Ensemble Monte-Carlo Planning: An Empirical Study

Alan Fern and **Paul Lewis**
School of EECS
Oregon State University
afern@eecs.oregonstate.edu, paularthurlewis@gmail.com

Abstract

Monte-Carlo planning algorithms, such as UCT, select actions at each decision epoch by intelligently expanding a single search tree given the available time and then selecting the best root action. Recent work has provided evidence that it can be advantageous to instead construct an ensemble of search trees and to make a decision according to a weighted vote. However, these prior investigations have only considered the application domains of Go and Solitaire and were limited in the scope of ensemble configurations considered. In this paper, we conduct a more exhaustive empirical study of ensemble Monte-Carlo planning using the UCT algorithm in a set of six additional domains. In particular, we evaluate the advantages of a broad set of ensemble configurations in terms of space and time efficiency in both parallel and single-core models. Our results demonstrate that ensembles are an effective way to improve performance per unit time given a parallel time model and performance per unit space in a single-core model. However, contrary to prior isolated observations, we did not find significant evidence that ensembles improve performance per unit time in a single-core model.

1 Introduction

UCT is a Monte-Carlo planning algorithm (Kocsis and Szepesvari 2006) that extends recent algorithms for multi-armed bandit problems to sequential decision problems including Markov Decision Processes and games. Most notably, UCT has served as the basis for some of the premiere algorithms for computer Go (Gelly and Silver 2007) and has also shown success in a variety of other domains (Balla and Fern 2009; Finnsson and Bjornsson 2008; Bjarnason, Fern, and Tadepalli 2009; Lang and Toussaint 2010).

At each decision epoch, UCT selects an action by first using a sequence of Monte-Carlo simulations to construct a look-ahead tree and then selecting the root action that looks best. Recently, several investigations (see Section 2) have observed improved performance by constructing an ensemble of multiple trees and letting them vote in order to select an action. This approach resembles ensemble methods from the area of machine learning, such as bagging (Breiman 1996) and tree randomization (Breiman 2001), for which thorough evaluations have shown broad, significant benefits. In contrast, there is not yet a thorough evaluation of the ensemble approach for Monte-Carlo planning algorithms such as UCT. Rather, the existing work has been limited to only two domains, Go (Chaslot, Winands, and van den Herik 2008) and Solitaire (Bjarnason, Fern, and Tadepalli 2009), and a very small number of ensemble configurations. As such, there is not a good understanding of this approach in terms of the types of advantages and their magnitude and breadth.

The main contribution of this paper is to conduct a broader evaluation of the ensemble approach, including a greater diversity of ensemble configurations and test domains. We consider a matrix of ensemble configurations that varies the ensemble size (number of trees) and the size of each tree in the ensemble. The configurations are evaluated in each of a diverse set of six domains including: Biniax, Backgammon, Connect 4, Havannah, and two versions of Yahtzee. Based on these results we consider the advantages of ensembles when used in both parallel and single-core models. Our main observations include: 1) In a parallel model, ensembles almost always improve performance given a fixed amount of time per decision, 2) In a single-core model, ensembles typically improve performance given a fixed memory limit, 3) In a single-core model, there is no clear evidence that ensembles improve performance given a fixed time bound and sometimes can decrease performance.

In what follows, we first review related work in Section 2. Next, we describe UCT and the ensemble approach in Section 3. Section 4 describes our evaluation domains and is followed by our evaluation and analysis of results in Section 5.

2 Related Work

Our work is inspired by recent investigations that provide isolated results for Monte-Carlo ensembles. First, there have been several attempts to design and evaluate parallel variants of UCT (Cazenave and Jouandeau 2007; Gelly et al. 2008; Chaslot, Winands, and van den Herik 2008). These approaches involve different ways of conducting Monte-Carlo simulations in parallel in order to arrive at approximately the same result as running UCT on a single processor, but much faster. A variety of approaches were evaluated in the game of Go, including ones involving complex message passing and synchronization. However, a surprising result was that perhaps the simplest "baseline" method, named root par-

allelization (Chaslot, Winands, and van den Herik 2008), was the dominant approach among those investigated. The root parallelization method simply runs multiple versions of UCT in parallel to construct an ensemble of trees and then aggregates the information contained in their root nodes at the end, thereby requiring minimal overhead for synchronization. That is, root parallelization is a simple ensemble approach where the ensemble members are each run on a dedicated processor. Even more surprising was that in some cases it was observed that the root parallelization approach achieved speedups even when evaluated in a single processor model. That is, given a fixed amount of time, better performance could be achieved by constructing ensemble members in sequence on a single processor and voting, compared to using that same amount of time to construct a single larger tree. While compelling, these studies focused exclusively on the game of Go.

