Joint work with
Raghuram Ramanujan and
Ashish Sabharwal

Upper Confidence bounds for Trees (UCT)

- *The **UCT algorithm** (Kocsis and Szepesvari, 2006), based on the UCB1 **multi-armed bandit algorithm** (Auer et al, 2002) has changed the landscape of game-playing programs in recent years.*
  - First program capable of master level play in 9x9 Go (Gelly and Silver, 2007)
  - UCT-based agent is a two-time winner of the AAAI General Game Playing Contest (Finnsson and Bjornsson, 2008)
  - Also successful in Kriegspiel (Ciancarini and Favini, 2009) and real-time strategy games (Balla and Fern, 2009)
  - Key to Google's AlphaGo success.

Understanding UCT

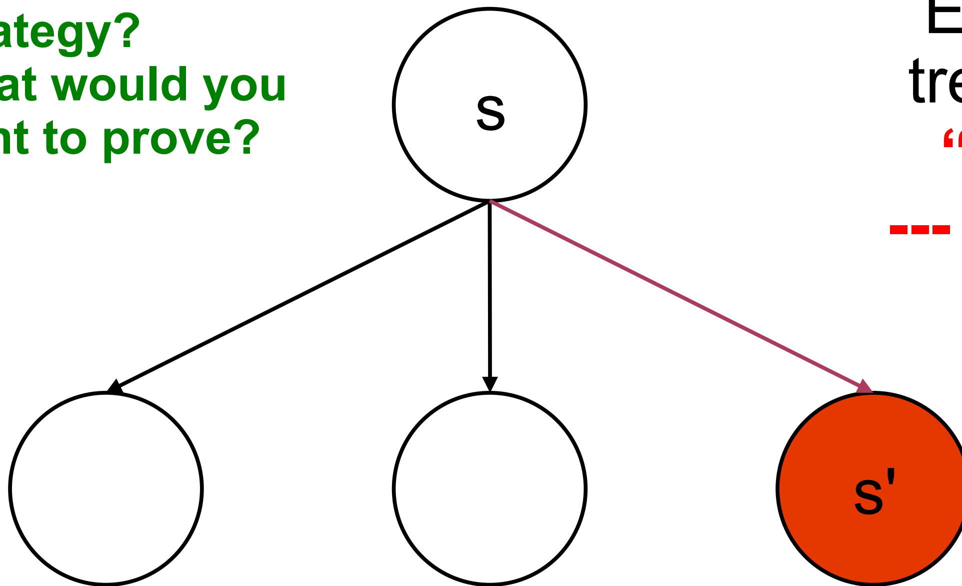
Despite its impressive track record, our current understanding of UCT is mostly anecdotal.

Our work focuses on gaining better insights into UCT by studying its behavior in search spaces where comparisons to Minimax search are feasible.

"Find the best slot machine"
--- while trying to make as much money as possible

The UCT Algorithm

Thm: Non-optimal arms are sampled at most log(n) times!



Every node in the search tree is treated like a multi-armed bandit.
"Find the best slot machine"
--- while trying to make as much money as possible

The ***UCB* score** is computed for each child and the best scoring child is chosen for expansion

Estimated utility = the average payout of arm so far

$$UCB(s') = Q(s') + c \cdot \sqrt{\frac{\log n(s)}{n(s')}}$$

Exploitation term
$Q(s')$ is the estimated utility of state s'

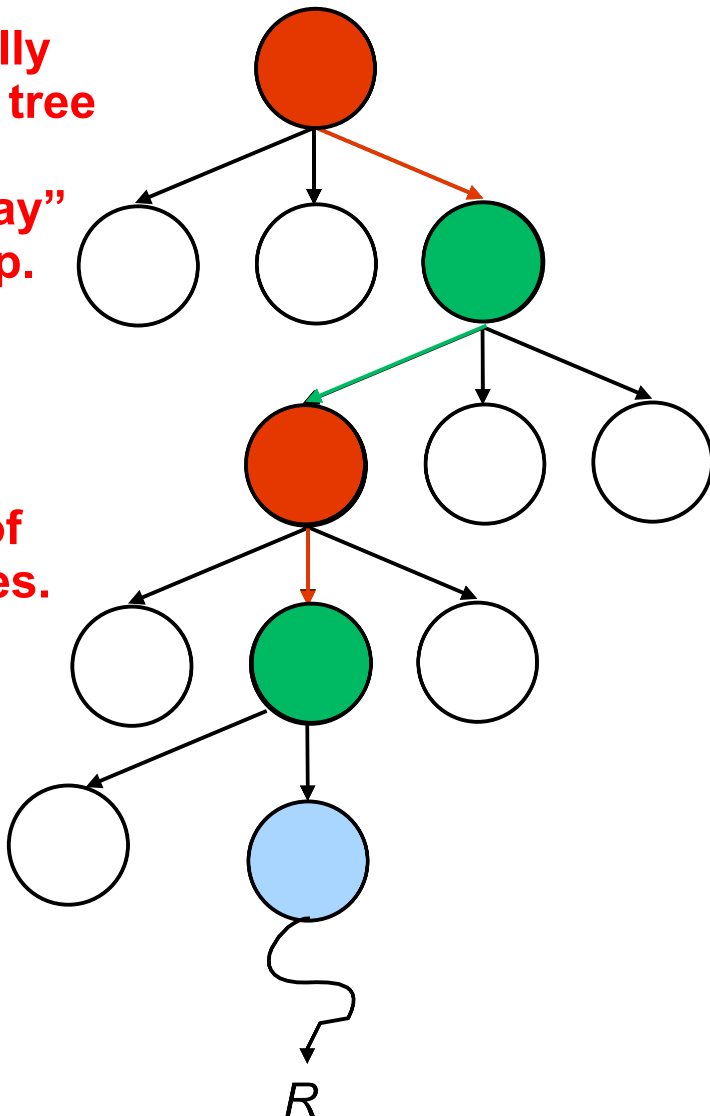
Exploration term
$n(s)$ is the number of visits to state s

The UCT Algorithm

Incrementally grow game tree by creating "lines of play" from the top.

