Adversarial Reasoning: Sampling-Based Search with the UCT algorithm

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

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.

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

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

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 a 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 Search

Vary the value of the exploration constant $c$ and measure the win-rate of UCT against a standardized Minimax player.

Optimal setting for $c$

$$U_{CB}(s') = Q(s') + c \cdot \sqrt{\frac{\log n(s)}{n(s')}}$$
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.*

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

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

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

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: $\text{UCT}_H$
- Heuristic much less noisy than random playout. Therefore, consider minimax backup.
  - We call it $\text{UCTMAX}_H$
Lesson: When a state evaluation heuristics is available, use minimax back-up in conjunction with UCT.

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

<table>
<thead>
<tr>
<th>Minimax Look-ahead Depth</th>
<th>Win-rate of UCTMAX$_H$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>6 Pits, 4 Stones per Pit</td>
</tr>
<tr>
<td>Hybrid</td>
<td>0.76</td>
</tr>
<tr>
<td>k = 6</td>
<td>0.76</td>
</tr>
<tr>
<td>k = 8</td>
<td>0.76</td>
</tr>
<tr>
<td>k = 10</td>
<td>0.75</td>
</tr>
<tr>
<td>k = 12</td>
<td>0.67</td>
</tr>
</tbody>
</table>

**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

State at risk

Level-3 search trap for white
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

<table>
<thead>
<tr>
<th>Level</th>
<th>UCT-best</th>
<th>Minimax-best</th>
<th>Trap move</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level-1 trap</td>
<td>-0.083</td>
<td>-0.092</td>
<td>-0.250</td>
</tr>
<tr>
<td>Level-3 trap</td>
<td>+0.020</td>
<td>+0.013</td>
<td>-0.012</td>
</tr>
<tr>
<td>Level-5 trap</td>
<td>-0.056</td>
<td>-0.063</td>
<td>-0.066</td>
</tr>
<tr>
<td>Level-7 trap</td>
<td>+0.011</td>
<td>+0.009</td>
<td>+0.004</td>
</tr>
</tbody>
</table>

Utility assigned by UCT to the move deemed best by Minimax

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

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!