In other recent work, motivated by planning in the domain of Klondike solitaire, UCT was combined with the idea of hindsight optimization, an idea that was then generalized to arrive at the Ensemble UCT approach (Bjarnason, Fern, and Tadepalli 2009). This approach constructs multiple smaller trees on a single processor, instead of one large tree, and selects an action according to a weighted vote among the trees. Some advantages of the ensemble approach were noted, including achieving better performance in a fixed amount of time. The evaluation, however, was far from exhaustive and focused on the single domain of Solitaire.

The above ensemble approaches are similar in spirit to the idea of *bagging predictors* (Breiman 1996) in the area of machine learning. Bagging methods construct an ensemble of predictors by running a learning algorithm multiple times, on random samples of the training data, and they make predictions by voting among ensemble members. This approach has been evaluated quite extensively (Dietterich 2000), and the main observation is that when the learning algorithm is "unstable" (has high variance), then bagging is very effective at improving performance compared to using a single predictor. Similar results have been observed for other ways of randomizing the learning algorithm to produce the ensemble (Breiman 2001). In Section 3, we will see that some of the explanations for why bagging works translate well to Monte-Carlo planning ensembles.

3 Ensemble Monte-Carlo Planning

In this section, we review the basic UCT algorithm and describe the particular ensemble approach that we will investigate. Next, we provide some rationale for why this approach might be expected to have advantages compared to traditional UCT.

3.1 UCT

We consider planning in Markov Decision Processes (MDPs) and 2-player alternating-turn stochastic games where each state has a finite set of actions, but the total number of states is too large for standard dynamic programming approaches to be applied. We assume without loss of generality that rewards are only received in a distinguished set of terminal states (e.g. end of game) and that any trajectory is guaranteed to reach and end at a terminal state. For MDPs, the objective is to maximize the expected reward, and for games the objective is to achieve the mini-max value.

UCT is an online planning algorithm, which, given the current state s, selects an action by building a sparse look-ahead tree over the state-space with s as the root, whose edges correspond to actions and their outcomes and whose leaf nodes correspond to terminal states. Each node in the resulting tree stores value estimates for each of the available actions that are used to select the next action to be executed. In the context of games, each tree node is associated with one of the players and the values are with respect to the maximizing, or root node, player.

UCT is distinct in the way that it constructs the tree and estimates action values. Unlike standard mini-max search or sparse sampling (Kearns, Mansour, and Ng 2002), which typically build depth bounded trees and apply evaluation functions at the leaves, UCT does not impose a depth bound and does not require an evaluation function. Rather, UCT incrementally constructs a tree and updates action values by carrying out a sequence of Monte Carlo rollouts of entire decision making sequences from the root to a terminal state. The key idea behind UCT is to intelligently bias the rollout trajectories toward ones that appear more promising based on previous trajectories while maintaining sufficient exploration. In this way, the most promising parts of the tree are grown first while still guaranteeing that an optimal decision will be made given enough rollouts.

It remains to describe how UCT conducts each rollout trajectory given the current tree (initially just the root node) and how the tree is updated in response. Each node s in the tree stores: the number of times the node has been visited in previous rollouts $n(s)$, the number of times each action a has been explored in s in previous rollouts $n(s,a)$, and a current action value estimate for each action $Q(s,a)$. Each rollout begins at the root and actions are selected via the following process. If the current state contains actions that have not yet been explored in previous rollouts, then a random unexplored action is selected. Otherwise, if all actions in the current node s have been explored previously then we select the action that maximizes an upper confidence bound given by:

$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}} \quad (1)$$

where c is a constant that is typically tuned on a per domain basis. The selected action is simulated and the resulting state is added to the tree if it is not already present. This action selection mechanism is based on the UCB rule (Auer, Cesa-Bianchi, and Fischer 2002) for multi-armed bandits and attempts to balance exploration and exploitation. The first term rewards actions whose action values are currently promising while the second term adds an exploration bonus to actions that have not been explored much and goes to zero as an action is explored more frequently.

Finally, after the trajectory reaches a terminal state and receives a reward of R, the action values and counts of each state along the trajectory are updated. In particular, for any

state-action pair (s, a) on the trajectory we perform the following updates:

$$n(s) \leftarrow n(s) + 1; \;\; n(s,a) \leftarrow n(s,a) + 1$$
$$Q(s,a) \leftarrow Q(s,a) + \frac{1}{n(s,a)} \left(R - Q(s,a)\right)$$

According to this update, the action value $Q(s, a)$ is equal to the average reward of rollout trajectories that went through (s, a). Once the desired number of rollout trajectories have been executed UCT returns the root action that achieves the highest Q-value. In our experiments, we follow the common practice of only adding a single node to the tree per rollout trajectory, which is the first node that occurs in the trajectory that is not already in the current tree. Thus, the number of trajectories will typically be equal to the number of nodes in the tree grown by UCT. Note that UCT has two parameters, the exploration constant c, and the number of rollout trajectories t.

