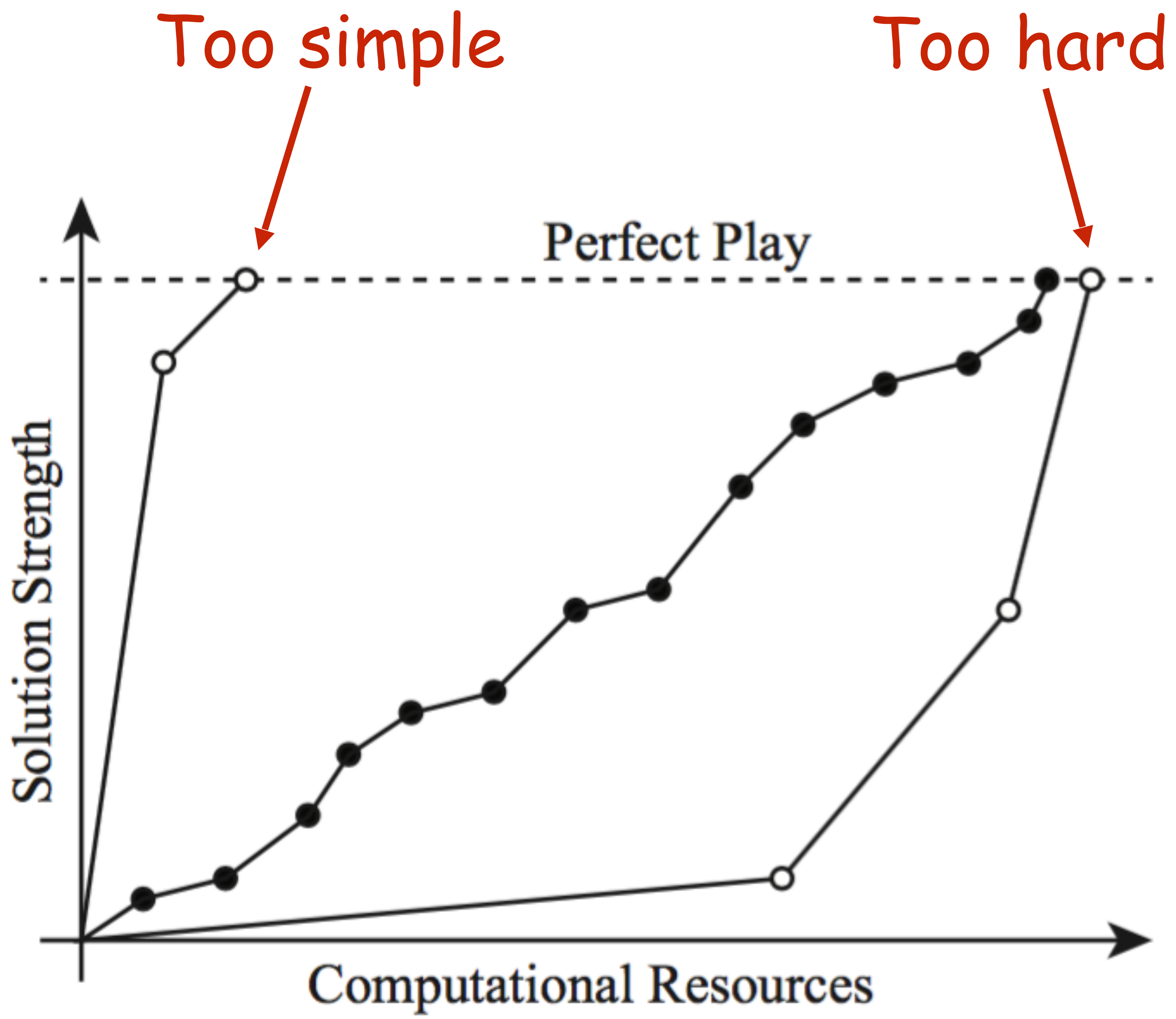
- Lantz *et al.* (2017)
  - Strategy ladder

