Incentive Learning in Monte Carlo Tree Search

Kuo-Yuan Kao, I-Chen Wu, *Member, IEEE*, Shi-Jim Yen, *Member, IEEE*, and Yi-Chang Shan

***Abstract*—Monte Carlo tree search (MCTS) is a search paradigm that has been remarkably successful in computer games like *Go*. It uses Monte Carlo simulation to evaluate the values of nodes in a search tree. The node values are then used to select the actions during subsequent simulations. The performance of MCTS heavily depends on the quality of its default policy, which guides the simulations beyond the search tree. In this paper, we propose an MCTS improvement, called *incentive learning*, which learns the default policy online. This new default policy learning scheme is based on ideas from combinatorial game theory, and hence is particularly useful when the underlying game is a sum of games. To illustrate the efficiency of incentive learning, we describe a game named *Heap-Go* and present experimental results on the game.**

***Index Terms*—Artificial intelligence, combinatorial games, computational intelligence, computer games, reinforcement learning.**

I. Combinatorial Games

SINCE the 1980s, combinatorial game theory has provided a common mathematical model for the analysis of many two-player, perfect information, zero-sum games. This section reviews the fundamentals of combinatorial game theory (see [1] and [2] for more detail). In combinatorial game theory, a *game* is defined as an ordered pair of sets of games

$$G = \{G^L | G^R\} \tag{1}$$

where G^L and G^R are sets of games. We call the two players of a combinatorial game *left* and *right*. Games in G^L are the *left*'s options and games in G^R are the *right*'s options. The simplest game is game 0, defined as

$$0 = \{\emptyset | \emptyset\}. \tag{2}$$

The *negation* of a game is defined as

$$-G = \{-G^R | -G^L\}. \tag{3}$$

The *sum* of two games is a game

$$G + H = \{G^L + H, G + H^L | G^R + H, G + H^R\}. \tag{4}$$

Manuscript received July 06, 2012; revised October 27, 2012 and January 17, 2013; accepted February 07, 2013. Date of publication February 21, 2013; date of current version December 11, 2013. This work was supported in part by the National Science Foundation of Taiwan under Grant 100-2221-E-346-007.

K.-Y. Kao is with the Department of Information Management, National Penghu University, Magong City 880, Taiwan (e-mail: stone@npu.edu.tw).

I-C. Wu and Y.-C. Shan are with the Department of Computer Science, National Chiao Tung University, Hsinchu 30050, Taiwan (e-mail: icwu@aigames.nctu.edu.tw; ycshan@aigames.nctu.edu.tw).

S.-J. Yen is with the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien 974, Taiwan (e-mail: sjyen@mail.ndhu.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCIAIG.2013.2248086

There is a *partial order* relation defined on the classes of games

$$G \geq H \Leftrightarrow G - H \geq 0 \tag{5}$$

where

$$G \geq 0 \Leftrightarrow \text{ no } x \text{ in } G^R \leq 0. \tag{6}$$

The partial order relation introduces an *equivalence* relation

$$G \equiv H \Leftrightarrow G \geq H \text{ and } H \geq G. \tag{7}$$

Equations (1)–(7) define a *commutative group* of games.

We also use the notation $G \prec | H$ when G is neither greater than nor equal to H.

One notable result of combinatorial game theory is as follows: *all numbers are games*, and they can be added in the usual way. We skip the formal definition of numbers here and just remind the reader that a game can be viewed as a tree where the branches are classified into left and right branches and the terminal nodes are numbers. A sum of games is a collection of trees where each player can choose one in which to move at a turn.

For each combinatorial game, there exist two important values, named the *mean* and the *temperature*. Roughly speaking, the mean is a measure of the average outcome, while the temperature is a rough measure of the move size of a game. The existence of mean values of games was first raised and proved in [3] and [4]. A constructive algorithm, *thermograph*, for the mean and the temperature was due to [1] and [2]. Another approach for calculating the mean and the temperature with partial information of a single option game was proposed in [5]. In this paper, we use the notation $m(G)$ and $t(G)$ to denote the mean and the temperature of a game G.