&

Update estimates of board values.



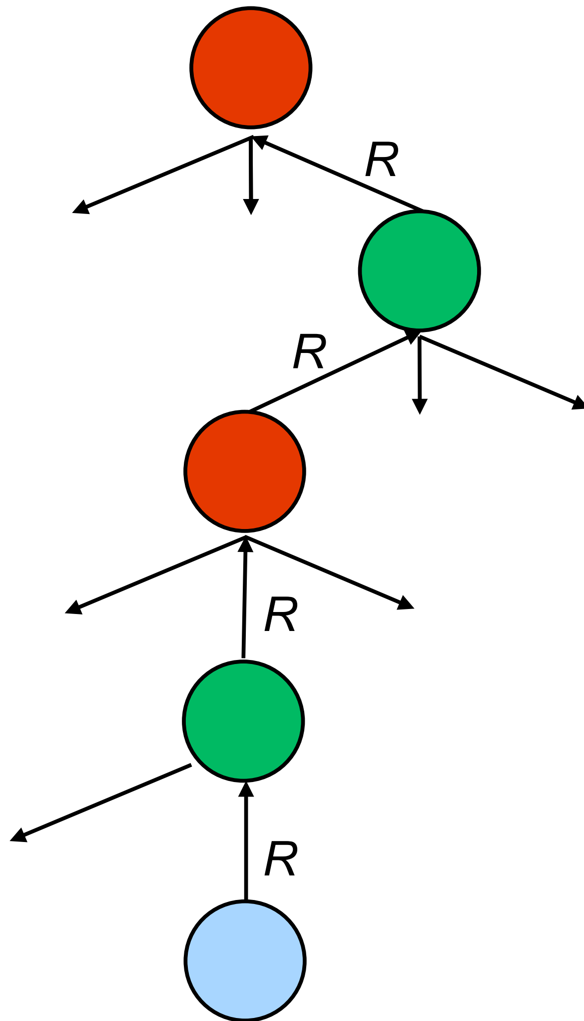
We descend the tree by starting at the root node and repeatedly selecting the **best scoring node**

At the opponent's nodes, a symmetric lower confidence bound is minimized to pick the best move

Node with unexplored child, a new child is created

A random playout is performed to estimate the utility *R* (+1,0, or -1) of this state

The UCT Algorithm



An averaging backup is used to update the value estimates of all nodes on the path from the root to the new node

$$n(s) \leftarrow n(s) + 1$$

Visit count update

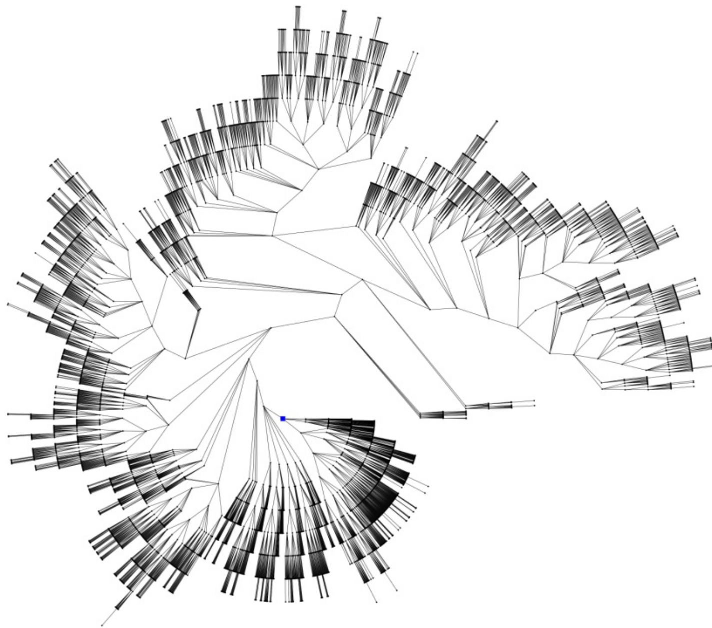
$$Q(s) \leftarrow Q(s) + \frac{R - Q(s)}{n(s)}$$

State utility update

Select each child node based on UCB scheme.

