Trade-Offs in Sampling-Based Adversarial Planning

Raghuram Ramanujan and **Bart Selman**
Department of Computer Science
Cornell University
Ithaca NY 14853-7501, U.S.A.
{raghu,selman}@cs.cornell.edu

Abstract

The Upper Confidence bounds for Trees (UCT) algorithm has in recent years captured the attention of the planning and game-playing community due to its notable success in the game of Go. However, attempts to reproduce similar levels of performance in domains that are the forté of Minimax-style algorithms have been largely unsuccessful, making any comparative studies of the two hard. In this paper, we study UCT in the game of Mancala, which to our knowledge is the first domain where both search algorithms perform quite well with minimal enhancement. We focus on the three key components of the UCT algorithm in its purest form — targeted node expansion, state value estimation via playouts and averaging backups — and look at their contributions to the overall performance of the algorithm. We study the trade-offs involved in using alternate ways to perform these steps. Finally, we demonstrate a novel hybrid approach to enhancing UCT, that exploits its superior decision accuracy in regions of the search space with few terminal nodes.

Introduction

Minimax search with alpha-beta pruning has been one of the most effective strategies in game tree search for many decades. It's notable successes are in traditional complete information games, such as Chess and Checkers. The game of Go was the one significant exception, due to its large branching factor and the lack of a sufficiently effective board evaluation function. In 2007, the work on MoGo (Gelly and Silver 2007) demonstrated the surprising effectiveness of the UCT algorithm, an adversarial search technique based on a selective, sampling-based exploration of the search space. MoGo led to a significant interest in the potential of sampling-based tree search methods for adversarial reasoning. One key question is how UCT compares to traditional Minimax search with alpha-beta pruning and what aspects of UCT make it perform well and on what types of domains.

One difficulty in comparing UCT with Minimax search is that in most domains, either UCT is too ineffective to obtain reasonable play, such as in Chess, or Minimax is too weak to compete with UCT, as in Go. In this paper, we study a board game called Mancala, in which both UCT and Minimax search with alpha-beta pruning lead to a good level of play. We can therefore study the various components of the UCT approach and make a detailed comparison to the relatively well-understood Minimax player in this domain.

We examine the three key components of UCT search. In UCT, the game tree is explored using a strategy motivated by algorithms for the multi-armed bandit problem. The search strategy involves a trade-off between exploration and exploitation. Exploitation leads to following certain lines of play deep into the search tree, while exploration leads to a more full-width style search down to a uniform depth level. We will show that there is a well-defined optimal setting for the amount of exploration versus exploitation. Moreover, the tree explored by UCT is quite different from that examined by traditional full-width Minimax search, and, for the same number of nodes explored, UCT provides better move decisions.

Instead of a board evaluation heuristic, UCT typically relies on so-called playouts to determine the quality of a leaf node. In a playout, each player moves randomly (or based on a basic but fast move strategy), leading to an eventual win, loss, or draw, which is used as a utility estimate for the leaf node. Even though random or pseudo-random playouts proceed very differently than actual games, we will see that there is real information in the playout signal. We also study the trade-off in terms of making more playouts per leaf node versus exploring more nodes in the search space. Somewhat surprisingly, we will show that the most effective UCT strategy uses a single playout while exploring as many nodes as possible. This is consistent with observations in the domain of Go.

Although the playouts contain real information, having even a basic board evaluation function is often much more effective. In particular, when a reasonable board evaluation function is known, Minimax search using the heuristic will strongly dominate UCT with playout information. This suggests that one should incorporate the board evaluation function into UCT, replacing the playout information. We show that such an approach does indeed boost the performance of UCT.

A third component of UCT concerns the way in which it propagates information from the playouts (or heuristics) at the leaf nodes back up to the root of the search tree. In the original UCT formulation, the information is averaged over multiple lines of play. This strategy is quite effec-

tive for random playout information. However, as we will demonstrate, when a board evaluation heuristic is available, it is much better to propagate information backwards using a Minimax style update rule. We refer to our final player as UCTMAX$_H$, i.e., a UCT player that incorporates a board evaluation function and uses Minimaxing to backup information. We will show that this player outperforms both a Minimax style player with alpha-beta pruning and the original UCT algorithm (with playouts and averaging). So, a careful study of the various trade-offs in the UCT framework can lead to the design of a superior adversarial search strategy.

Background

The standard approach to planning in adversarial domains involves performing a Minimax search (typically combined with alpha-beta pruning) to some fixed depth cutoff, at which point a static evaluation function is used to estimate the strength of the states reached. This technique was used to great effect in the high-profile match-up between Deep Blue and Gary Kasparov (Campbell, Hoane, and Hsu 2002), in which a computer defeated the reigning human Chess champion for the first time.

More recently, Monte-Carlo tree search methods such as UCT have caught the eye of the research community, due to their impressive performance in domains that could not be tamed by Minimax-style approaches. The interest in UCT was spurred by the initial success of MoGo, the first program to achieve master level play in 9x9 Go (Gelly and Silver 2007; 2008). Since then, UCT has been successfully applied to other domains such as Kriegspiel (Ciancarini and Favini 2009), real-time tactical assault planning (Balla and Fern 2009) and general game-playing (Finnsson and Björnsson 2008).