Conway [1] first introduced the concept of a game's incentive. In [1], given a game $G, \nabla^L(G)$ and $\nabla^R(G)$ are called the *left* and *right incentives* of G, which measure the *precise* move size of a game, as defined as follows:

$$\nabla^L(G) = G^L - G \tag{8}$$
$$\nabla^R(G) = G - G^R. \tag{9}$$

The incentive measures the precise difference in game value before and after a move. Since the sum or the difference of two games is also a game, the incentive is game valued. Note that the *left* and *right* incentives of a game may not be of the same value. One drawback of incentives is that they are not totally ordered, since games are not totally ordered. On the other hand, the temperature of a game G measures the difference between G^L and G^R. There is only one unique game temperature, and it is simplified as a numerical value. Because a single numerical value cannot capture all the detailed information of a move, the temperature is only a rough measure of a game's move size.

When playing a sum of games, players are always concerned about which move has the biggest move size. Although both the temperature and the incentive are measures of the move size, the calculation of these values in a given game is not a simple task.

The goal of this study is to design an automated learning algorithm that can learn incentive values in a *relative manner*. By relative manner, we mean that, instead of learning the exact value of a move size, we learn which move has a *relatively* greater move size than the rest. This paper first discusses how incentive learning works for combinatorial games. Then, a complete framework of incentive learning for state spaces of general groups is presented.

This paper is organized as follows. Section II introduces Monte Carlo tree search (MCTS) research. Section III presents the incentive learning algorithm. Section IV introduces the combinatorial game *Heap-Go*. Section V describes the experiments. Section VI provides concluding remarks.

II. Monte Carlo Tree Search

MCTS [6] is a search paradigm for two-player, perfect-information, and zero-sum games. It has been applied to various computer games [7]–[14], and [19]. This section reviews the basic ideas of MCTS and its related policy improvements, following the definitions of [16].

Let S denote the state space of a game, and s_t be the state of a game at time t. Let A denote the space of actions and let $A(s)$ denote the set of legal actions from state s. The two players alternate turns, at each turn t selecting an action a_t in $A(s_t)$. The game finishes upon reaching a terminal state with outcome z. One player's goal is to maximize z; the other player's goal is to minimize z. A policy π is defined as a stochastic action selection strategy that determines the probability of selecting actions in any given state. $Q^{\pi}(s,a)$ is defined as the expected outcome after playing action a in state s, and then following policy π for both players until termination

$$Q^{\pi}(s,a) = E_{\pi}[z|s_t = s, a_t = a]. \tag{10}$$

The basic idea of MCTS is to evaluate online the expected outcome from simulated games. Each simulated game starts from a root state s_0, and sequentially samples actions until the game terminates. At each step t of the simulation, a simulation policy π is used to select an action a_t. The outcome z of each simulated game is used to update the Q-values encountered during that simulation. This update can be implemented incrementally by incrementing the state-action simulation count $N(s_t, a_t)$ and updating the Q-value toward the outcome z

$$N(s_t,a_t) \leftarrow N(s_t,a_t) + 1 \tag{11}$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \frac{z - Q(s_t,a_t)}{N(s_t,a_t)}. \tag{12}$$

There are two policy stages in the simulations. A *tree policy* is used to select actions in the state s_t represented in the search tree, while a *default policy* is used in those states that are not in the tree, mainly for the playout simulations.

The second idea of MCTS is policy improvement. The values in the search tree are used as references to select actions during subsequent simulations. As the number of simulations increases, the policy π continues to improve. Eventually, with sufficient simulations, the policy will reach the optimal strategy.

The upper confidence bounds applied to trees (UCT) [15] is an example of a policy improvement scheme. It selects actions by using the upper confidence bounds (UCB) algorithm which maximizes an upper confidence bound on the value of actions. Specifically, the Q-value is augmented by an exploration bonus that is high for rarely visited actions, and the policy selects the action a^* maximizing the augmented value

$$Q^{\text{UCB}}(s,a) = Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}} \tag{13}$$

$$a^* = \arg\max_a Q^{\text{UCB}}(s,a) \tag{14}$$

where c is a scalar exploration constant, $N(s)$ is the number of visits to state s, and $\log$ is the natural logarithm. The underlying idea of UCT is to provide a balance between exploitation of the current best action and exploration of other potential better actions.

The rapid action value estimation (RAVE) [16] uses the all-moves-as-first (AMAF) heuristic to estimate the value of each action. The AMAF heuristic assumes there is one general *outcome* value z for each *move*, regardless of when it is played. The RAVE value function $Q^{\text{RAVE}}\ (s,a)$ is the expected outcome from state s, given that a move a is taken at some subsequent turn

$$Q^{\text{RAVE}}(s,a) = E[z|s_t = s, \exists\, u \geq t \text{ s.t. } a_u = a]. \tag{15}$$

RAVE provides a simple way to share knowledge between *similar* actions in the search tree, resulting in a rapid estimate of the action values.