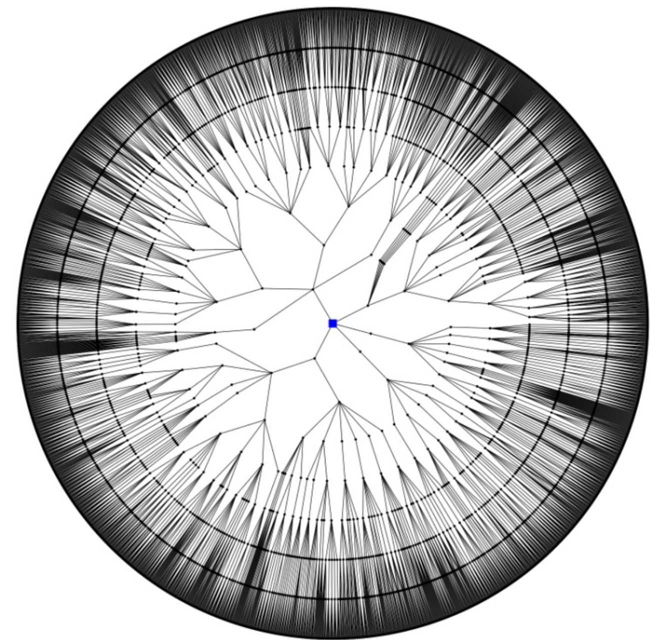
UCT versus Minimax

UCT Tree



- Asymmetric tree
- Best-performing method for Go

Minimax Tree



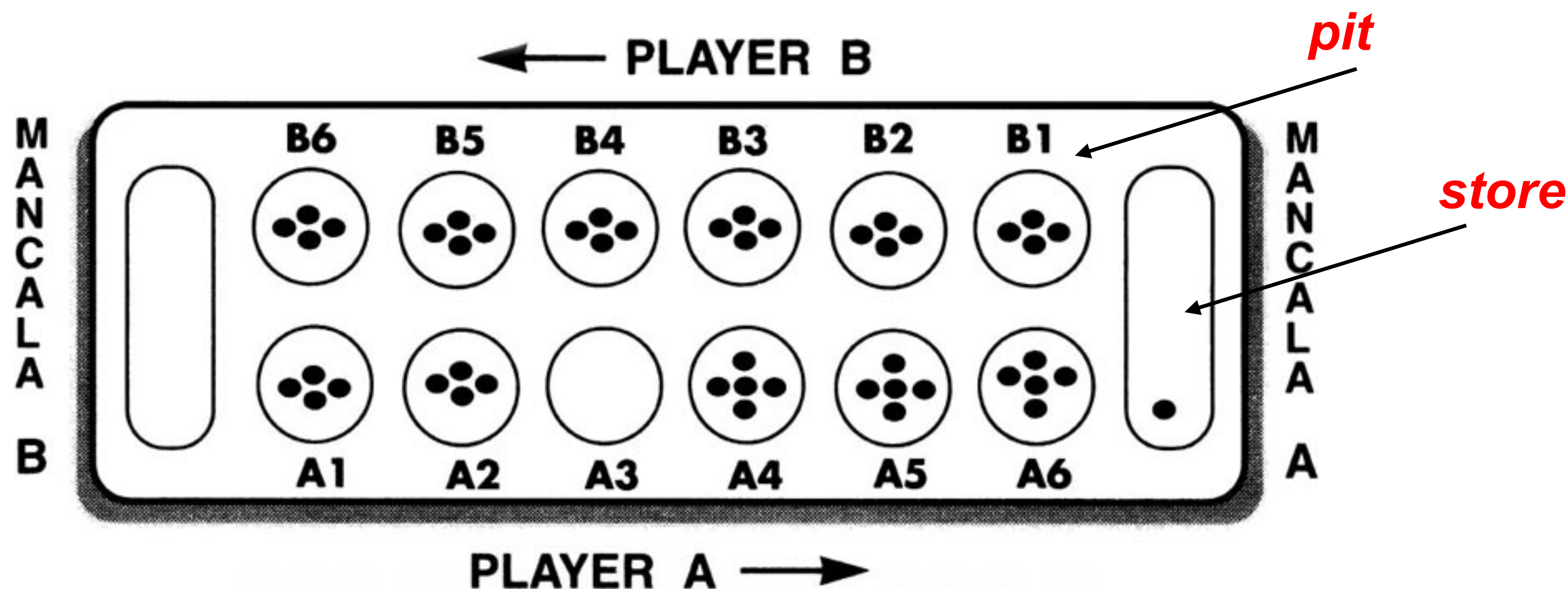
- Full-width tree up to some depth bound *k*
- Best-performing method for eg Chess

Search Spaces

- Minimax performs poorly in Go due to the large branching factor and lack of good board eval
- To understand UCT better, we need domain with another competitive method.
- Considered domains where Minimax search produces good play
  - **Chess**
    - A domain where Minimax yields world-class play but UCT performs poorly
  - **Mancala**
    - A domain where ***both*** Minimax and UCT yield competent players ‘out-of-the-box’
  - **Synthetic games (ongoing work)**

Mancala

Both **UCT and Minimax search produce good players.**



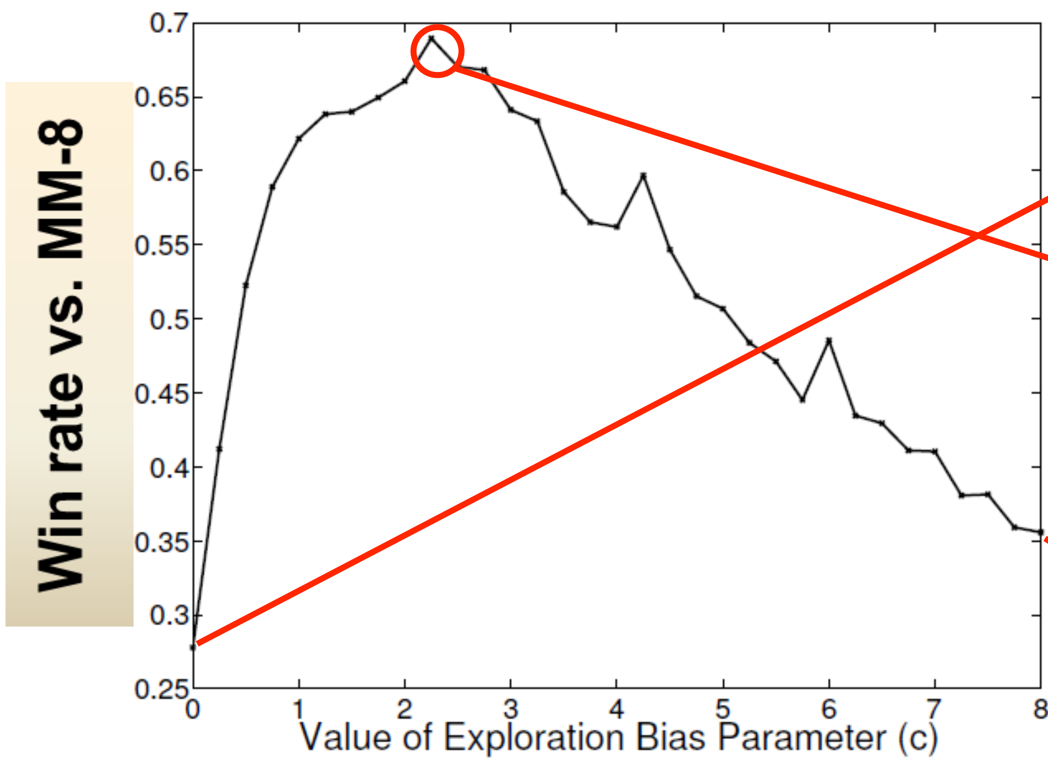
- A move consists of picking up all the stones from a pit and ‘sowing’ (distributing) them in a counter-clockwise fashion
- Stones placed in the stores are captured and taken out of circulation
- The game ends when one of the sides has no stones left
- Player with more captured stones at the end wins