Upper Confidence bounds applied to Trees (UCT)

The UCT algorithm (Kocsis and Szepesvári 2006) builds on UCB1 (Auer, Cesa-Bianchi, and Fischer 2002), an algorithm designed to optimally trade-off exploration and exploitation in multi-armed bandit problems. UCT builds a search tree from a given state by iterating over the following steps:

- **Selection:** At a state s, the algorithm selects an action a that maximizes an upper confidence bound on the utility of the action value according to:

$$\pi(s) = \operatorname*{argmax}_a \left(Q(T(s,a)) + c \cdot \sqrt{\frac{\log n(s)}{n(T(s,a))}} \right)$$

  Here, $T(s,a)$ is the transition function that returns the state reached by taking action a in state s, $Q(s)$ is the current estimate of the utility of state s and $n(s)$ is the number of previous visits to state s. If $n(T(s,a)) = 0$ for an action a, then it is selected before any other actions are re-sampled. The constant c is tuned empirically to strike a good balance between exploring under-sampled moves and exploiting known good moves. Note that at states where the opposing player is on move, the action that *minimizes* a symmetric lower confidence bound is picked, i.e., the selection operator is:

$$\pi'(s) = \operatorname*{argmin}_a \left(Q(T(s,a)) - c \cdot \sqrt{\frac{\log n(s)}{n(T(s,a))}} \right)$$

- **Estimation:** Starting at the root node, the selection procedure outlined above is repeatedly applied until either a previously unvisited node, or a terminal node, is reached. If the state is non-terminal, a state evaluation heuristic is applied to obtain a valuation R of this state's utility (in the case of terminal nodes, this is a value from the set $\{-1, 0, +1\}$ based on the outcome of the game). This heuristic may either take the form of random playouts, or a static position evaluator if one is available. The newly-visited node is then added to the seach tree. Under this scheme, the size of the tree grows by one node every iteration.
- **Value Back-up:** The reward R from the current UCT episode is used to update the utility estimate of each state s on the path from the leaf node to the root as follows:

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

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

  This update assigns to each state the average reward accrued from every episode that passed through it.

The above steps are repeated until search time expires, at which point the action leading to the state with the highest mean utility is executed.

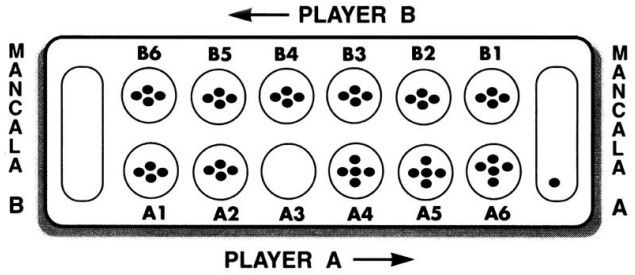

Figure 1: A sample Mancala game state

The Game of Mancala

Mancala is a deterministic, 2-player game that is thousands of years old and popular in many parts of the world. It is played on a rectangular board like the one depicted in figure 1. Initially, the six pits on each side of the board contain 4 stones each (though variants with different numbers of pits and stones exist). The two larger pits at the ends (termed the "stores") hold any stones that the players capture.

A move consists of a player picking up all the stones from a pit on his side and placing them one at a time in each of the following pits, in a counter-clockwise order. The player's own store is included in this "sowing" process, but the opponent's store is skipped. A stone that lands in a store is

deemed captured and is permanently removed from circulation. The game is not strictly turn-taking — depending on where the last stone is placed, one of three things may happen:

1. If the last stone lands in the player's own store, then he goes again.
2. If the last stone lands in an empty pit on the player's own side, then a capture occurs — the single stone is immediately moved to the player's store, as are any stones in the pit directly across from the empty pit (on the opponent's side).
3. Otherwise, the turn ends.

Figure 1 shows the state of the game after player A has made the first move of the game, A3. After this move, player A would go again, since his last stone landed in his store. The game ends when either side has no legal moves left. At this point, the player with the emptied pits captures any stones on the opponent's side of the board. The winner is the player with the greater number of stones in his store at the game's end. Mancala has previously received some attention from the AI community. Most notably, the game has been weakly solved for a variety of starting configurations (Irving, Donkers, and Uiterwijk 2000).

A pertinent question at this point is: *Why Mancala?* Minimax reigns supreme in games with strong tactical components and many "trap states", such as Chess, and attempts to reach similar levels of performance in these domains using UCT have failed (Ramanujan, Sabharwal, and Selman 2010a; 2010b). UCT's notable successes so far have been in domains where Minimax-based approaches have fallen short. These are typically games with large branching factors such as Go or Hex (Arneson, Hayward, and Henderson 2010) and/or for which good heuristics are unavailable. One exception here is the game Lines of Action, a domain previously dominated by Minimax, where UCT with a significant amount of knowledge engineering is now competitive (Winands and Björnsson 2009). But by and large, the set of domains where Minimax and UCT both produce competent players is disjoint. To our knowledge, Mancala is the first domain where UCT and Minimax *both* produce players that are competitive with nearly *no enhancements* to the basic algorithms. This latter feature is particularly important since it allows us to study these two algorithms side-by-side in their purest form.

