

# Evolutionary Learning of Policies for MCTS Simulations

James Pettit, David Helmbold

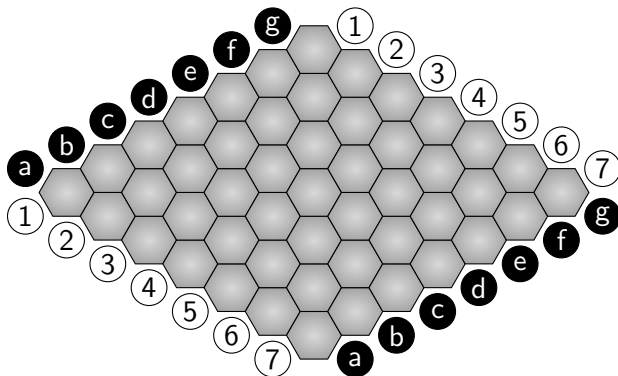
University of California, Santa Cruz  
jpettit@soe.ucsc.edu

July 2012

# Motivation

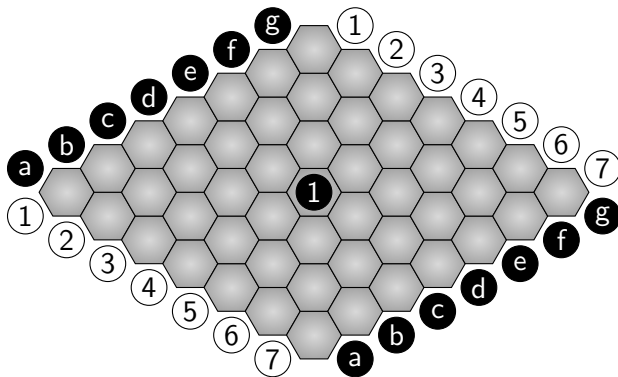
- ① Cutting and connecting games are hard
- ② Computer Go - Next AI Grand Challenge
- ③ Monte Carlo techniques promising
- ④ Tuning policies difficult
- ⑤ Solution: use evolution and self-play to learn better policies

# The Game of Hex - Good for AI



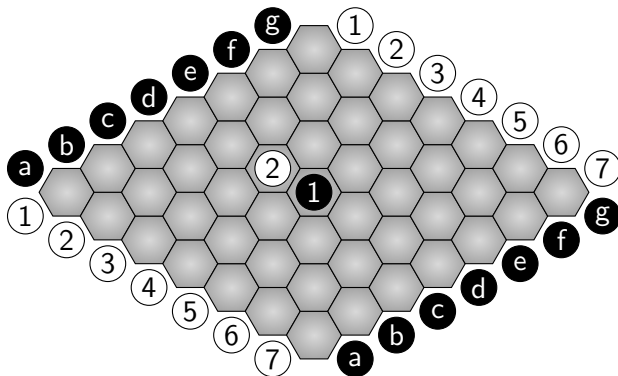
- 2 player, perfect information
- Easy to program
- Clear-cut winning condition
- Large problem space
- Solved for boards up to 7x7
- Common sizes: 11x11, 13x13

# The Game of Hex - Example Game



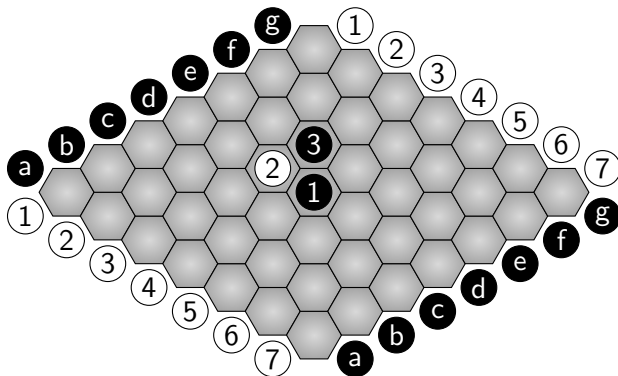
- Connect opposing sides
- Cut opponents connections

# The Game of Hex - Example Game



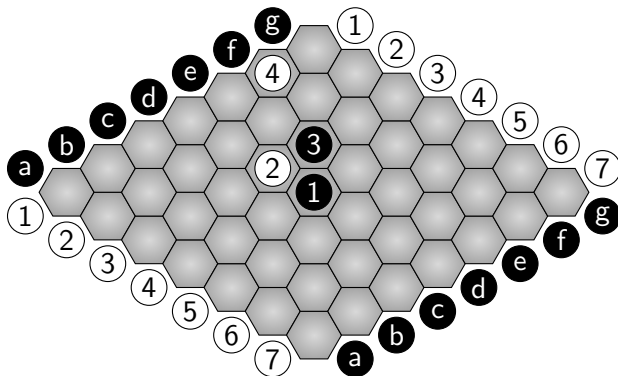
- Connect opposing sides
- Cut opponents connections