Interestingness

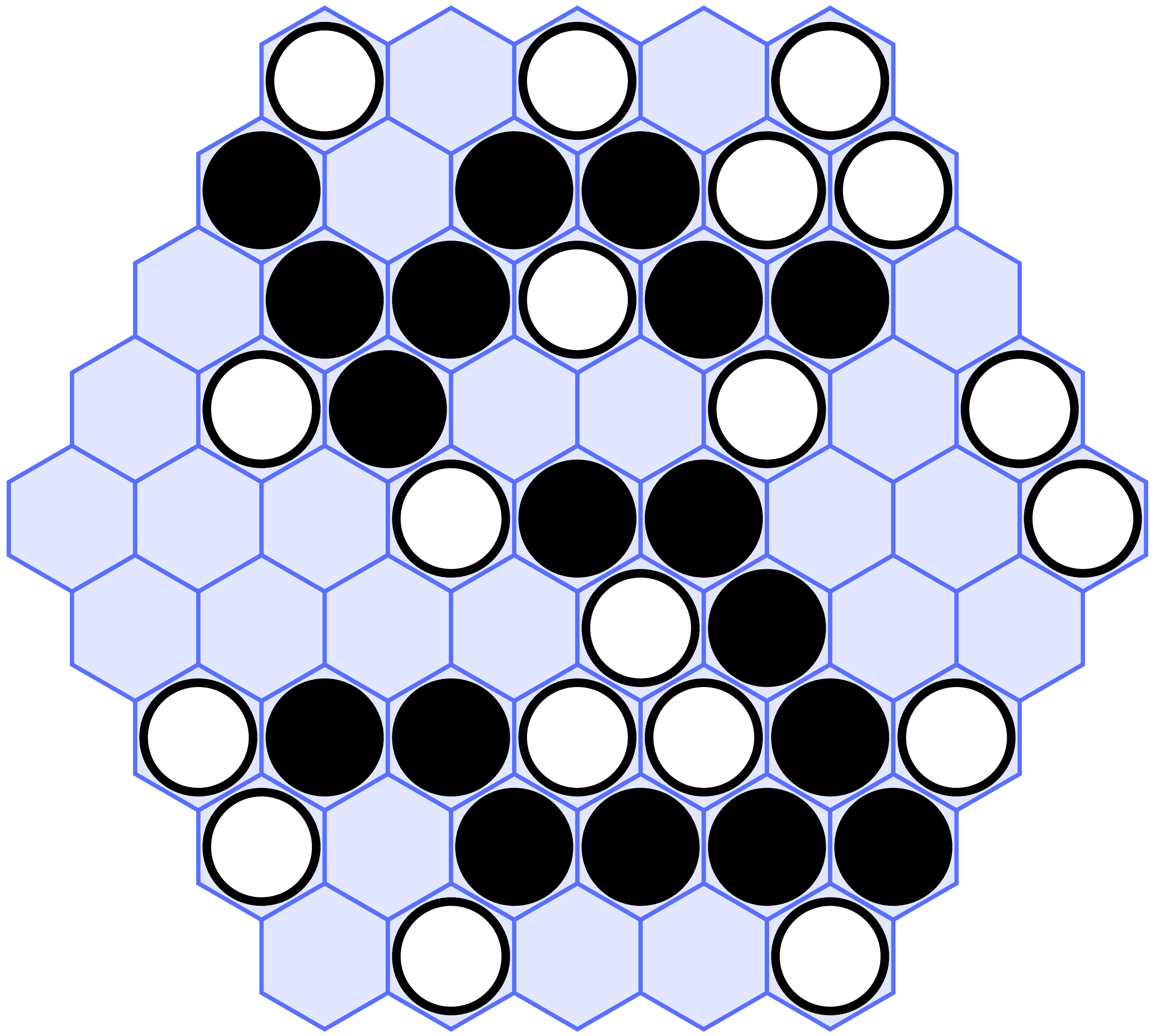
- Allis *et al.* (1991)
  - *"intellectual challenge neither too simple nor too hard"*

Hypothesis

- Each related subset of patterns encodes a strategy



Strategy Example (I)