3.2 Ensemble UCT

We consider an ensemble UCT approach that is parameterized by an ensemble size n, a number of trajectories t, and an exploration constant c. Given the current state, the ensemble approach makes n independent calls to UCT, using parameters t and c, resulting in n sparse look-ahead trees that are all rooted at s. In an initial investigation we considered various ways of combining the root node statistics from these trees in order to make an overall ensemble decision. We found that the approach used by root parallelization (RP) (Chaslot, Winands, and van den Herik 2008) was nearly always a top performer, which lead us to focus on it. Given the n trees, let $Q^{(i)}(s, a)$ and $n^{(i)}(s, a)$ be the state-action values and counts stored in the root of the i'th tree. The root parallelization action value $Q_{RP}(s, a)$ is then given by:

$$Q_{RP}(s,a) = \frac{\sum_i Q^{(i)}(s,a) \cdot n^{(i)}(s,a)}{\sum_i n^{(i)}(s,a)}$$

which is just the total reward through (s, a) over all trajectories in all trees divided by the total number of times a was tried in s across all trees. Note that this need only be computed for the current state s. The action selected by the ensemble is the one that maximizes $Q_{RP}(s, a)$. We will refer to this action selection approach as Ensemble UCT.

The value of $Q_{RP}(s, a)$ can be viewed as scoring an action a according to the weighted average of scores across the ensemble members, with weights equal to the number of times an ensemble member considered a at the root. In contrast, the combination rule described in (Bjarnason, Fern, and Tadepalli 2009) used an unweighted average of scores, which we found to be less robust in some domains for some ensemble configurations. More complex combining rules such as plurality voting, instant runoff voting, and Borda count voting, were sometimes better than this unweighted average, but never significantly outperformed the simpler RP rule and were often worse.

3.3 Potential Benefits of Ensembles

Clearly Ensemble UCT can be parallelized with very little communication overhead by simply executing the n calls to UCT on independent processors and returning the root node statistics to a central process. This means that if an ensemble with t trajectories per tree achieves higher reward than a single tree with t trajectories, then parallelization will allow us to achieve that higher reward without increasing the time per decision. We will refer to this potential advantage as the *parallel time advantage*.

Similarly, a single-core implementation of Ensemble UCT has approximately the same memory requirement as running only a single instance of UCT, since we need to only remember the root statistics of each tree after it is constructed. Thus, if an ensemble with t trajectories per tree achieves better performance than a single tree with t trajectories, then we can achieve the higher reward without increasing the memory requirements. We will refer to this potential advantage as the *single-core memory advantage*.

Finally, for a single-core implementation, the time per decision is linearly related to the total number of trajectories used to construct the ensemble, which equals $n \cdot t$. Thus, if an ensemble can outperform a single tree with $t' = n \cdot t$ trajectories, then the improved performance can be achieved on a single core with no additional time overhead. We will refer to this potential advantage as the *single-core time advantage*.

Previous work has provided isolated evidence for each of these three advantages; and our experiments will consider these further. However, there remains the question of why we might expect to observe any of these advantages. To help answer this it is useful to draw on the general understanding of why and when bagging predictors improve performance. Bagging has been shown to significantly improve performance for learning algorithms that have high variance (Breiman 1996). In a simplified sense, this is due to the fact that averaging the outcomes of n i.i.d. random variables converges quickly to their expected value. Further, the variance of the average is less than the variance of the individual variables. In the case of bagging, the variance is due to the learning algorithm outputting different predictors on each run, and averaging across those predictions results in a more stable and accurate output.

In analogy, UCT is a randomized algorithm that for a fixed number of trajectories can often have significant variance in the suggested action across independent runs, resulting from the randomness inherent both in the rollout process and in state transitions for stochastic domains. For example, if an early trajectory through an action yields an unlucky result, then the action's value will be temporarily small and UCT will select the action infrequently until the impact of the unlucky run disappears. A very large number of trajectories may be required to overcome such cases. However, such events will not occur in each run of UCT. Thus, if UCT selects good actions "on average" across independent runs, then combining the decisions of a large enough ensemble is likely to be good. As a simplified example, if there are two actions and t is large enough so that the optimal action is returned with probability slightly better than 0.5, then a simple majority vote ensemble will select the optimal action with high probability. This is one reason that one might expect an ensemble of trees with t trajectories per tree to out-

Figure 1: Comparison of Domains

Domain	Agents	c	APS	SPA	STOC
Backgammon	2	1	$\geq 1K$	15	Yes
Biniax	1	8	4	$\geq 7K$	Yes
Connect 4	2	1	7	1	No
Havannah	2	1	61	1	No
Yahtzee	1	64	32	252	Yes

perform a single tree with t trajectories, providing a parallel time and single-core memory advantage.