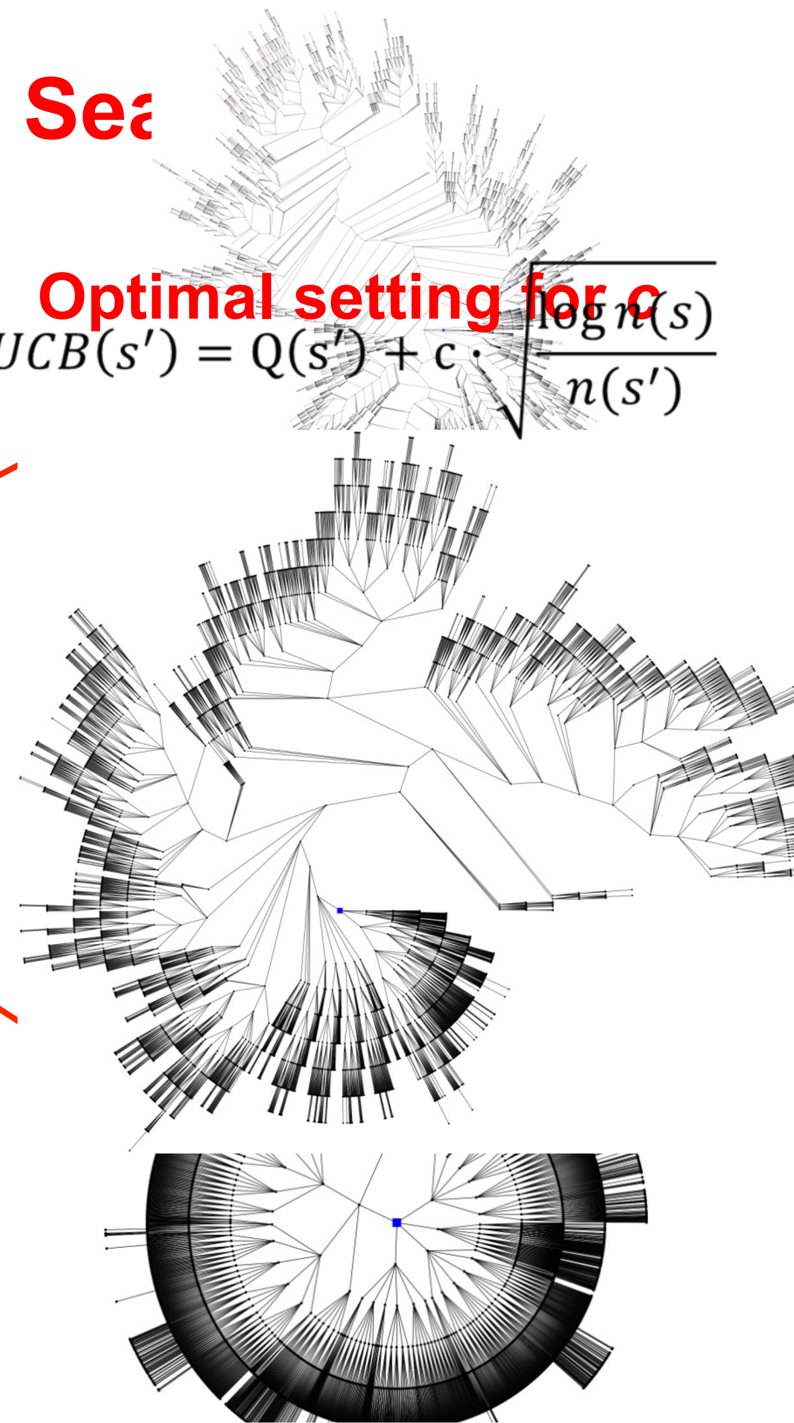
UCT on Mancala

- We examine the three key components of UCT
  1. ***Exploration vs. Exploitation***
  2. ***Random playouts***
  3. ***Info backup: Averaging versus minimax***

- Insights lead to improved UCT player
- We then deploy the winning UCT agent in a novel partial game setting to understand *how* it wins

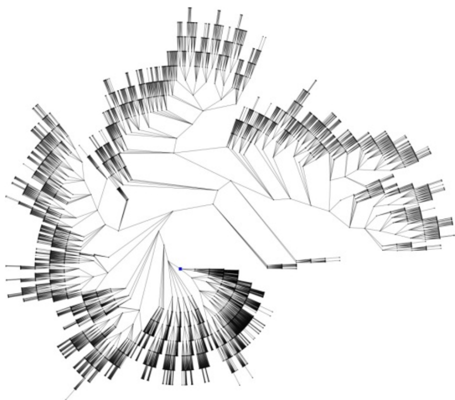
I) Full-Width versus Selective Sea

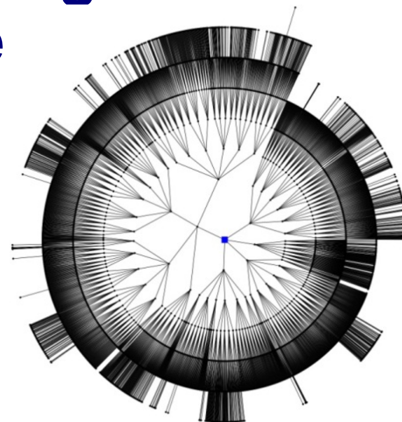




$$UCB(s') = Q(s') + c \cdot \sqrt{\frac{\log n(s)}{n(s')}}$$

Previous experience game tree search: "full width" + selective extensions outperforms "trying to be smart in node expansion."