Compared to MCTS, RAVE learns very quickly, but its accuracy and convergence is not as good as the accuracy and convergence of MCTS, when a sufficiently large number of games are simulated. The MCTS–RAVE algorithm [16] overcomes this issue by combining the rapid learning of RAVE with the accuracy and convergence guarantees of MCTS

$$\hat{Q}(s,a) = (1-\beta)Q(s,a) + \beta Q^{\text{RAVE}}(s,a) \tag{16}$$

where $0 \leq \beta \leq 1$ has an initial value of 1 and gradually decreases as the number of simulations increases.

We end this section with some remarks on the previous results. The underlying idea of RAVE is to have one general *outcome* value for each move, regardless of when it is played. Because of the property of "*regardless of when it is played*," RAVE learns the outcome value by sharing knowledge between *similar* nodes in the search tree. On the other hand, for combinatorial games, we also know there is an *incentive* value for each action, regardless of when it is played. Section III studies how to learn these action values of by sharing knowledge between nodes in the search tree.

III. Incentive Learning for Combinatorial Games

In this section, we focus on combinatorial games. From combinatorial game theory, we know that each action has an incen-

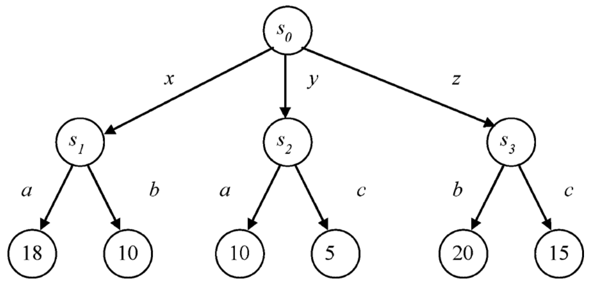

Fig. 1. Monte Caro Tree Search.

tive, as defined in (8) and (9). Note that although the *left*'s goal is to maximize the outcome, and the *right*'s goal is to minimize the outcome, both players try to maximize their own move incentives. In the following discussion, without the loss of generality, we assume that it is the *left*'s turn to move and use the notation ∇a to denote the incentive of action a for the *left*. Each game is a state. Note that an action a in a state s will result in a state $s + \nabla a$.

Given two actions a and b, we have

$$\begin{aligned}&\nabla a \geq \nabla b\\ &\quad\Rightarrow \forall s \in S,\ s+\nabla a \geq s+\nabla b\\ &\quad\Rightarrow \forall s \in S,\ Q(s,a) \geq Q(s,b)\\ &\quad\Rightarrow E\{Q(s,a)-Q(s,b)|s\in S\}\geq 0, \quad \text{(with probability 1).}\end{aligned} \tag{17}$$

So, learning the partial order relation between incentives can help the decision of choosing an action from a state. The rest of this section discusses how to learn the partial order relation between incentives. We define the *incentive difference* between two actions a and b as

$$D^{\text{RIDE}}(a,b) = E\{Q(s,a)-Q(s,b)|s \in S\} \tag{18}$$

where RIDE is the rapid incentive difference evaluation.

Note that the value of (18) can also be learned from MCTS simulations. The idea is that we first learn the incentive difference of all pairs of incentives from the MCTS simulation and then utilize the information to *guess* the partial order relation between the incentives.

Fig. 1 shows an example of a Monte Carlo tree with six simulations. Each node represents a state and each link corresponds to an action. Each terminal node has a value indicating the reward of the simulation from the root to itself. We focus on inference about which action, a or b, is a better action.

Both actions a and b happen twice in the simulation from state s_0. The RAVE values are

$$\begin{aligned}Q(s_0,a) &= (18+10)/2 = 14\\ Q(s_0,b) &= (10+20)/2 = 15.\end{aligned}$$

So RAVE thinks action b is better than action a.

Although actions a and b happen twice in the simulation from state s_0, it happens only once that the two actions start from the same state s_1. The RIDE value between actions a and b is

$$D^{\text{RIDE}}(a,b) = (18-10)/1 = 8.$$

So the RIDE thinks action a is better than action b.