In the remainder of this paper, we will take a closer look at the three steps that comprise an iteration of UCT — targeted expansion of nodes in the search tree, the use of random playouts to evaluate leaf node positions and the propagation of information from the leaves to the internal nodes through averaging. For each step, we will look at how it contributes to UCT's decision making process and consider the tradeoffs involved in applying alternate approaches. In the final section, we present a partial-game setting that highlights a major strength of UCT-style approaches — *the ability to make good decisions in regions of the search space that have few terminal nodes.* This offers a potential mechanism for explaining UCT's success in Go, while laying the groundwork for future hybrid approaches to adversarial planning that combine the strengths of UCT and Minimax.

Experimental Methodology

In the sections that follow, we present data that averages the outcome of thousands of Mancala games between different search algorithms. Here, we outline some of the methodology that was used to run and score those games.

Since Mancala is deterministic, we need a way to introduce variance between individual games. We accomplish this by generating random initial configurations as follows: starting with an empty board, we place a stone uniformly at random in one of the pits (excluding the stores), until there are no stones left. This may however generate positions that are heavily biased in favor of one player. To combat this, we play duplicate games. Given a starting position, two games are played, with the players' sides swapped after the first game. Player A is deemed to have won if he beats player B's result while playing from one side, while at least matching player B's result when the sides are switched. Otherwise, we disregard the entire board. For example, if player A wins the first game, and draws the second, he is the winner on that board; if player A were to lose the second game, then the board would be deemed too skewed to be useful. We report the win-rates for the different algorithms — this is simply the ratio of the number of duplicate games won to the total number of non-skewed boards that were used.

The other important variable we control for in comparisons between a Minimax agent and a UCT agent is the amount of search effort. We measure this in terms of the number nodes expanded by the algorithm. In all comparisons involving a UCT player and a Minimax player, the former is allowed to expand at most the number of nodes its opponent would expand at *the given position in the game*, unless stated otherwise. This is important given the wide variation in the branching factor of the tree as the game progresses. Also, our Minimax player always uses alpha-beta pruning.

Full-Width Search vs Selective Search

In many applications, it has been observed that the value of the exploration bias parameter c in the selection step of UCT has a large role in the overall performance of the algorithm (for example, see Lorentz 2008). We concretely illustrate this phenomenon in Mancala. Figure 2 depicts the nodes expanded by a depth 8 Minimax player (MM-8) on a typical Mancala position. The graph layout, which was generated using `GraphViz` (Gansner and North 2000), places nodes at the same depth from the root (denoted by the blue square) at the same radius. The board evaluation heuristic applied at the leaf nodes is the difference of the stone counts in the stores of the two players. This simple heuristic is nevertheless quite effective and difficult to improve upon with additional features such as mobility (Gifford et al. 2008).

We now consider the search trees that are created by a variant of UCT that uses the same leaf evaluation heuristic, rather than random playouts (henceforth, UCT$_H$). Figure 3 shows the trees that are built by UCT$_H$ for different values of c, for the same board that was used in figure 2. With $c = 0$,

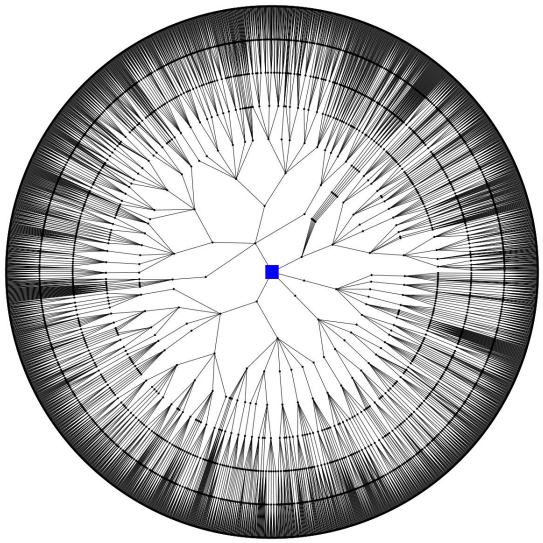

Figure 2: Search tree expanded by MM-8 with alpha-beta pruning

the tree reaches some very deep positions, but is sparse at any given level. With too much exploration ($c = 20$), the tree becomes more regular and "Minimax-like". At $c = 2.5$, the best-performing value for this domain, we obtain a tree that strikes a balance between the two extremes; it is dense at shallow depths, but more sparse and focused at deeper levels.

This has important implications for the performance of the UCT player. In figure 4, we plot the win rate of UCT_H against a fixed Minimax opponent (MM-8), while varying the value of c. Each data-point is the average of several hundred games.