Quantum Leap

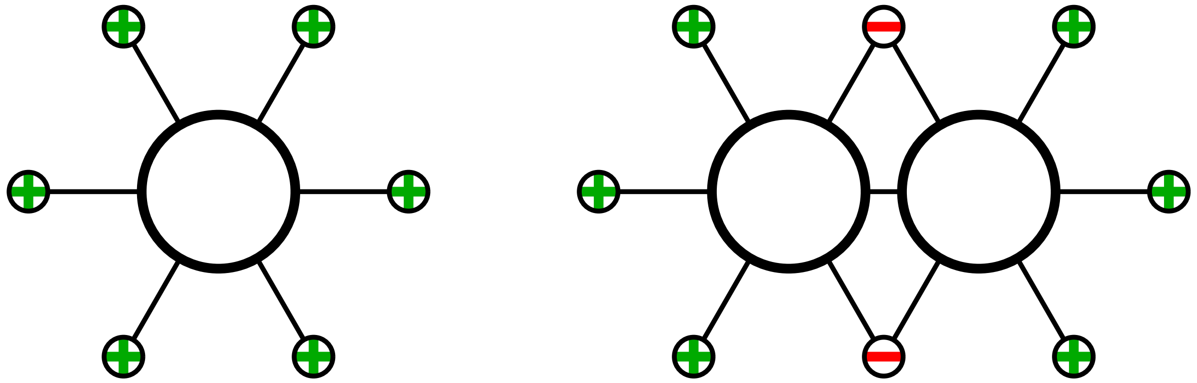
- Move in line to capture
- Distance = friendly nbors

MCTS

- Unbeatable with 1–2s
- Random playouts

Strategies

- 1. Form groups (max. movable pieces)
- 2. Form *thin* groups (max. moves)



Expected Patterns

- 1. Form groups (left)
- 2. Expand thinly (right)

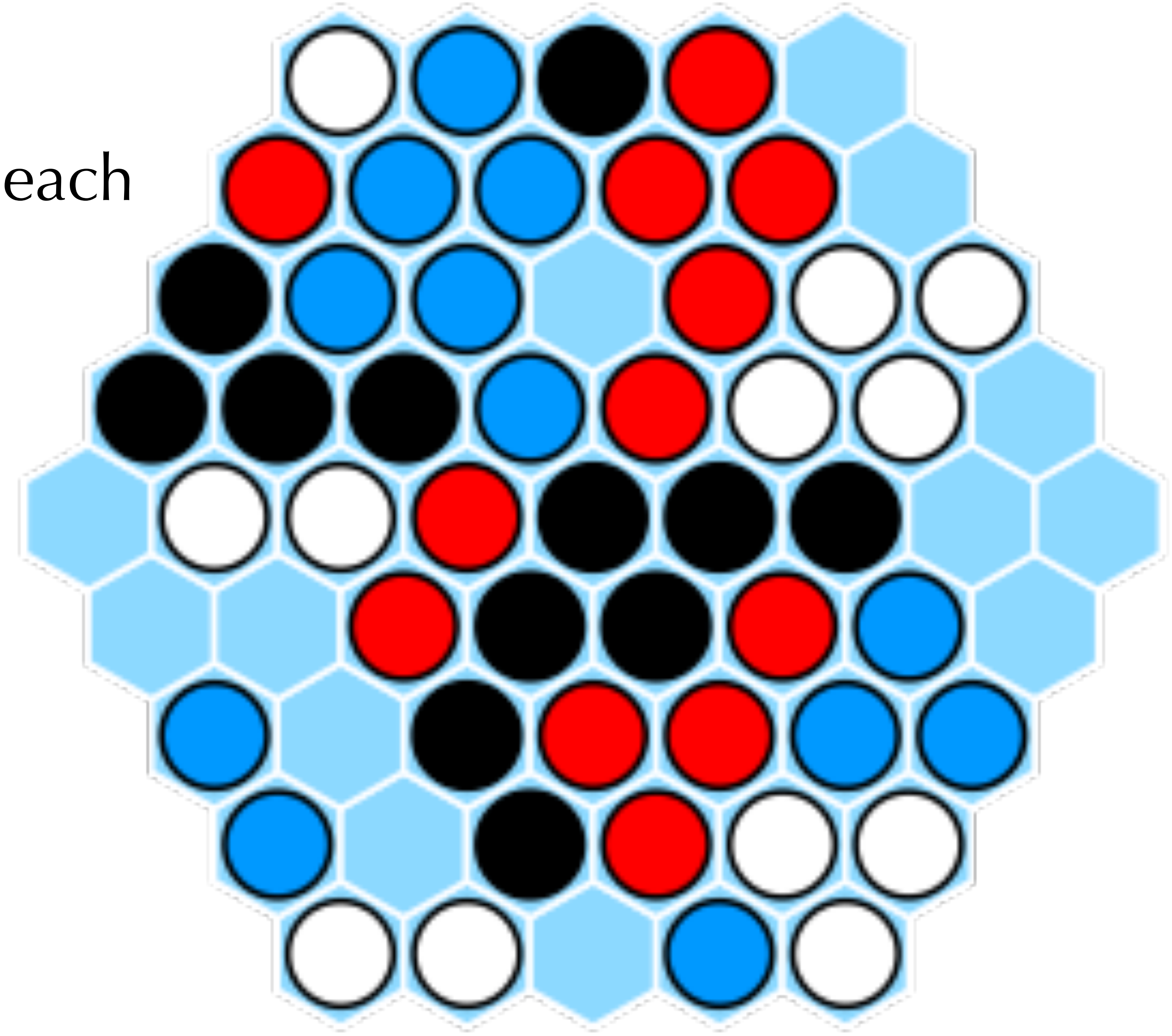
Strategy Example (II)