Although both RAVE and RIDE try to estimate the difference between the action values of a and b by sampling from the simulations, their sampling methods are different. RAVE applies an independent sampling method, and any simulation that contains action a or action b is collected into the samples. RIDE applies the pairwise sampling method, and only states that contain both actions a and b as the next actions are collected into the samples.

From (17) and (18), we have

$$D^{\text{RIDE}}(a,b) < 0 \Rightarrow \nabla a \prec |\nabla b \tag{19}$$

and

$$D^{\text{RIDE}}(a,b) > 0 \Rightarrow \nabla a| \succ \nabla b. \tag{20}$$

Let L be a sequence of actions

$$L = \langle a_1, \ldots, a_n \rangle. \tag{21}$$

Ideally, we would like to order the actions in L such that

$$\nabla a_1| \succ \ldots | \succ \nabla a_n. \tag{22}$$

In reality, a sequence of actions in (21) may not be well ordered. So the goal is to find a sequence as *close* to (22) as possible. To define the term "close" more precisely, let D^{RIDE} be the incentive difference function defined in (18), and let L be a sequence of actions as in (21). We define the ordering cost of the sequence L under D^{RIDE} as

$$\text{Cost}(L, D^{\text{RIDE}}) = \sum_{i>j} D^{+}(a_i,a_j) \tag{23}$$

where

$$D^{+}(a_i,a_j) = \begin{cases} D^{\text{RIDE}}(a_i,a_j), & \text{if } D^{\text{RIDE}}(a_i,a_j) > 0\\ 0 & \text{otherwise.}\end{cases} \tag{24}$$

The essential meaning of (23) is that $\text{Cost}(L, D^{\text{RIDE}})$ measures the ordering cost in L by summing up the incentive difference value of all misordered incentive pairs. A sequence with smaller cost is said to be *closer* to (22). The agent can utilize the information of D^{RIDE} to learn a sequence L^* with the minimum ordering cost

$$L^* = \arg\min_{L} \text{Cost}(L, D^{\text{RIDE}}). \tag{25}$$

L^* can be updated incrementally at each time step when the D^{+} value of an action pair has been changed. Without the loss of generality, we assume that the value of $D^{+}(aj, a_i)$ is increased and $i < j$ (if $i > j$, then there is no need to update L^*). One only needs to consider three possible updates to L^*:

- (L_A) move and insert a_i right after a_j;
- (L_B) move and insert a_j right before a_i;
- (L_C) keep L^* unchanged.

The update with the minimum cost will be chosen.

Once L^* is learned, the agent can then utilize this information in the playout simulations. The rule of action selection is simple: select the action with the maximum incentive (ordered by the relation $| \succ$).

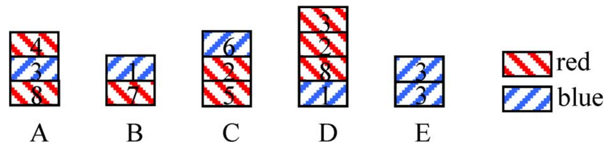

Fig. 2. Setup of *Heap-Go*.

We summarize the incentive learning process as follows.

1) Use the UCB algorithm to select the in-tree path of a new simulation (tree policy).
2) Use L^* to select the off-tree (playout) path of the new simulation (default policy).
3) Start a new simulation $(s_0, \ldots, s_n)$ and learn the outcome z from the simulation.
4) For each action a_t on the path of the simulation, t ranges from $n-1$ down to 0:
    a) update $Q(s_t, a_t)$ based on the outcome z;
    b) compare $Q(s_t, a_t)$ with any other $Q(s_t, a)$ and use the information to update $D(a_t, a)$; update L^* using the information of D.
5) Go to step 1).

The major cost of incentive learning is memory. Let $|A|$ be the size of the action space A, then the incentive learning algorithm needs memory of size order $|A|^2$ to store the D-values.

IV. The Game Of *Heap-Go*

This section describes a combinatorial game named *Heap-Go*, which was first introduced in [18]. In Section V, we will use *Heap-Go* as the test bed for experiments on the incentive learning algorithm introduced in Section III. *Heap-Go* is played on a number of heaps of counters. Each counter has a weight and is colored either blue or red. Fig. 2 shows an example of the initial setup.

Two players, the *left* and the *right*, move alternatively and their legal moves are different.

- When it is the *left*'s turn to move, the *left* can choose any one of the heaps and repeatedly removes the top counter until either he has removed a red counter or the heap has become empty.
- When it is the *right*'s turn to move, the *right* can choose any one of the heaps and repeatedly removes the top counter until either he has removed a blue counter or the heap has become empty.