# The Game of Hex - Example Game



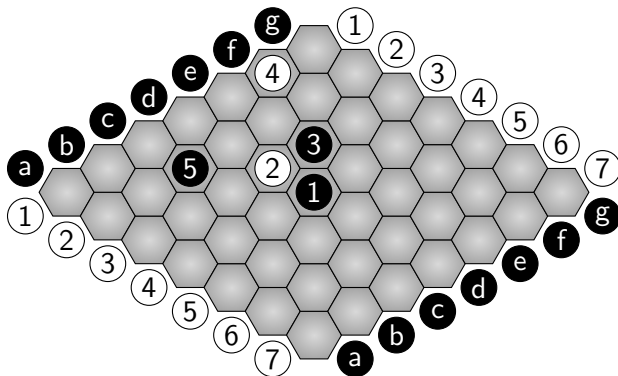
- Connect opposing sides
- Cut opponents connections

# The Game of Hex - Example Game



- Connect opposing sides
- Cut opponents connections

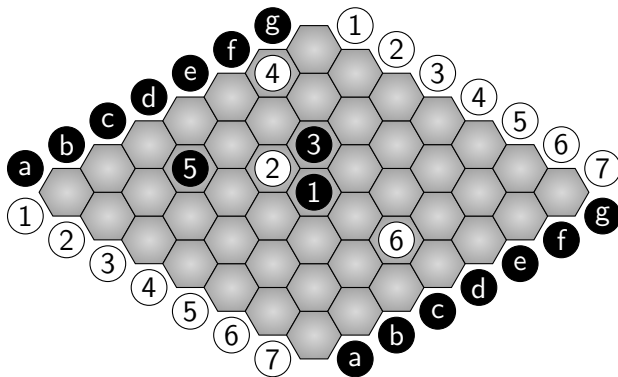
# The Game of Hex - Example Game



- Connect opposing sides
- Cut opponents connections

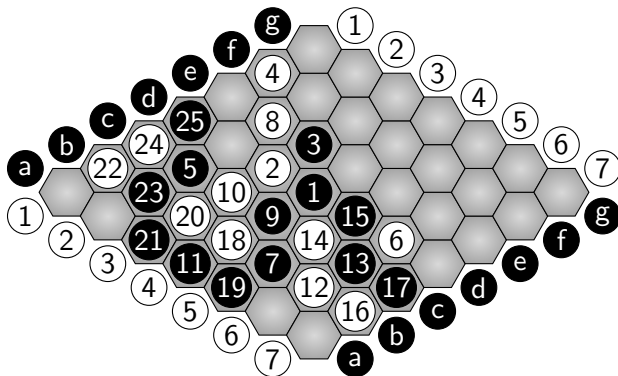


# The Game of Hex - Example Game



- Connect opposing sides
- Cut opponents connections

# The Game of Hex - Example Game



- Connect opposing sides
- Cut opponents connections

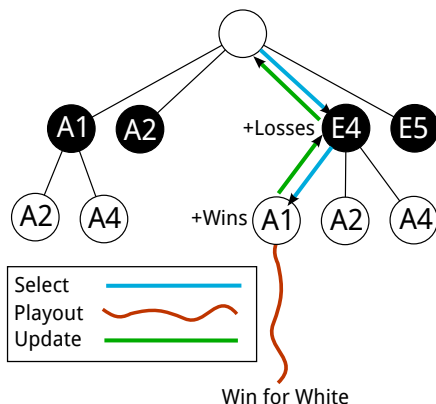
# Tree Search

- Game tree grows exponentially
  - ▶ Hex on a general board is PSPACE-complete
- No effective position evaluation heuristic
  - ▶ Previous champion until 2006, (*Six*, used circuit simulation
- Limited opportunities for provable pruning
  - ▶ Current champion, *MoHex*, does lots of this

# Monte Carlo Tree Search

- Grow tree dynamically
- Use random playouts to estimate minimax value
- In the limit, converges to true minimax value
- Simple idea, remarkably successful
- Computationally expensive