Omega (2010)

- Players place a piece of each colour per turn
- Score = product of group sizes

Who is winning?

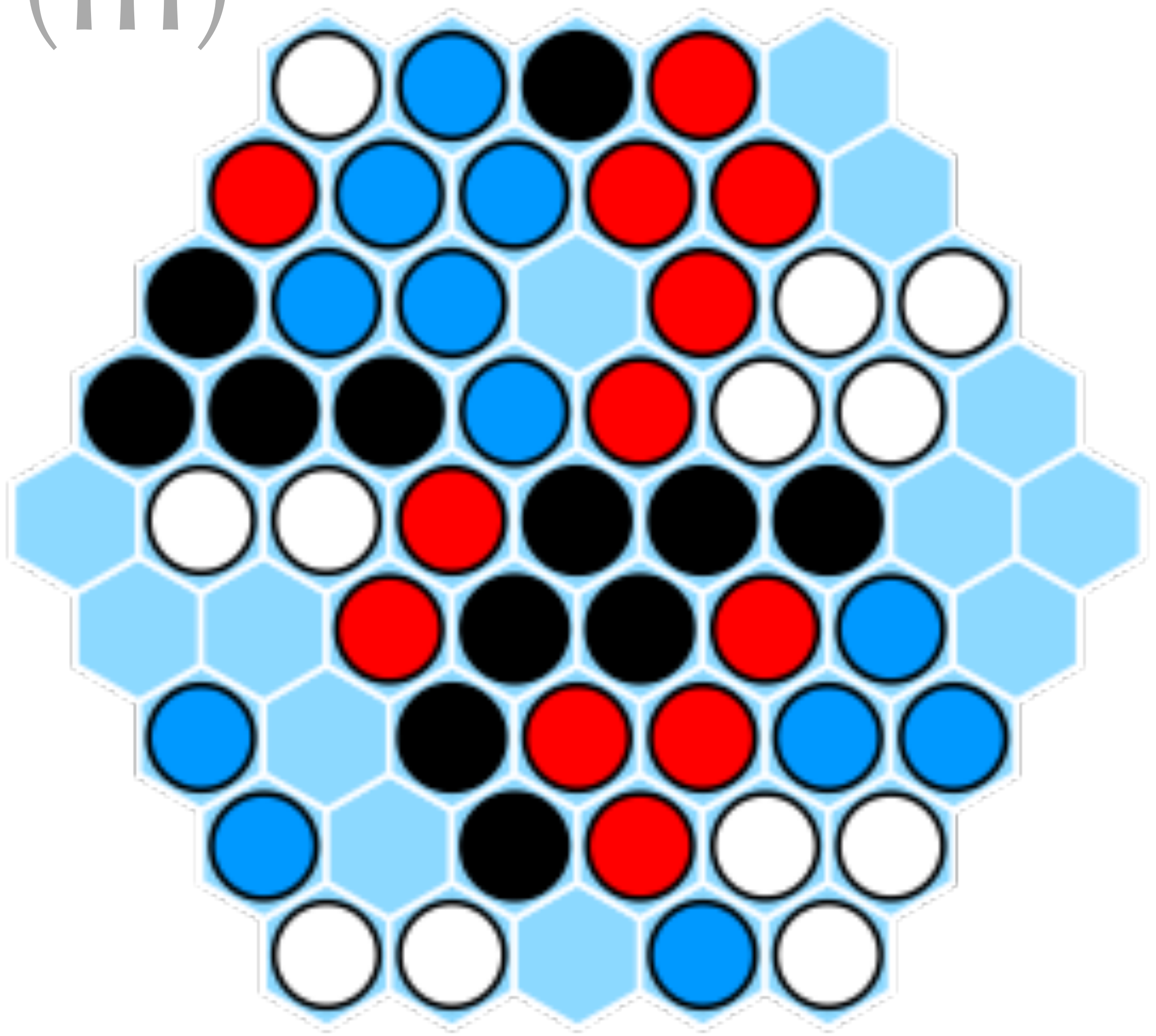
- Opaque
- Unpopular
- No strategy



White: 1×2×2×3×4 = **48** **Blue:** 1×2×3×6 = **36**

Red: 1×2×4×5 = **40** **Black:** 1×4×7 = **28**

Strategy Example (III)

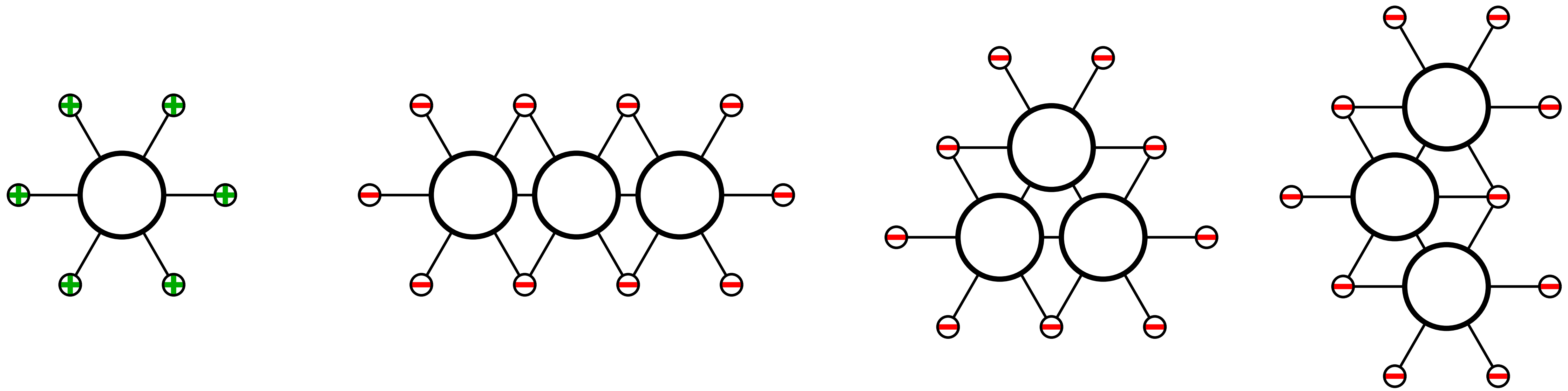


MCTS

- Strong with 1–2s
- Random playouts
- Emergent strategy:
  – Prefer groups of 3

Expected Patterns

- 1. Grow singletons (left)
- 2. Discourage groups > 3



Feature Learning

Feature Extraction

- Harvest from random self-play games
- Frequent pattern mining

Frequent Tuples

- 1-tuple, 2-tuple, … , 6-tuple
- Within three steps
- Types: empty / off / friend / enemy / !empty / !off / !friend / !enemy