Note, however, that to observe a single-core time advantage, an ensemble of small trees must outperform a single larger tree. In particular, if the single tree uses t' trajectories, then an ensemble of size n must use no more than t'/n trajectories per tree. Unfortunately, the performance of each tree will decrease to some degree with fewer trajectories due to increased variance and bias of UCT. Thus, there is a fundamental trade-off between using more trees, which can have a variance reduction effect, and more trajectories per tree, which can decrease variance and bias. The current theoretical results for UCT are too coarse to provide significant guidance in making these choices. Thus, this trade-off must be evaluated empirically and is likely to vary from domain to domain.

4 Description of Evaluation Domains

This section gives an overview of the five domains we use in our experiments, which are all discrete, fully observable, and either deterministic or stochastic. Figure 1 gives a comparison of the domains according to the following features: number of agents, value used for the UCT constant, an upper bound on actions per state (APS), an upper bound on the possible next states per action (SPA), and whether or not the domains is stochastic. Space precludes providing all domain rules and details.

Backgammon. Backgammon is an ancient two player, stochastic game for which computers have achieved master level play via reinforcement learning techniques (Tesauro 1994). Each move begins with the roll of two six sided dice, which determines the possible locations to which a player may advance their pieces, the choice of which constitutes the action space. An interesting aspect of the game is that the number of possible actions in a state can vary wildly, from none to over a thousand. In our implementation, a reward of 1 is received for a win and -1 for a loss.

Biniax. Biniax is a single-agent, highly stochastic, arcade-style game for which we are unaware of prior evaluations of any AI techniques. The agent controls a single element on a 5 by 7 board and actions may move the single element to one of the four adjacent, non-diagonal locations, but with some restrictions depending on the surrounding elements. After every pair of moves, all elements on the board shift down one location and the top row is replaced by random elements. The game ends when the agent can make no more legal moves. Note that due to the large number of possible random states after selecting an action, due to the random selection of the top row, standard UCT converges far too slowly in this domain. Thus, we use a sparse version of UCT described in (Bjarnason, Fern, and Tadepalli 2009), which limits the number of next states included in the tree for any given action to a specified constant number.

Connect 4. Connect 4 is a well known deterministic, 2-player game where pieces are dropped into the columns of a vertical 6x7 grid with the goal of forming a straight line of 4 connected pieces. There are at most 7 actions per state, since placing a piece in a column is a legal action only if that column has at least one empty location. The reward is 1, 0, or -1 for a win, draw, or loss, respectively. An optimal rule-based computer player has been built for Connect 4 and it has been shown that the first player has a winning strategy (Uiterwijk, Van den Herik, and Allis 1989).

Havannah. Havannah is a deterministic, 2-player connection game, based on the game Hex. A player wins when they achieve one of several types of connection patterns. No existing computer agent is capable of beating the best human players on a full sized board (side length of 10) and the game designer, Christian Freeling, has put out a prize for anyone that can create an agent that can beat him in 1 of 10 games. The standard board has sides of length 8, but we used length 5 boards in our experiments to speed-up simulation times. The number of possible states is enormous and for our board size a maximum number of actions of 61.

Yahtzee. Yahtzee is a stochastic, single-agent dice game. The game consists of 13 rounds and in each round any number of 5 dice may be rolled up to 3 times. Different configurations of dice correspond to different categories (e.g. a straight), and a category must be selected for scoring each round. An optimal algorithm exists which achieves an average score of 254.5896 with a standard deviation of 59.6117 (Glenn 2006). Human experts are also able to average close to 250 points. The actions in the game correspond to both selecting which die to roll during a round, followed by choosing which category to score.

5 Empirical Results

In this section, we first describe our experimental setup and then present our empirical results for a variety of ensemble configurations and domains.

5.1 Experimental Setup

Each of the five domains was implemented in Java along with our implementation of UCT and Ensemble UCT. The implementations use a common API for communicating between domains and Monte-Carlo planners, making it easy to incorporate new domains and planners into the framework. The source for these domains and planners, among others, are available upon request.

We selected the UCT constant c on a per domain basis, and used the selected value for all ensemble configurations in a particular domain. We selected the value that optimized the performance of a single instance of UCT (ensemble of size 1). In particular, for a range of values of c we generated UCT curves that plotted performance vs. number of trajectories. In all cases, we were able to find a value of c whose curve was nearly uniformly best and selected that value of c

for our experiments (see Figure 1). In results not included in this paper, we have verified that this value of c is typically also best for a wide range of ensemble configurations. Thus, in our experience, it is unnecessary to re-optimize the constant for each ensemble configuration, which simplifies the evaluation.