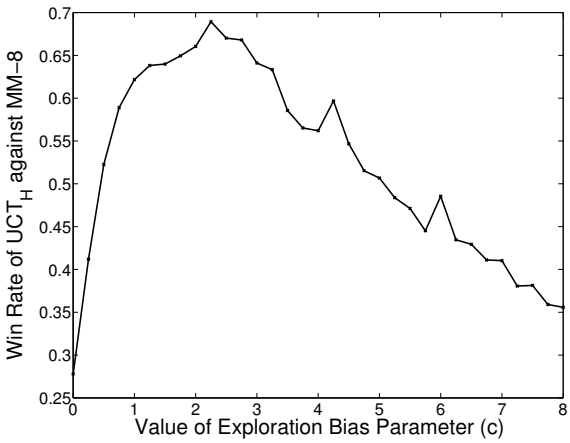

Figure 4: Win rate of UCT against an MM-8 opponent, while varying the exploration bias parameter (c)

The trend is unmistakable — for small values of c, UCT misses too many moves and does poorly. With too large a c, UCT is not sufficiently focused and the action utilities do not converge quickly enough to produce competent play. There is a clear optimal setting for the value of c. This establishes that Mancala is a challenging domain for a planning agent, and any algorithm that tries to balance exploration-exploitation must do so carefully. Furthermore, as we will show in later sections, a variant of UCT that builds trees similar to that shown in the middle panel of figure 3 outperforms Minimax search in this domain. This is striking — traditional wisdom in the game-playing community is that search methods that employ forward pruning techniques generally perform much worse than their full-width search counterparts (Biermann 1978; Truscott 1981). Yet, here we have a domain where an algorithm that performs unsafe pruning outperforms a full-width search.

Playouts vs Nodes Expanded

In its original incarnation, the UCT algorithm uses a random playout to estimate the utility of leaf nodes in the search tree. This is particularly useful in games where good heuristics to statically evaluate positions are not known, since playouts offer a domain-independent solution to this problem. While any number of playouts may be performed from a node to estimate its utility, typically only a single playout is performed due to run-time considerations. In this section, we address the question: *Given a fixed budget, what is the best way to allocate the playouts?* Should we expand fewer nodes in the search tree while evaluating each one carefully, or should we look at more nodes without being overly worried about the accuracy of their evaluation?

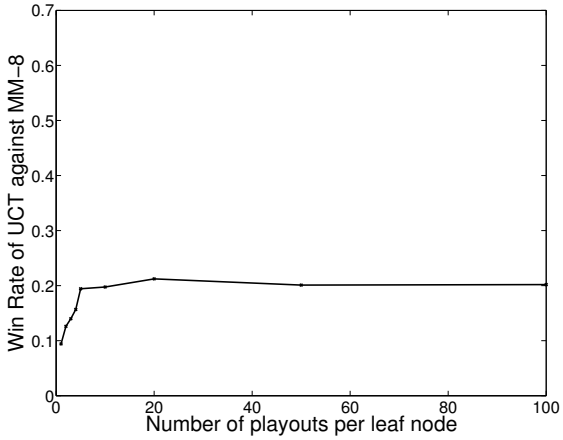

Figure 5: Win Rate of UCT against MM-8, while varying the size of the UCT search tree

In our first experiment, we play a UCT agent using random playouts against a fixed opponent (an MM-8 search using the hand-engineered heuristic) in a series of games. We vary the number of random playouts that are averaged to produce the leaf utility estimates for UCT, holding all other parameters constant. The results are presented in Figure 5.