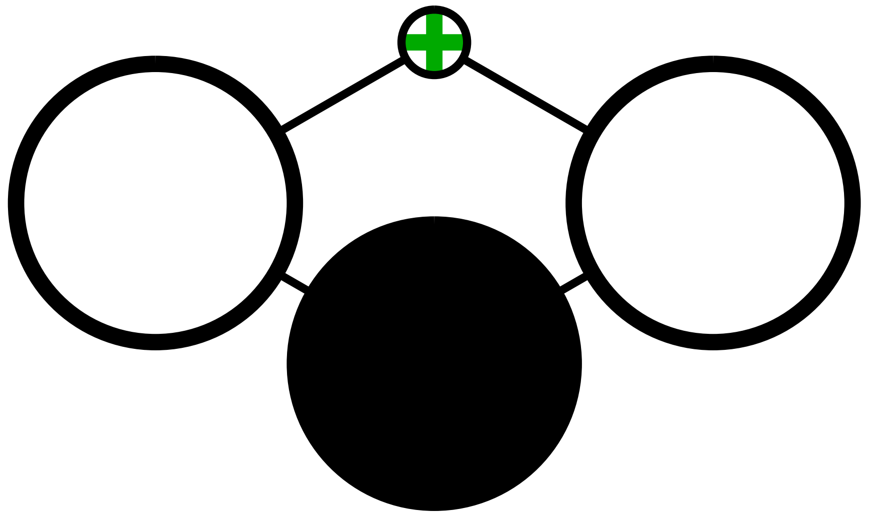
Feature Selection

- Self-play tournaments
- Biased MCTS playouts
- Optimise combinations

Random Self-Play (I)

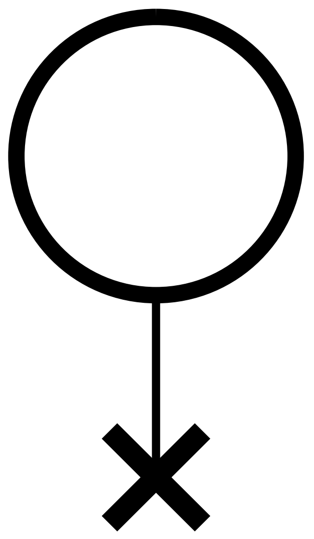
Random Self-Play

- Good for generation
- Not for evaluation!

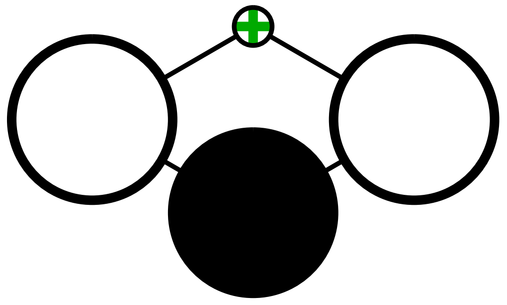


Example

- Hex: Two common patterns
  – $\boldsymbol{P_b}$: Bridge completion (reactive)
  – $\boldsymbol{P_e}$: Prefer enemy edge (proactive)

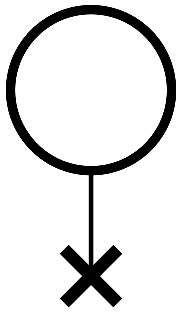


Random Self-Play (II)