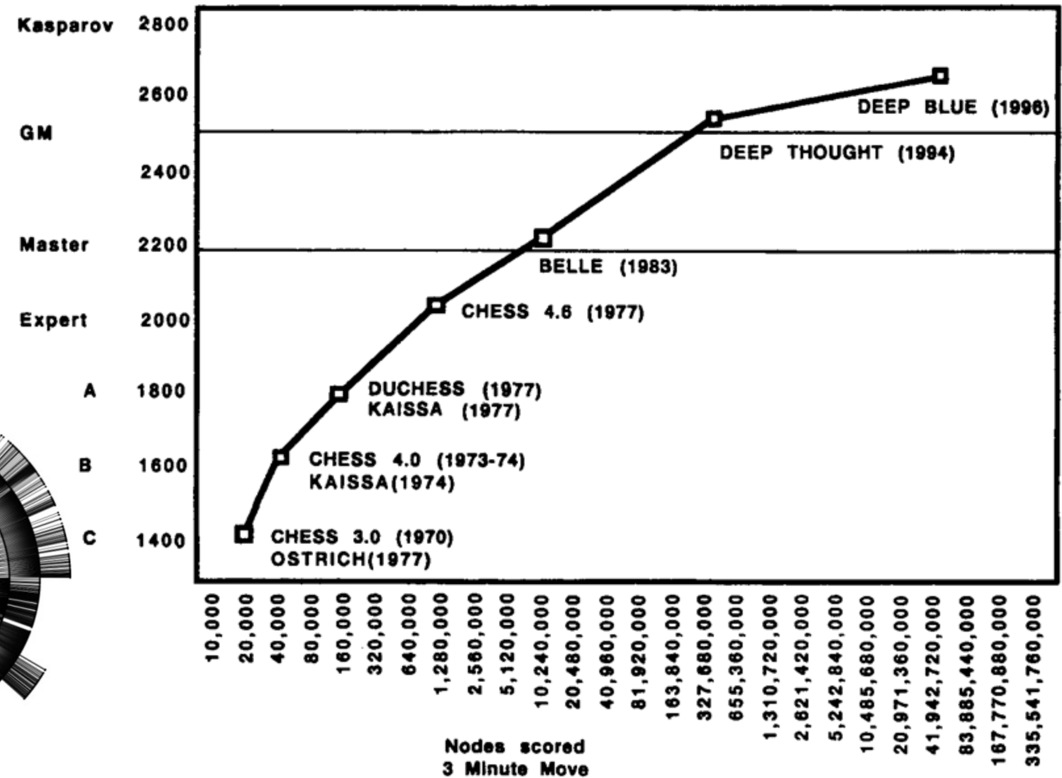


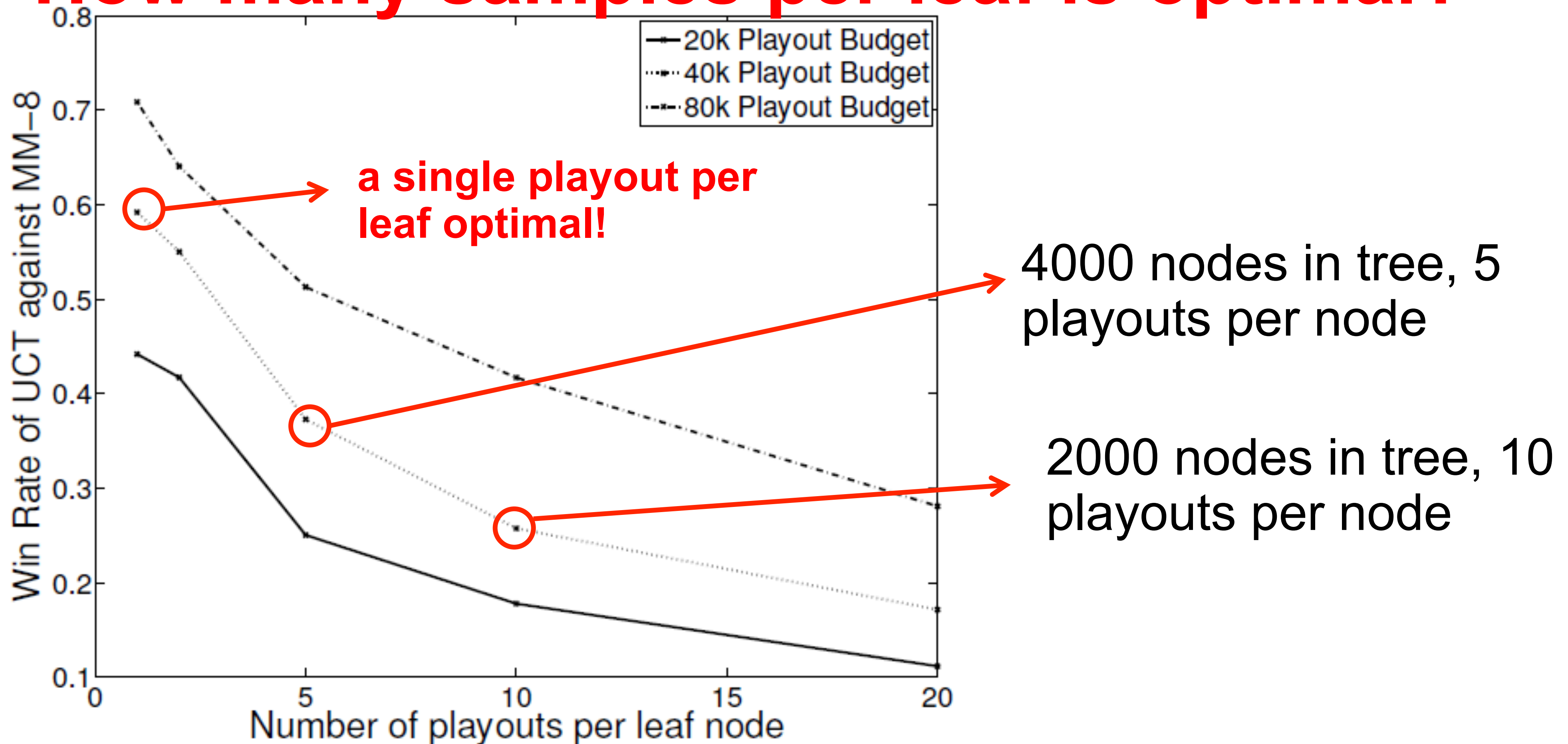
Figure 6.23. Relationship between the level of play by chess programs