The game is finished once all the counters in all the heaps have been removed. In the end, the player who removed more total weights is the winner.

Fig. 3 shows the game tree of heap C in Fig. 2. The numbers at the terminal nodes are the net scores of the paths from the root to these nodes. The *left*'s score is counted positive and the *right*'s score is counted negative. For example, consider the path *LR*. The *left* gets eight points for the first move (removed two counters); the *right* gets five points for the second move (removed one counter); the net score is 3.

Note that in *Heap-Go*, each heap is a game with m states, where m is the number of counters in the heap and each state corresponds to a subheap of the heap. The global state is the sum of all the heaps. In Fig. 2, the global state space has a size of $3\times2\times3\times4\times2 = 144$. Each action can only change the state of one heap. Although it seems there are many possible actions, the size of the space of actions is limited. In Fig. 2, there are only $3+2+3+4+2 = 14$ distinct actions, each action corresponding to a move at a state of a heap. In general, a sum of heaps with N counters in total will have N distinct actions in the game tree of the sum. So, the memory requirement for incentive learning for *Heap-Go* is not a problem. One of the major reasons we choose *Heap-Go* as the test bed is because the complexity of the game can be adjusted by simply changing the number of heaps and the number of counters in each heap.

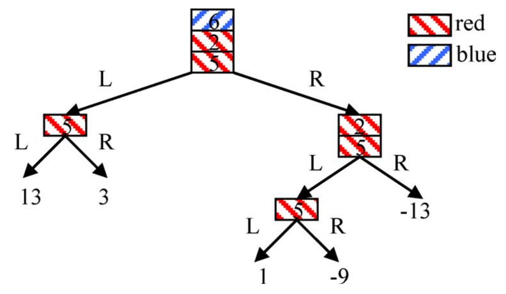

Fig. 3. Game tree of heap C.

Our goal is to learn the partial order relations between the incentives in the *Heap-Go* game. We need a mechanism to evaluate the learning result. From combinatorial game theory, we know that each heap has a temperature. Both the incentive and the temperature can measure the size of a move. For hot games, a move with a higher temperature usually implies a greater incentive, and *vice versa*. So we can measure the performance of incentive learning by comparing the order of the temperatures with the learned incentive orders L^*.

The temperatures of the heaps are calculated outside the incentive learning process. Kao [18] has developed a polynomial time algorithm to calculate the mean and the temperature of a heap. The complexity of the algorithm is $O(n^2)$ where n is the number of counters in a heap. Hence, we can measure the performance of incentive learning by comparing the output action sequence L of incentive learning with the derived temperatures of the heaps, viewed as the correct results.

V. Experimental Result

We performed an experiment of the new temperature learning algorithm for the *Heap-Go* game. The setup of the experiment is as follows.

1) The experiment consists of several conditions with varied game sizes, varied numbers of simulations for training, and varied default policies. Each session runs 100 random games.
2) At the beginning of each run, the experiment randomly selects m heaps, each with n counters, and the weights and colors of the counters are randomly generated where the weights range between 1 and 10 and the colors are either blue or red.
3) The temperature of each heap state is calculated before the learning process starts. These temperatures are used

TABLE I
R^L MATCH RATE (PERCENT) (DEFAULT POLICY: MAXI)

no. of simulations / game size $(m\times n)$	$(m\times n)^1$	$(m\times n)^2$	$(m\times n)^3$
3×3	66.56	88.26	89.30
4×4	62.51	84.98	93.68
5×5	61.48	81.27	93.63
6×6	60.77	78.70	91.59
7×7	60.38	75.90	89.87

to setup the priority relation R^T between any two actions, where actions with higher temperatures have higher priorities.

4) The learning process outputs an incentive list L^*, which is used to set up the priority relation R^L between any two incentives.
5) The relations R^T and R^L are compared to each other, and the percentage of agreed match, called *match rate* between R^T and R^L, is recorded. Actions belonging to the same heap are excluded from comparison, since the rule of *Heap-Go* implies that these actions will never happen at the same time.
6) Within each run, the $Q^{\rm RAVE}$ values of all the actions are also recorded. These $Q^{\rm RAVE}$ values are used to set up the priority relation R^Q between any two actions. The relations R^T and R^Q are compared to each other, and the match rate between R^T and R^Q is recorded.