For the 2-player game domains, we evaluated the performance of Ensemble UCT when playing against a single UCT agent with a fixed number of trajectories. The number of trajectories was selected to balance the trade-off between experimental time and strength of the opponent, which both increase with the number of trajectories. For Connect 4 the fixed opponent used $2^{12} = 4096$ trajectories while Havannah and Backgammon used the smaller numbers of 128 and 256 respectively. The relatively small number of trajectories used in the later cases were due to the relatively slower implementations for these domains, which made it impractical to conduct thorough experiments against opponents with more trajectories. Nevertheless, our results show useful trends for ensemble performance against these opponents.

Generally, in our analysis we used the number of trajectories as a proxy for runtime per decision, which is accurate under the assumption that adding a single trajectory to a tree always takes the same amount of time. Our timing results (not shown) for various ensemble configurations, including single trees, largely agree with this assessment. However, we found this assumption to be slightly violated in domains where there were small but measurable increases in the times to run trajectories for larger trees (on the order of milliseconds per decision). This is domain dependent and primarily due to the fact that for larger trees more node updates are performed for each trajectory as well as more evaluations of the UCT action selection rule. This means that for some domains the performance of ensembles that use smaller trees is slightly better than indicated in terms of the performance achieved per unit time.

Our main experimental results are in Figures 2 through 7, with one table for each of the 5 domains. We also included an additional table for a modified version of the Yahtzee domain (called Modified Yahtzee), which uses a slightly different scoring system that alters the optimal policy compared to the normal Yatzhee domain.[1] Each column in the tables corresponds to a particular ensemble size, or number of trees, n. Each row corresponds to a particular number of trajectories per tree. Thus, each table entry corresponds to a particular ensemble configuration characterized by the number of trees and trajectories per tree. The value recorded in a table entry is the average reward achieved over a series of between 1000 and 4000 games, along with the corresponding 99% confidence interval.[2]

Note that the total number of trajectories used to create an ensemble is simply equal to the number of trees times the trajectories per tree, the value of which is linearly related to the time requirements of a single core implementation. In each table, the range of ensemble sizes and trajectories per tree differs. We attempted to consider the largest possible ranges, the size of which depends on the memory and time requirements for the simulator of each domain.[3] The table entries for configurations that we were unable to run are left blank. In all cases, the ensemble sizes and trajectories per tree are increased by a factor of two across the tables. Due to this choice the ensembles along a diagonal of a table use the same total number of trajectories. For example, a size one ensemble with 2^{12} trajectories per tree, uses the same total trajectories as a size 2 ensemble with 2^{11} trajectories per tree.

5.2 Results

We now consider our experimental results from the perspective of the three potential ensemble advantages: parallel time, single-core memory, and single-core time.

Parallel Time Advantage. Recall that there is a parallel time advantage if the performance of an ensemble with t trajectories per tree is better than the performance of a single tree with t trajectories. In each table, the single tree performance for various values of t can be observed along the first column of results. As we move along each row the ensemble size increases while t remains fixed. We see that, almost uniformly, for all domains and values of t there is an ensemble size that significantly improves on the single tree performance; and typically this happens with only an ensemble of two trees. This provides strong evidence that the ensemble approach can robustly provide a parallel time advantage. The fact that this advantage can be measured with only very small ensembles is particularly useful given the wide availability of machines with a small number of multiple cores.

We further see that for most domains for a fixed value of t the performance steadily increases as the ensemble size grows and typically is substantially better than a single tree for the largest ensemble size. This provides good evidence that the ensemble approach can effectively leverage additional parallel processors when they are available. In some cases the potential speedups allowed by parallelization are quite impressive. For example, in Connect 4, an ensemble of size 16 with $t = 2^{15}$ performs on par with a single tree with $t = 2^{19}$, indicating a speedup of at least a factor of 16.

The Biniax domain was somewhat of an outlier in that there was very little improvement with ensemble size. In fact, for the smaller values of t there is no improvement and even a slight decrease in performance, though not significant according to the confidence intervals. Here there is a significant improvement for the larger ensemble sizes, but it is quite small compared to the other domains. After investigating this issue, we made the following observations. The Biniax domain only has at most four actions per state and we

[1] This domain arose from an early implementation with a "bug" in the scoring compared to the original Yatzhee. We elected to include the results since they represent a huge amount of computation time and add to the diversity of domains considered.

[2] Time constraints prevented running all domains and configurations for the maximum of 4000 games, particularly for larger ensembles and the more costly domain simulators.

[3] Our results were generated on a machine with 4 2.67GHz dual core Intel Xeon processors and 24 gigabytes of memory. Our Java library only ran a single thread so each test used one core for processing and a second core for garbage collection.