- **Lesson:** *UCT challenges general notion that forward (i.e. unsafe) pruning is a bad idea in game tree search – not necessarily so! Smart exploration/ exploitation driven search can work.*

II) Random Playouts (Samples)

- During search tree construction, UCT estimates the value of a newly added node using a "random playout" (resulting in a +1, -1, or 0).
- Appeal: Very general. Don't need to construct a board/state evaluation function.
- But, random games look very different from actual games. Any real info in the result of random play?
- A: ***Yes, but it's a very "faint" signal. Other aspects of UCT compensate.***
- Also, how many playouts ("samples") per node? (surprise)

How many samples per leaf is optimal?



It is better to quickly examine many nodes than to get more random playout samples per node.

Given a fixed budget, should we examine fewer nodes more carefully or more nodes, with each node evaluated less accurately?

Lesson: ***There is a weak but useful signal in random game playout. *** Single *** sample is enough.***

III) Utility estimates backup: Averaging versus Minimax

- UCT uses an averaging strategy to back-up reward values. What about using a minimax backup strategy?
- For random playout information (i.e., a very noisy reward signal) averaging is best.
- Confirmed on Mancala and with synthetic game tree model.

- **Aside: Related to "infamous" game tree pathology: Deeper minimax searches can lead to worse performance! (Nau 1982; Pearl 1983).**

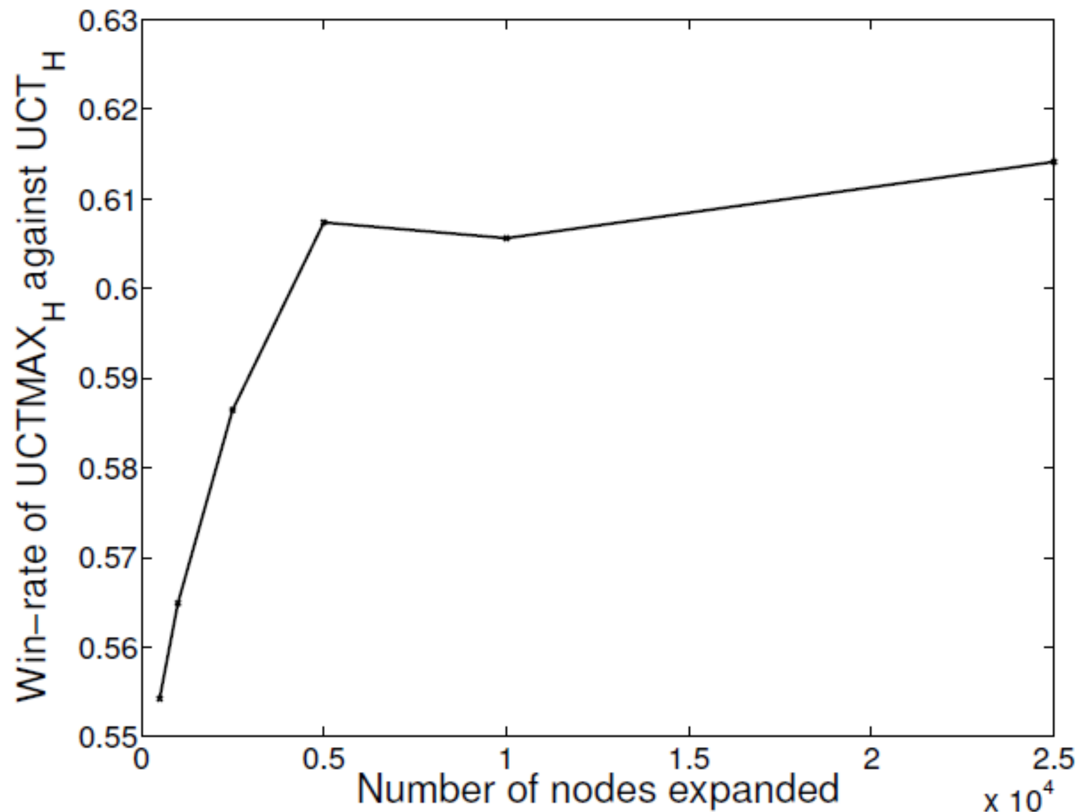
Phenomenon is related to very noisy heuristics --- such as random playout information.

Minimax uses heuristic estimates only from fringe of the explored tree --- ignoring any heuristic information from internal nodes.

UCT's averaging over all nodes (including internal ones) can compensate for pathology and near-pathology effects.