In the experiment, UCT is used as the tree policy, and there are two versions of the default policy. The first version, called *random default* or *RAND default*, selects the action randomly; the second version, called *maximum incentive default* or *MAXI default*, selects the move according to L^*. The experimental results are shown in the following.

The data in Table I are summarized as follows.

- Even with a very small number $m \times n$ of simulations, the average match rate (between R^L versus R^T) quickly runs above 60%. The average match rate increases as the number of simulations increases and almost reaches 90% within $(m \times n)^3$ simulations. This implies that incentive learning converges fast and the agent's default policy is getting closer to the *thermostrategy*, which selects the action with the maximum temperature.
- The average match rate never reaches 100%, even if a game tree is completely explored (in the case of game size 3×3). However, the high match rate, above 89%, implies that an action sequence ordered by action temperatures is very close to the action sequence with the minimum ordering cost.

  The average match rate decreases as the action space size increases, under a fixed number of simulations. A low match rate implies that the agent's action policy is still far from the thermostrategy. The match rate can also be used as an indicator of the performance of MCTS.

Fig. 4 shows the comparison of the match rate between two different default policies at games with size 5×5. It shows that

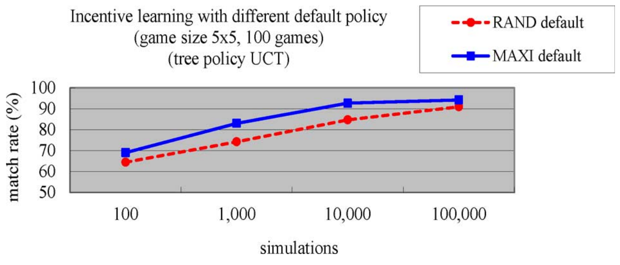

Fig. 4. Incentive learning with different default policy.

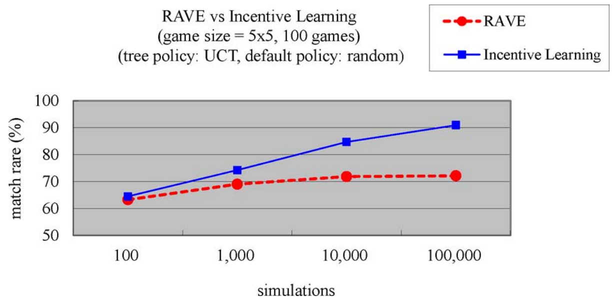

Fig. 5. RAVE and incentive learning.

the MAXI default policy has a higher match rate than the RAND default policy. Note that to achieve the same match rate as the MAXI default policy, the RAND default policy needs almost ten times as many simulations. This indicates that in incentive learning the MAXI default policy does improve the performance of MCTS significantly.

Fig. 5 shows the comparison of the match rate between RAVE and incentive learning. Both RAVE and incentive learning use the same UCT tree policy and the random default policy at games with size 5×5. Fig. 5 shows that the match rate of incentive learning is far above the one of RAVE. Overall, the incentive learning result is very positive, either compared to no learning (match rate 50%) or compared to RAVE.

Each 5×5 *Heap-Go* game has 25 distinct actions (each action acts on the jth counter of the ith heap), and $25 \times 20/2 (= 250)$ action pairs (actions on the same heap are not compared to each other). Because 100 games are played in each condition in the experiments of Figs. 4 and 5, there are in total $100 \times 250 = 25\,000$ action pairs. The match rate is measured on the space of action pairs. The statistical outcome of 25 000 samples should be significant enough (with a standard error much less than 1%).

Fig. 6 shows the comparison between the winning rates of two different default (playout) policies: the RAND default and the MAXI default; the latter applies the outcome of RIDE to guide the playouts. Both default policies are equipped with the UCT tree policy. The experiment takes 1000 sample games and has a range of numbers of simulations from 100 to 100 000 for each move. For each random game, the two default policies have two matches; one of them plays first at each match. The sum of the scores in the two matches is added together to determine the winner of the game. The outcome clearly shows that the MAXI default has a higher winning rate than the RAND default when the number of simulations is more than 100. It is also interesting to see that the percentage of ties also increases as the number of

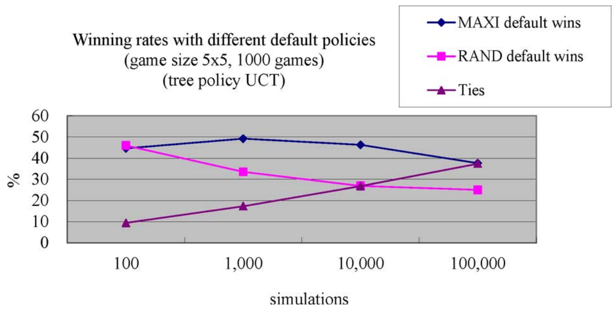

Fig. 6. Winning rates of different default policies.

simulations increases. Indeed, if two players are both optimal, then the tie rate should be 100%. The increase of the tie rate implies that both players are playing better as the number of simulations increases.