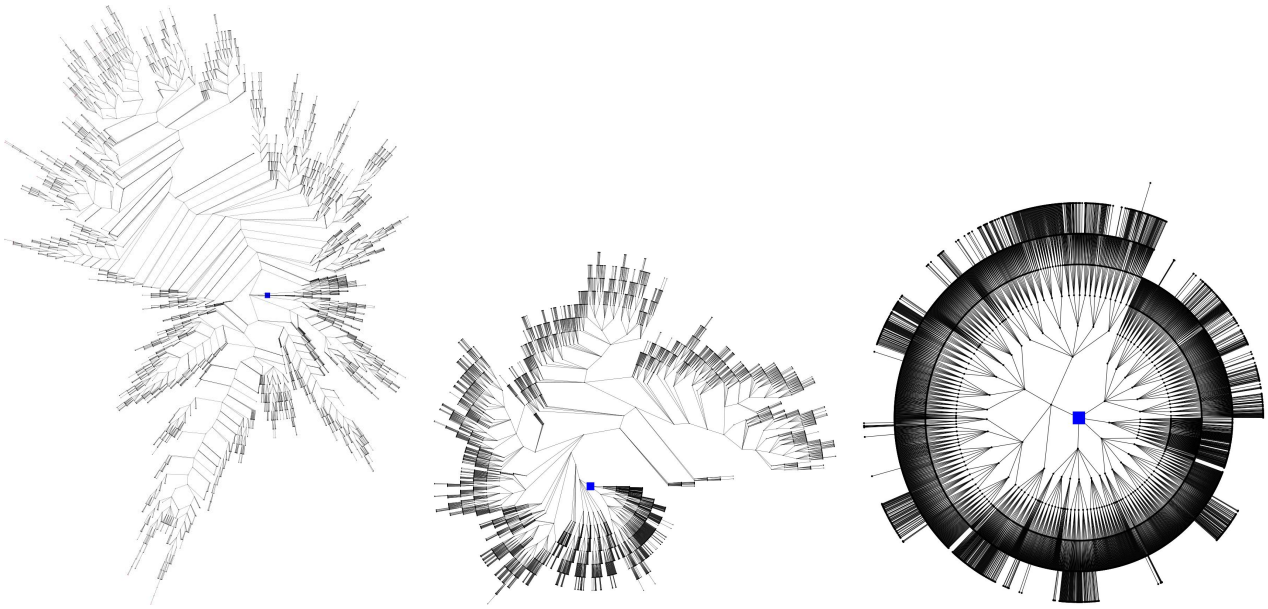

Figure 3: From left to right: search trees expanded by UCT$_H$ with $c = 0$, $c = 2.5$ and $c = 20$ respectively

In a second experiment, we fix the number of random playouts at 1, while varying the number of iterations for which we run UCT. The results of this experiment are shown in Figure 6.

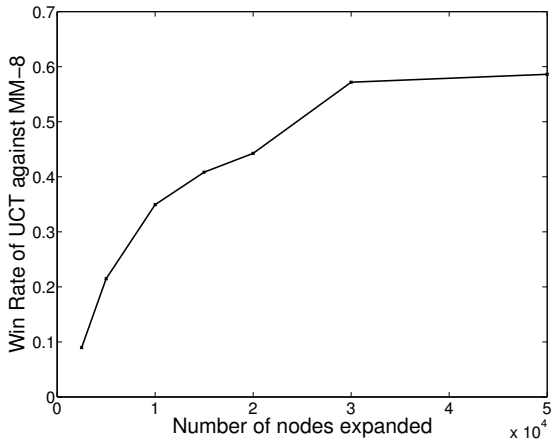

Figure 6: Win Rate of UCT against MM-8, while varying the size of the UCT search tree

The two plots taken together highlight an interesting phenomenon. Increasing the number of playouts per leaf only has a modest impact on the playing strength of the agent (quickly leveling off), while increasing the number of UCT iterations (i.e. nodes expanded) yields a much more substantial improvement. This suggests that given a fixed computational budget (in a domain where heuristics are unavailable), it is far better to run more iterations of UCT with fewer playouts per leaf, than to run fewer iterations with more playouts. This is illustrated concretely in figure 7: against MM-8, we give UCT a fixed total playout budget and vary the number of random playouts per leaf. For example, with 2 playouts per leaf, we run UCT for $10,000$ iterations, with 4 playouts per leaf, we run UCT for 5000 iterations and so on. Note that in this experiment, we are not concerned with normalizing the number of nodes expanded by the two algorithms. We are only interested in the impact that different allocations of the playout budget has on the playing strength of UCT, against a fixed opponent.

These findings are consistent with past observations that random playouts do contain some information but the quality of their feedback is limited (Ramanujan, Sabharwal, and Selman 2010b). Thus, increasing the number of playouts per leaf only helps to some extent. Why does UCT perform better with fewer playouts, but more iterations? The answer lies in the fact that as UCT builds up its search tree, it begins to "freeze" into a principal variation in the higher reaches of the tree, which introduces a progressive bias in subsequent random playouts. Put another way, as the search advances, random playouts are carried out starting from nodes deeper in the tree which increases their predictive accuracy.

Averaging vs Minimaxing

UCT's tree-building and information-propagation steps are interleaved. The building process is guided by a Minimaxing descent through the existing tree, that determines which section of the tree will be sampled next. The new information discovered on the current iteration is propagated up by an averaging operator, which informs the future growth of the tree. While the Minimaxing descent is justified by theoretical results for bandit-algorithms (Auer, Cesa-Bianchi, and Fischer 2002), it is not clear that averaging is the best way to propagate information up the tree. In the limit, the

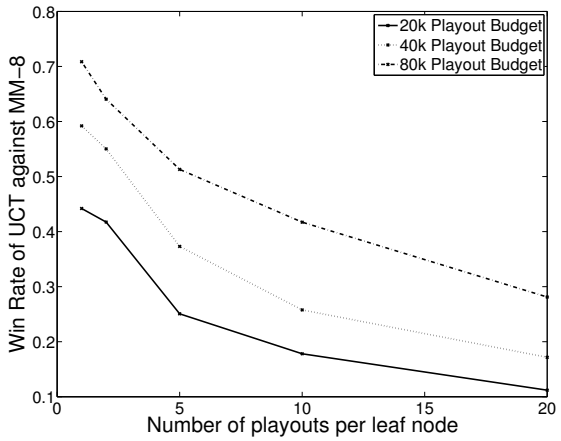

Figure 7: Win Rate of UCT against MM-8, with fixed total playout budget, while varying number of playouts per leaf node

action utilities computed by UCT are guaranteed to converge to the true Minimax values (Kocsis and Szepesvári 2006), but the time to convergence may be super-exponential (Coquelin and Munos 2007).