Figure 2: Connect 4 Ensemble Table

Trajectories per Tree	Ensemble Size				
	1	2	4	8	16
2^{10}	$-.522 \pm .048$	$-.370 \pm .052$	$-.299 \pm .053$	$-.233 \pm .055$	$-.189 \pm .055$
2^{11}	$-.256 \pm .054$	$-.139 \pm .055$	$-.102 \pm .056$	$-.011 \pm .057$	$-.056 \pm .056$
2^{12}	$.011 \pm .056$	$.121 \pm .056$	$.227 \pm .055$	$.253 \pm .054$	$.284 \pm .076$
2^{13}	$.234 \pm .054$	$.413 \pm .051$	$.507 \pm .048$	$.543 \pm .067$	$.608 \pm .064$
2^{14}	$.470 \pm .049$	$.646 \pm .043$	$.765 \pm .051$	$.842 \pm .042$	$.841 \pm .042$
2^{15}	$.648 \pm .042$	$.793 \pm .048$	$.859 \pm .040$	$.899 \pm .034$	$.918 \pm .031$
2^{16}	$.727 \pm .054$	$.884 \pm .037$	$.886 \pm .036$	$.926 \pm .029$	
2^{17}	$.811 \pm .045$	$.898 \pm .035$	$.917 \pm .024$		
2^{18}	$.871 \pm .038$	$.910 \pm 0.31$			
2^{19}	$.903 \pm .032$				

Figure 3: Yahtzee Ensemble Table

Trajectories per Tree	Ensemble Size				
	1	2	4	8	16
2^9	161.1 ± 1.5	172.4 ± 1.8	179.9 ± 1.8	187.2 ± 1.9	193.4 ± 2.0
2^{10}	172.6 ± 1.5	181.0 ± 1.8	187.9 ± 1.8	193.4 ± 1.8	200.8 ± 2.0
2^{11}	184.0 ± 1.8	190.0 ± 1.8	193.8 ± 1.8	200.1 ± 2.0	204.9 ± 2.0
2^{12}	190.7 ± 1.9	195.0 ± 1.9	200.3 ± 2.0	203.3 ± 2.0	206.8 ± 2.0
2^{13}	196.7 ± 1.8	201.2 ± 2.0	203.3 ± 1.9	205.5 ± 2.0	209.1 ± 2.1
2^{14}	202.4 ± 2.1	205.4 ± 2.0	208.6 ± 2.1	209.7 ± 2.2	
2^{15}	206.1 ± 2.1	208.8 ± 2.2	209.8 ± 2.1		
2^{16}	208.0 ± 2.0	208.4 ± 2.0			

Figure 4: Modified Yahtzee Ensemble Table

Trajectories per Tree	Ensemble Size				
	1	2	4	8	16
2^7	160.3 ± 2.5	167.9 ± 1.5	175.3 ± 2.8	186.3 ± 2.8	193.5 ± 3.3
2^8	172.3 ± 2.8	179.2 ± 1.6	185.9 ± 2.8	193.7 ± 3.0	202.2 ± 3.7
2^9	183.1 ± 2.7	190.2 ± 1.8	197.0 ± 3.4	205.0 ± 3.9	208.3 ± 3.2
2^{10}	191.8 ± 2.8	199.9 ± 1.9	204.0 ± 3.3	207.9 ± 3.2	214.2 ± 3.7
2^{11}	197.9 ± 2.5	206.2 ± 2.0	211.0 ± 3.6	214.7 ± 3.8	217.4 ± 3.7
2^{12}	208.1 ± 3.7	211.1 ± 2.1	214.9 ± 3.9	215.6 ± 3.5	220.6 ± 2.7
2^{13}	209.0 ± 3.3	214.9 ± 1.8	216.4 ± 3.4	218.9 ± 4.0	221.4 ± 2.9
2^{14}	215.2 ± 4.0	217.1 ± 2.2	219.8 ± 2.8	223.4 ± 3.1	221.3 ± 4.0
2^{15}	215.0 ± 3.5	220.7 ± 2.1	220.9 ± 3.7		
2^{16}	216.6 ± 3.7	221.0 ± 3.2			

Figure 5: Backgammon Ensemble Table

Trajectories per Tree	Ensemble Size				
	1	2	4	8	16
2^8	$.021 \pm .057$	$.323 \pm .041$	$.563 \pm .036$	$.734 \pm .030$	$.826 \pm .032$
2^9	$.405 \pm .045$	$.616 \pm .035$	$.758 \pm .029$	$.831 \pm .024$	$.883 \pm .027$
2^{10}	$.659 \pm .033$	$.793 \pm .027$	$.858 \pm .023$	$.885 \pm .020$	$.884 \pm 0.24$
2^{11}	$.787 \pm .027$	$.861 \pm .022$	$.868 \pm .022$	$.888 \pm .020$	
2^{12}	$.881 \pm .021$	$.895 \pm .020$	$.901 \pm 0.21$		
2^{13}	$.899 \pm .019$	$.906 \pm .026$			
2^{14}	$.910 \pm .026$				