Improving over random playout info: Heuristic evaluation

- Unlike Go, we *do* have a good heuristic available for Mancala
  - Replace playouts with heuristics: UCT_H
- Heuristic much less noisy than random playout. Therefore, consider minimax backup.
  - We call it $UCTMAX_H$



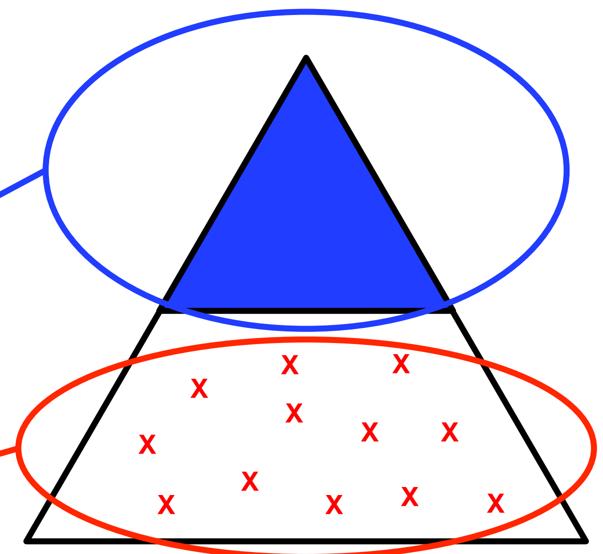
Allow both algorithms to build the same sized trees and compare their win-rates

Lesson: When a state evaluation heuristics is available, use minimax back-up in conjunction with UCT.

Final step: $UCTMAX_H$ versus Minimax

- We study the win-rate of $UCTMAX_H$ against Minimax players searching to various depths
  - Both agents use the same state heuristic

- We consider a hybrid strategy



Region with no search traps, play $UCTMAX_H$ versus Minimax

Region with search traps, play MM-16 versus MM-16

Mancala: $UCTMAX_H$ vs. Minimax (w. alpha-beta)

Minimax Look-ahead Depth	Win-rate of $UCTMAX_H$			
	6 Pits, 4 Stones per Pit		8 Pits, 6 Stones per Pit	
	Hybrid		Hybrid	
k = 6	0.76	0.72	0.71	0.68
k = 8	0.76	0.71	0.71	0.67
k = 10	0.75	0.65	0.69	0.65
k = 12	0.67	0.61	0.66	0.61

$UCTMAX_H$ hybrid outperforms+ MM

$UCTMAX_H$ outperforms MM

Note: exactly same number of nodes expanded.
Difference fully due to exploration / exploitation strategy of UCT

UCT recap:

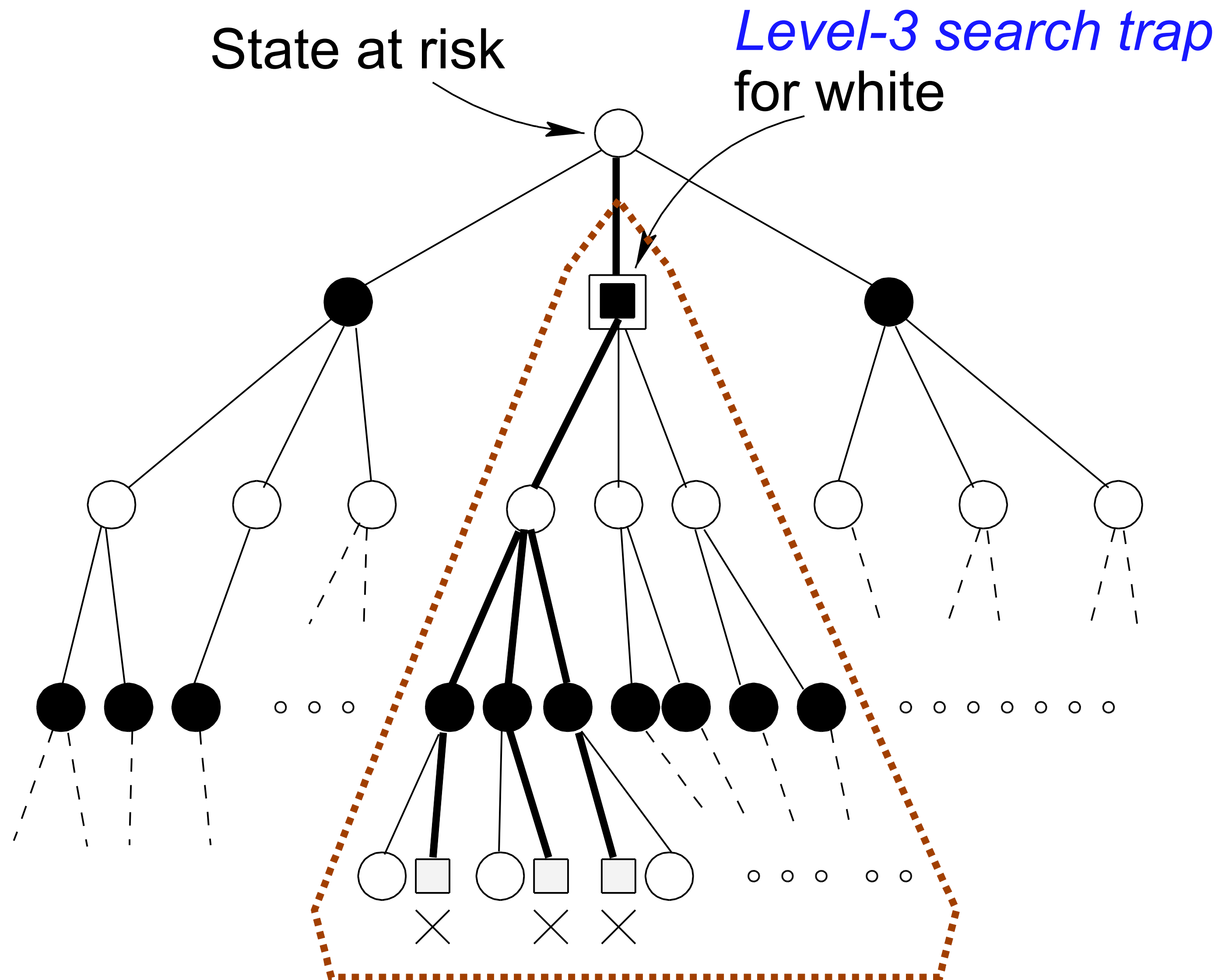
- First head-to-head comparison UCT vs. Minimax and first competitive domain.
- Multi-armed bandit inspired game tree exploration is effective.
- Random playouts (fully domain-independent) contain weak but useful information. Averaging backup of UCT is the right strategy.
- When domain information is available, use $UCTMAX_H$. Outperforms Minimax given exactly the same information and # node expansions.
- But, what about Chess?