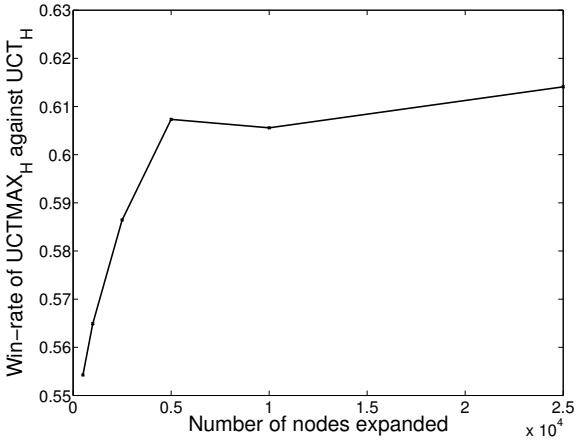

Figure 8: Win Rate of UCTMAX$_H$ against UCT$_H$ on complete Mancala games vs the number of iterations

Coulom empirically showed that when using random playouts with UCT, averaging is the superior back-up operator when a node has been visited few times, whereas Minimaxing is better when node visit counts are higher (Coulom 2006). Our results confirm that when using playouts, this phenomenon occurs in Mancala as well. However, while good heuristics are not known for Go, we *do* have a heuristic available for our domain. Thus, in Mancala, even a single visit to a node may yield a reasonably good estimate of its utility. Given this, can Minimaxing do a better job as a back-up operator, than the standard averaging approach?

In our first experiment, we play games between a UCT$_H$ agent and a UCT agent that uses Minimax back-ups as defined below with the same heuristic (henceforth, UCTMAX$_H$).

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

$$Q(s) \leftarrow \begin{cases} n(s) \cdot \max\limits_{s' \in succ(s)} Q(s') & \text{if } s \text{ is a maximizing node} \\ n(s) \cdot \min\limits_{s' \in succ(s)} Q(s') & \text{if } s \text{ is a minimizing node} \end{cases}$$

Figure 8 shows the win rate of UCTMAX$_H$, as we vary the number of iterations that both players are given (all other parameters are fixed to the same values for both players). Note that even with just 500 iterations (nodes expanded), UCTMAX$_H$ is already on-par with UCT$_H$. This suggests that in domains where good heuristics are available, UCT can be significantly boosted by replacing the averaging back-up operator with Minimaxing *even on small trees*. Increasing the number of iterations increases the performance gap between the two approaches.

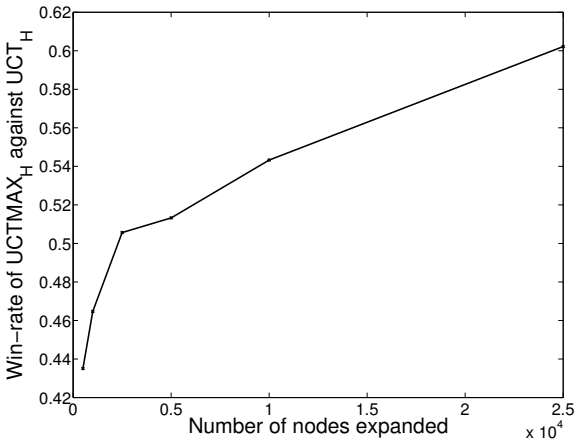

Figure 9: Win Rate of UCTMAX$_H$ against UCT$_H$ on partial Mancala games vs the number of iterations

One possible cause for why UCTMAX$_H$ outperforms UCT$_H$ could be that the former propagates information from terminal nodes much more efficiently than the latter. To test this hypothesis, we ran "partial"-games between UCTMAX$_H$ and UCT$_H$ as follows: for the first 14 plys of a game, we play UCTMAX$_H$ against UCT$_H$, and play two MM-16 players against each other thereafter. This lets us evaluate the relative strengths of the two algorithms in parts of the search space with few terminal nodes. Figure 9 plots the win rate of UCTMAX$_H$ in this set-up, as we vary the number of iterations. Once again, we observe an upward trend as the number of iterations is increased. With limited compute time, UCTMAX$_H$ makes decisions that are about as good (or marginally worse) than UCT$_H$ in regions of the search space where terminal nodes are absent. However, this is compensated by UCTMAX$_H$'s ability to efficiently handle terminal nodes when they do appear (as evidenced by its higher win rate in complete games, with the same parameter settings). With more compute time, UCTMAX$_H$ makes better decisions, even in regions that have few terminal nodes.

Hybrid Strategies

In the previous section, we introduced the idea of a partial game so we could isolate the effects of the visibility of terminal nodes on the overall strength of a game-playing algorithm. Here we return to that idea.