Figure 6: Havannah Ensemble Table

Trajectories per Tree	Ensemble Size			
	1	2	4	8
2^7	$.002 \pm .058$	$.135 \pm .057$	$0.235 \pm .056$	$0.507 \pm .050$
2^8	$.237 \pm .056$	$.487 \pm .050$	$0.565 \pm .048$	$0.774 \pm .036$
2^9	$.613 \pm .045$	$.772 \pm .037$	$0.816 \pm .033$	$0.919 \pm .023$
2^{10}	$.876 \pm .028$	$.948 \pm .018$	$0.951 \pm .018$	$0.981 \pm .011$
2^{11}	$.970 \pm .014$	$.984 \pm .010$	$0.986 \pm .010$	
2^{12}	$.999 \pm .003$	$.988 \pm .009$		
2^{13}	$.996 \pm .005$			

Figure 7: Biniax Ensemble Table

Trajectories per Tree	Ensemble Size				
	1	2	4	8	16
2^8	102.1 ± 1.2	102.0 ± 1.2	100.9 ± 1.2	101.2 ± 1.4	101.8 ± 2.4
2^9	103.9 ± 1.2	104.0 ± 1.2	104.4 ± 1.2	103.0 ± 1.4	103.9 ± 2.4
2^{10}	105.9 ± 1.2	105.3 ± 1.2	105.0 ± 1.2	106.6 ± 2.4	107.7 ± 2.4
2^{11}	108.0 ± 1.2	107.9 ± 1.2	107.4 ± 1.2	108.3 ± 2.4	108.7 ± 2.4
2^{12}	109.0 ± 1.2	109.5 ± 1.2	110.6 ± 2.4	110.5 ± 2.4	
2^{13}	110.6 ± 1.2	112.1 ± 1.2	113.8 ± 2.4	114.0 ± 2.4	
2^{14}	111.9 ± 1.2	113.9 ± 1.2			
2^{15}	113.2 ± 1.2				

observed that the variance of UCT's action choices in this domain was very low, in that repeated runs of the algorithm in a state would yield very similar results. We conjecture that this is due to the fact that there is a fairly good and easy to detect strategy in this domain, which is to strive for northward movement. The low variance is a likely explanation for the relative ineffectiveness of ensembles here, since ensembles primarily serve as a variance reduction mechanism. For the larger values of t, the variance appears to increase slightly, which is likely due to the fact that the deeper trees allow for more look-ahead which can randomly vary from run to run. This is a possible explanation for the improved effectiveness for larger t in this domain compared to smaller values of t. This observation follows the conventional wisdom for applying the bagging ensemble approach for learning predictors. In particular, bagging is only effective when the base learning algorithm is unstable or has high variance.

Single-Core Memory Advantage. Recall that there is a single-core memory advantage if the performance of an ensemble with t trajectories per tree is better than the performance of a single tree with t trajectories. Since this is the same condition required for a parallel time advantage, the above discussion applies here and we can conclude that there is a clear single-core memory advantage almost uniformly. It is interesting, however, to point out some notable instances of this advantage. From above, we saw that in Connect 4 an ensemble of size 16 with $t = 2^{15}$ performed on par with a single tree with $t = 2^{19}$ while using 16 times less space. We were unable to run experiments beyond $t = 2^{19}$ due to memory limitations. However, from the trends in the table there is every indication that we could have further improved performance by increasing ensemble size without paying a penalty in terms of space. We see a similar situation in both of the Yahtzee domains where ensembles of 16 small trees can equal the performance of the largest tree that our memory was able to support.

Thus, in domains where there is a single core and the bottleneck is memory, rather than decision time, ensembles can be an extremely effective way to improve performance. An alternative would be to attempt to implement UCT in a way that can effectively use disk space to store the tree. However, this alternative appears to be much more of an engineering effort than the simple ensemble approach we describe. An interesting possibility is considering more complex ensemble configurations for improving performance on multi-core systems when the amount of memory per core becomes the limiting factor. In this case, it seems likely that running ensembles of smaller trees on each core would allow for a way around the memory bottleneck.

Single-Core Time Advantage. There is a single-core time advantage if the performance of an ensemble of size n with t trajectories per tree is better than a single tree with the same number of total trajectories, which is equal to $n \cdot t$. In all of our tables, ensemble configurations on the diagonals going from left-to-right, bottom-to-top use the same number of total trajectories, $n \cdot t$, and hence approximately require the same amount of time per decision on a single core.