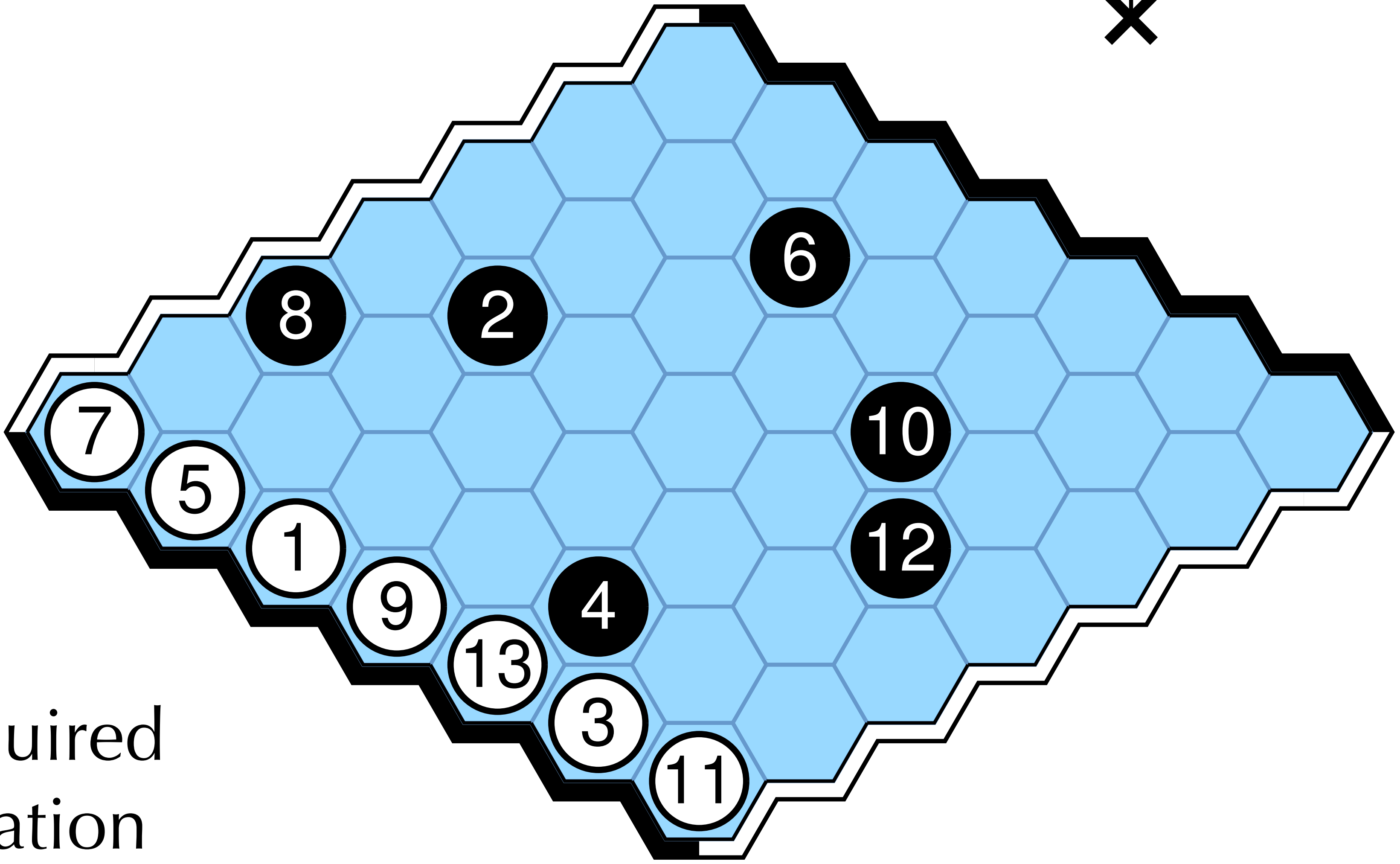


Random Self-Play

- Edge pattern $\boldsymbol{P_e}$ encodes degenerate strategy
- Outscores bridge pattern $\boldsymbol{P_b}$ in random play!



	Rand	MCTS
$\boldsymbol{P_b}$	65%	85%
$\boldsymbol{P_e}$	90%	35%



- MCTS slower but required for meaningful evaluation

Summary

Aim

- Improve AI for general game playing
- Strong human-level play
- Standard equipment

Progress

- Game representation finalised
- Feature representation finalised
- System implemented and working

Next

- Feature learning (extraction and selection)
- Further testing
- Further games