In domains like Chess, "search-traps" or early terminal nodes appear at all levels of the search tree and present problems for sampling-based approaches such as UCT (Ramanujan, Sabharwal, and Selman 2010a). Minimax, being a full-width search, does not miss such trap states, so long as they occur within the horizon of the search (including search extensions). In Go, trap states do not occur until the later stages of the game. A possible reason why UCT has been such a success in Go could be that it is a fundamentally better decision-making algorithm than Minimax, so long as the search space it is exploring *does not contain too many terminal positions*. If this is indeed the case, it is plausible that UCT builds up a sufficient positional advantage in the early stages of the game (where methods like Minimax have poor visibility) that its opponents cannot overcome later on. Minimax handles terminal nodes in an optimal fashion which we cannot expect to improve upon — so the question is whether UCT can outperform Minimax when no terminal positions occur within the search horizon.

To answer this, we ran partial games pitting UCTMAX$_H$ against Minimax in which the two algorithms are played against each other, until the ratio of terminal to non-terminal nodes in the search tree expanded by the Minimax player first exceeds 0.003. Thereafter, the game is completed by two MM-16 players. We do this for a range of Minimax opponents with different lookahead depths. We also repeated the experiment on a larger Mancala board, with 8 pits and 6 stones per pit (denoted $(8,6)$). The results are presented in figures 10 and 11. For comparison, we also include win-rate data for UCTMAX$_H$ on complete games between the two players.

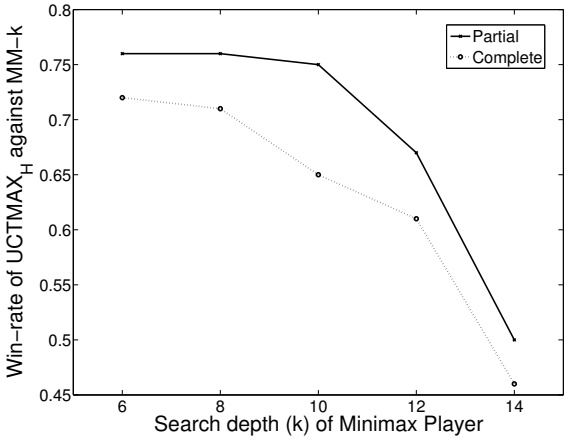

Figure 10: Win-rate of UCTMAX$_H$ against MM-k on partial and complete games of size $(6,4)$

The key observation here is that, in both board sizes, and independent of the lookahead depth of the Minimax player, UCTMAX$_H$ always does better in the hybrid games than the complete games. We have found that this trend holds (though not as pronounced) even when the standard UCT implementation, with averaging back-ups, is used. This supports our hypothesis that UCT is a better predictor of good moves in regions of the search space in which terminal nodes are largely absent.

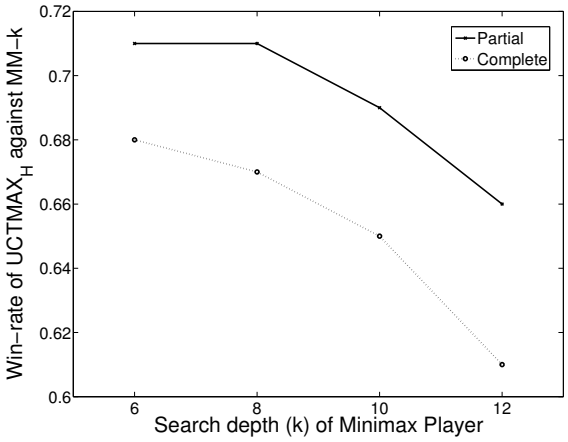

Figure 11: Win-rate of UCTMAX$_H$ against MM-k on partial and complete games of size $(8,6)$

Another trend that is apparent from figures 10 and 11 is that the win-rate of UCTMAX$_H$ declines in the partial games with increasing k. There are two reasons for this. As we increase k, the terminal node ratio threshold of 0.003 is reached faster by the Minimax player. For example, in $(6,4)$-size games between UCTMAX$_H$ and MM-6, the switch-over to the MM-16 playout phase occurs 21 plys into the game on average. In contrast, against MM-14, the threshold is exceeded after only 8 plys, which hardly gives UCT sufficient opportunity to steer the game decisively in it's favor. Secondly, even when terminal nodes are not directly visible, they make their presence felt more strongly to deeper searching players by influencing the heuristic estimates of nearby positions. This effect is diluted when the depth of the lookahead is smaller.

Related Work

It is generally believed that searching deeper is beneficial in adversarial planning domains; indeed, advances in algorithms and hardware that enabled one to search deeper within the same time constraints were one of the key reasons behind the success of Deep Blue. However, this phenomenon is not universally true. Notably, there are families of so-called *pathological* games where increased look-ahead leads to worse decision-making (Beal 1980; Nau 1982). One reason that is cited to explain the lack of pathological behavior in real games is the presence of early terminal nodes in the search tree that can be evaluated perfectly (Pearl 1983).

Given the comparatively poor performance of Minimax in regions of the Mancala search space with low terminal node density, it is natural to wonder if this indicates the presence of look-ahead pathology in the initial stages of the game. We have found that this is not the case. In particular, in partial games between MM-k and MM-$(k+n)$ players ($n \geq 2$), we have found that the deeper searching agent always plays significantly better. This trend persists for a wide range of values of k, n and board sizes.