# Monte Carlo Tree Search - Overview

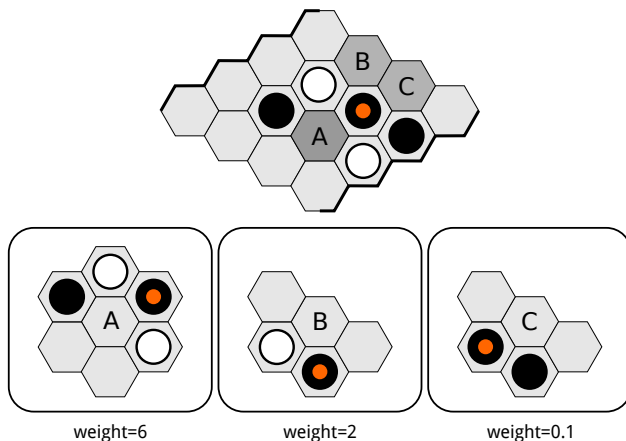


- *Select* - using win/loss statistics, traverse tree to a leaf
- **Playout** - from selected position, *randomly* play out game until final position
- *Update* - using results of playout, update win/loss statistics

# Monte Carlo Tree Search - Playout Policies

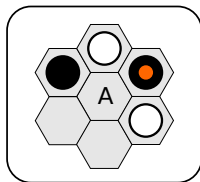
- Playout policy a critical part of overall performance
- We examine 4 different policies:
  - ▶ Default - uniform random
  - ▶ Uniform Local - play in local neighborhood first
  - ▶ Uniform Local with Tekuki (Play-Away) - possibly play in local neighborhood first
  - ▶ Local Pattern Weighted - stochastic, weighted local neighborhood

# Weighting the Playout Policy



- Use patterns to weight moves
- Weight moves local to last-played move
- If local area is filled, use default policy

## Example Encoding



## Encoding

Empty: 00

Black: 01

White: 10

Off: 11

vertex:	color	n/a	5	4	3	2	1	0
	1	...	00	00	10	01	10	01
bit:	31	12-30	10-11	8-9	6-7	4-5	2-3	0-1

`0b0000000000000000000000000000000010011001 = 102`

- Fast lookup table mapping pattern to weight
- Plenty of unused space in upper bits for storing extra info



# Monte Carlo Tree Search - Playout Policy

- Execution time is critical
- Requires deep insight to develop weights, then careful and expensive testing to verify improvement
- Naively “improving” the strength of the playout policy can hurt overall performance
  - ▶ Subtle interaction between selection, update, and playout
  - ▶ Example: fully deterministic policy equivalent to an evaluation function
- We want a technique to learn weights automatically

# Evolutionary Learning

- Idea: Evolve playout policy
- Policy = set of all possible local patterns + weights for those patterns
- Individual policies compete via self-play to propagate
- Best individuals are selected for reproduction
- Mutation/recombination operates on individual's pattern weights
- Individuals play *complete* games against each other using *complete* MCTS system

# Evolution

- Init
  - 1 Generate  $n$  individuals (population)
- Each Generation
  - 1 Evaluate fitness
  - 2 Rank by fitness
  - 3 Select top  $c$  individuals,  $c < n$
  - 4 Recombine individuals into  $n$  new children
  - 5 Mutate children

# Results of learning after training on 7x7)

Figure: Trained on 7x7 board

player	opponent		
	default	uniform local	uniform local (tenuki)
uniform local	70.50%		
uniform local (tenuki)	61.00%	50.00%	
learned	90.00%	84.00%	86.00%

- All-play-all tournament of the 4 policy variants.
- Each element is the percent win-rate of the row variant versus the column variant.
- 200 games per pair.

# Generalization to Different Board Sizes

	default		uniform local	
	11x11	13x13	11x11	13x13
learned	92.5 %	94 %	88.5%	85%

Figure: Trained on 7x7 board

	default	uniform local
learned	90.00%	84.00%

# Results versus MoHex

	MoHex on 7x7
default	11.75%
uniform local	26%
learned	42%

	MoHex on 11x11	Relative CPU
default	0.5%	100%
uniform local	0.0%	107%
learned	11%	122%

- MoHex, the current world-champion, uses lots of expert and domain-specific knowledge
- Good results on small board
- Dramatic improvement for modest overhead

# Conclusion

- Computer Hex/Go present a huge problem space
- MCTS is an effective technique to navigate that space
- Shown we can automatically learn to guide MCTS to play computer Hex better
- Future work - apply results to different games
- Source code available at  
<https://github.com/etherealmachine/hivemind>

# Conclusion

Thank you for the time  
Questions?