There is a clear trend across the domains when looking at the diagonals in the tables. When the number of total trajectories is small, the performance usually degrades slightly as the ensembles become larger. In particular, there is typically a small decrease in performance when going from a single

tree to multiple trees. For example, in Connect 4, a single tree with $t = 2^{13}$ achieves a performance of 0.234, two trees with $t = 2^{12}$ achieves 0.121, and four trees with $t = 2^{11}$ achieve a losing strategy of -0.102. This shows that when the number of trajectories is small, ensembles quite reliably *do not* exhibit a single-core time advantage, but rather exhibit a single-core time disadvantage. To understand why this is the case, note that for a fixed number of total trajectories, if the ensemble size increases, then the value of t must decrease by the same factor. Thus, as the ensemble size grows, each tree in the ensemble becomes less powerful and for a small enough t the quality of their individual decisions degrades to a point that can not be overcome by aggregating results. More formally, as t decreases for the ensemble members, the bias and variance of the individual trees increases to a point where the variance reduction effect of ensembles is not substantial enough. This is similar to the observation that bagging predictors that are highly inaccurate will often not be productive.

However, as the total number of trajectories on a diagonal increases, we typically see that the performance no longer degrades as the ensemble size increases. In most domains, there is no clear increase in performance along the diagonal, indicating that it does not appear to matter whether one adds trajectories to an ensemble by doubling the number of trees or by doubling the number of trajectories per tree. This is a fairly surprising observation as the two choices result in very different computational processes.

While we never observed statistically significant performance improvement along a diagonal, there do appear to be such trends in the Modified Yahtzee and Connect 4 domains for larger numbers of total trajectories. For example, in Modified Yahtzee we see that the largest single tree (i.e. $t = 2^{16}$) has a mean performance that is worse than all ensemble configurations on its diagonal. A similar observation holds in Connect 4. However, none of the individual differences are statistically significant, which prevents us from drawing strong conclusions without increasing the number of repeated runs to decrease the confidence intervals.

Thus, in summary, though prior work in Go and Solitaire offered isolated evidence of a single-core time advantage, we are not able to provide evidence for such an advantage here. However, there are trends in the data, so far not statistically significant, that suggest there might be a single-core time advantage in some of our domains when the total number of trajectories is large enough.

6 Summary and Future Work

We evaluated a simple ensemble approach for UCT, where UCT is used to build multiple independent trees and a final decision is made via a weighted vote. To our knowledge this is the first such evaluation to be conducted over a sizable set of domains and configurations. Our main observations are the following: 1) Ensembles can significantly improve performance per unit time in a parallel model, 2) Ensembles can significantly improve performance per unit memory in a single-core model, and 3) Contrary to some prior observations, we did not observe a significant improvement in performance per unit time in a single-core model.

In future work, we would like to gain a better theoretical understanding of why ensembles work when they do, for example, by better understanding the bias-variance behavior of UCT. We are also interested in exploring more complex configurations that optimize performance given constraints on time, space, and the available CPUs. Finally, we are interested in considering ways of further improving the effectiveness of ensemble methods. For example, work in machine learning has considered various methods of increasing ensemble diversity (such as noise injection or problem transformations), which can sometimes improve performance. While our initial investigations in this direction have not shown benefits, there are many avenues left to be explored.

Acknowledgments

This work was supported by NSF grants IIS-0546867 and IIS-0905678.

References

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

Balla, R., and Fern, A. 2009. UCT for tactical assault planning in real-time strategy games. In *IJCAI*, 40–45.

Bjarnason, R.; Fern, A.; and Tadepalli, P. 2009. Lower bounding Klondike solitaire with Monte-Carlo planning. In *ICAPS*, 26–33.

Breiman, L. 1996. Bagging predictors. *Machine learning* 24(2):123–140.

Breiman, L. 2001. Random forests. *Machine learning* 45:5–32.

Cazenave, T., and Jouandeau, N. 2007. On the parallelization of UCT. In *Computer Games Workshop*, 93–101.

Chaslot, G.; Winands, M.; and van den Herik, H. 2008. Parallel Monte-Carlo tree search. *Computers and Games* 60–71.

Dietterich, T. 2000. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning* 40(2):139–157.

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

Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In *ICML*.

Gelly, S.; Hoock, J.; Rimmel, A.; Teytaud, O.; and Kalemkarian, Y. 2008. On the parallelization of Monte-Carlo planning. In *ICINCO*.

Glenn, J. 2006. An optimal strategy for Yahtzee. *Loyola College in Maryland, Tech. Rep. CS-TR-0002*.

Kearns, M.; Mansour, Y.; and Ng, A. 2002. A sparse sampling algorithm for near-optimal planning in large Markov Decision Processes. *Machine Learning* 49:193–208.

Kocsis, L., and Szepesvari, C. 2006. Bandit based Monte-Carlo planning. In *ECML*, 282–293.

Lang, T., and Toussaint, M. 2010. Planning with Noisy Probabilistic Relational Rules. *JAIR* 39:1–49.

Tesauro, G. 1994. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation* 6(2):215–219.

Uiterwijk, J.; Van den Herik, H.; and Allis, L. 1989. *A knowledge-based approach to connect-four: the game is over: white to move wins!* University of Limburg.