Intriguingly, recent work by Nau et al. suggests that closely related variants of Mancala *are* pathological (Nau et al. 2010). In their work, some changes are made to the rules to make the game tree more regular. Also, the game is terminated after 8 plys and the player with the stone advantage at that point is declared the winner. The heuristic value of a node is defined to be it's true backed-up minimax value that has been corrupted with Gaussian noise. In experiments, they found that an MM-1 search produces higher quality decisions than an MM-5 search. While we were able to reproduce their results, we also discovered that the pathological behavior was quite brittle. In particular, if the heuristic values were made more accurate at nodes deeper in the tree (by reducing the variance of the Gaussian noise), the pathology disappeared. Nevertheless, it suggests that localized near-pathological behavior may occur even in real games.

Conclusions

This work was a study of the UCT algorithm in the game of Mancala, a domain where Minimax-style search methods have traditionally performed well. We demonstrated that UCT, with a minimal amount of knowledge engineering can be competetive in this domain. To our knowledge, this is the first domain where this has been possible. We carried out a dissection of the algorithm and showed that: (a) selective search *can* outperform full-width tree searches in some adversarial domains, (b) given a fixed computational budget, it is more prudent to build larger trees with just a few random playouts per leaf node, and (c) when a reasonable heuristic function is known for a domain, using a Minimaxing backup operator can offer a significant performance boost over the traditional averaging operator. Finally, in partial game settings, we demonstrated that the quality of decisions made by UCTMAX$_H$, a variant of standard UCT, can be superior to those of Minimax search in regions of the search space where terminal nodes are rare. This offers up the intriguing possibility of hybrid adversarial search algorithms that can exploit the strengths of both UCT and Minimax to produce stronger game-playing agents.

Acknowledgements

This work was supported by NSF Expeditions in Computing award for Computational Sustainability, 0832782; NSF IIS grant 0514429; and IISI, Cornell Univ. (AFOSR grant FA9550-04-1-0151). We would like to thank Nikhil Gupta, Sajeev Krishnan and Ashish Sabharwal for stimulating discussions and useful insights about Mancala and UCT.

References

Arneson, B.; Hayward, R. B.; and Henderson, P. 2010. Monte Carlo tree search in hex. *Computational Intelligence and AI in Games, IEEE Transactions on* 2(4):251–258.

Auer, P.; Cesa-Bianchi, N.; and Fischer, P. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine Learning* 47(2-3):235–256.

Balla, R.-K., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *Proceedings of the 21st international joint conference on Artifical intelligence*, 40–45. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Beal, D. 1980. An analysis of minimax. In M.R.B., C., ed., *Advances in Computer Chess 2*, 103–109. Edinburgh University Press.

Biermann, A. W. 1978. Theoretical issues related to computer game playing programs. *Personal Computing* 86–88.

Campbell, M.; Hoane, Jr., A. J.; and Hsu, F.-h. 2002. Deep Blue. *Artif. Intell.* 134:57–83.

Ciancarini, P., and Favini, G. P. 2009. Monte Carlo tree search techniques in the game of Kriegspiel. In *IJCAI-09*.

Coquelin, P.-A., and Munos, R. 2007. Bandit algorithms for tree search. *CoRR* abs/cs/0703062.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, 72–83.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI-08*, 259–264. AAAI Press.

Gansner, E. R., and North, S. C. 2000. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience* 30(11):1203–1233.

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *24th ICML*, 273–280.

Gelly, S., and Silver, D. 2008. Achieving master level play in 9×9 computer Go. In *23rd AAAI*, 1537–1540.

Gifford, C.; Bley, J.; Ajayi, D.; and Thompson, Z. 2008. Searching and game playing: An artificial intelligence approach to Mancala. Technical Report ITTC-FY2009-TR-03050-03, Information Telecommunication and Technology Center, University of Kansas, Lawrence, KS.

Irving, G.; Donkers, J.; and Uiterwijk, J. 2000. Solving Kalah. *ICGA Journal* 23(3):139–148.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *17th ECML*, volume 4212 of *LNCS*, 282–293.

Lorentz, R. J. 2008. Amazons discover Monte-Carlo. In *Computers and Games*, 13–24.

Nau, D. S.; Luštrek, M.; Parker, A.; Bratko, I.; and Gams, M. 2010. When is it better not to look ahead? *Artif. Intell.* 174:1323–1338.

Nau, D. S. 1982. An investigation of the causes of pathology in games. *Artif. Intell.* 19:257–278.

Pearl, J. 1983. On the nature of pathology in game searching. *Artif. Intell.* 20(4):427–453.

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010a. On adversarial search spaces and sampling-based planning. In *20th ICAPS*, 242–245.

Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010b. Understanding sampling style adversarial search methods. In *26th UAI*.

Truscott, T. R. 1981. Techniques used in minimax game-playing programs. Master's thesis, Duke University, Durham, NC.

Winands, M. H. M., and Björnsson, Y. 2009. Evaluation function based Monte-Carlo LOA. In *ACG*, 33–44.