Chess

What is it about the nature of the Chess search space that makes it so difficult for sampling-style algorithms, such as UCT?

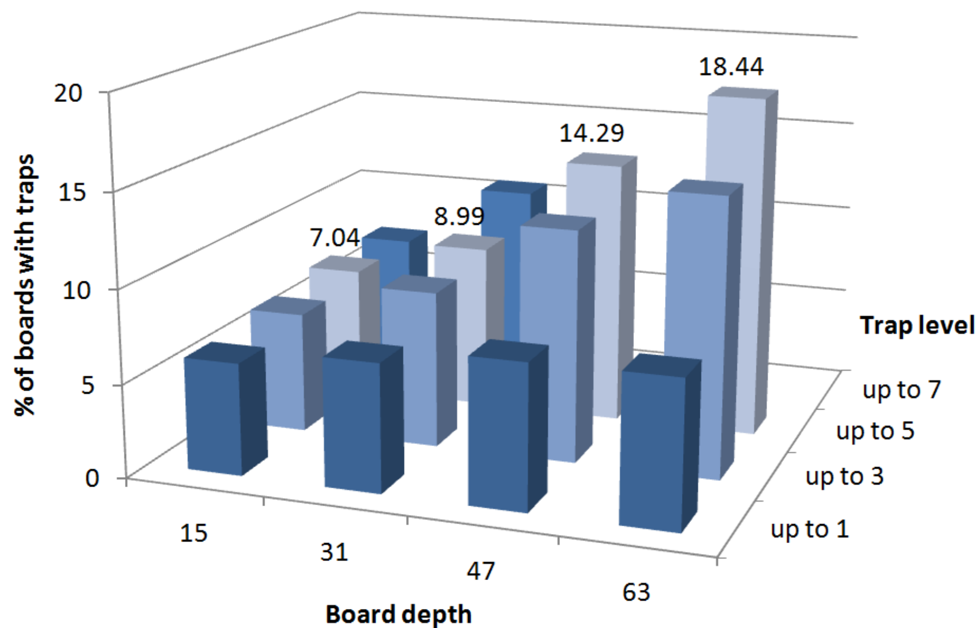
We will look at the role of (shallow) search traps

Search Traps

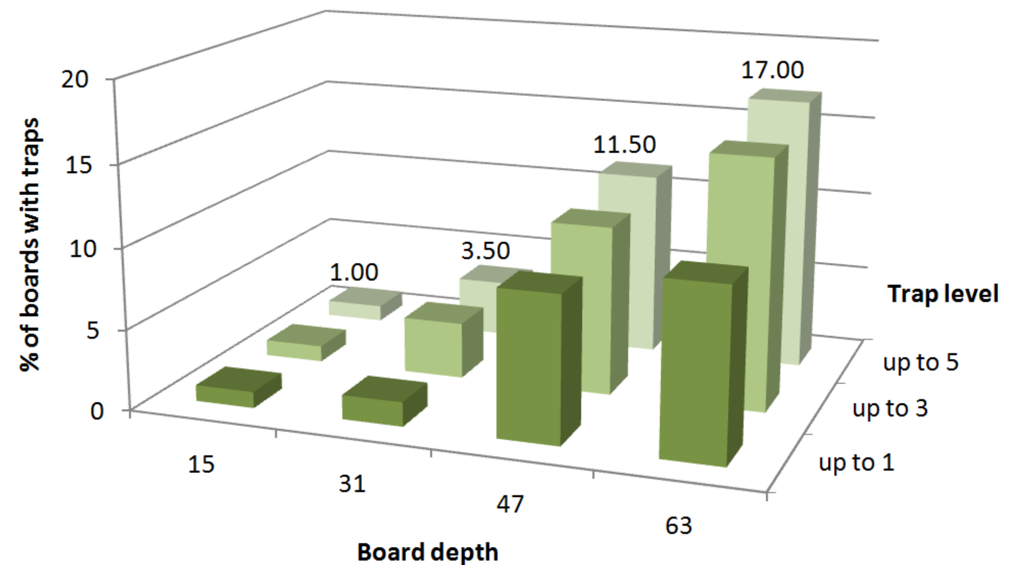


Search Traps in Chess

Random Chess Positions



Positions from GM Games



Search traps in Chess are sprinkled throughout the search space.

Identifying Traps

- Given the ubiquity of traps in Chess, how good is UCT at detecting them?
  - Run UCT search on states at risk
  - Examine the values to which the utilities of the children of the root node converge
  - Compare this to the recommendations of a deep Minimax search (the ‘gold-standard’)

Identifying Traps

Utility of move that UCT thinks is the best

Utility assigned by UCT to the move deemed best by Minimax

Utility assigned by UCT to the trap move. True utility: -1

	UCT-best	Minimax-best	Trap move
Level-1 trap	-0.083	-0.092	-0.250
Level-3 trap	+0.020	+0.013	-0.012
Level-5 trap	-0.056	-0.063	-0.066
Level-7 trap	+0.011	+0.009	+0.004

For very shallow traps, UCT has little trouble distinguishing between good and bad moves

With deeper traps, good and bad moves start looking the same to UCT. UCT will start making fatal mistakes.

UCT conclusions:

- Promising alternative to minimax for adversarial search.
- Highly domain-independent.
- Hybrid strategies needed when search traps are present.
- Currently exploring applications to generic reasoning using SAT and QBF.
- Also, applications in general optimization and constraint reasoning.
- **The exploration-exploitation strategy has potential in any branching search setting!**