VI. Conclusion and Further Consideration

This paper introduces the following heuristic: "if an action leads to better rewards than other actions from the same state, then it is highly likely that this action has a greater incentive value than other actions." We present an online incentive learning scheme applied to MCTS. The outcome of incentive learning is a list of actions ordered by their incentives. This information is then used by the default policy to select the action with the maximum incentive.

Each simulation in MCTS carries a lot of information. The quality of MCTS depends on how much one can learn from each simulation. The majority of past research focuses on the learning of action values. This paper introduces the learning of incentive orders. Our experiment on *Heap-Go* has shown very positive results with incentive learning.

We have seen in this paper how incentive learning can be applied to the state space of combinatorial games. Indeed, the general framework of incentive learning, as shown in (18), (23), (24), and (25), can be applied to general state spaces.

Both RAVE (15) and RIDE (18) use the same AMAF heuristic: there is one general action (incentive) value for each action, regardless of when it is played. The major difference between RAVE and RIDE is their sampling strategy. RAVE applies an independent sampling strategy, while RIDE applies a pairwise sampling strategy. In order to compare the action value of two actions a and b, RAVE estimates the action values of a and b independently and regardless of their starting states, and then compares their values. RIDE estimates the pairwise action-value difference directly, as long as the starting states are the same.

Bouzy and Chaslot [20] also came up with the idea of learning by pairwise comparison and using the result to help play out the simulations. They used a formula with certain parameters to learn the differences of action values between actions, and each action finally received an absolute action value

$$Q(a) \leftarrow Q(a) - \alpha(Q(a) - Q(b) - C(V_a - V_b))$$

where V_a and V_b are the state values of two MC evaluations, C is a constant ranging from 0.7 to 1.0, and α is proportional to the inverse of the square root of a number of visits to action a. The RIDE method introduced in this paper compares the relative orders between action pairs, and there is no absolute action value for each action. Indeed, Bouzy and Chaslot's formula is similar to RAVE (15) in the sense that an action value is updated regardless of when it is played. On the other hand, RIDE only learns the difference between two actions when these actions are from the same state (18).

One issue of incentive learning is required memory. Let $|A|$ be the size of the action space A. The incentive learning algorithm needs memory of size order $|A|^2$ to store D-values. This issue may incur a limit on the applicability of incentive learning to other games. The game of *Heap-Go* has an incentive space size equal to the total number of counters in the *Heap-Go* game. Hence, the memory problem is not critical. For games with a large or infinite action space, further research is needed to handle the memory problem.

The state space of the *Heap-Go* game could be very large. An $m \times n$ *Heap-Go* game with m heaps of n counters has a state space size of n^m. The state space size of a 19×19 *Go* game is around $3^{361} (\fallingdotseq 10^{172})$, which is about the same size as the 361×3 or 172×10 *Heap-Go* game. The method introduced in this paper can be applied to any $m \times n$ *Heap-Go* game as long as memory can store $(m \times n)^2$ action pairs.

Another thing to note about incentive learning is its divide-and-conquer feature. The task of incentive learning can be arranged in a divide-and-conquer manner. For example, consider a *Heap-Go* game with $2n$ heaps. At the first run, one can first divide the $2n$ heaps into two groups A and B, where each group has n heaps, and learns the incentive orders separately within each group. Since the incentive of an action at one heap is independent of the states of other heaps, the learning results from the two groups A and B are reliable. The outputs of the first run are two ordered lists, say $L_A = \langle a, b, c \rangle$ and $L_B = \langle d, e, f \rangle$. At a second run, one can divide the 2 n heaps into another two groups C and D and learn the incentive orders separately within each group. The outputs of the second run are another two ordered lists, say $L_C = \langle a, e, f \rangle$ and $L_D = \langle b, c, d \rangle$. It is possible that the learning result from the two runs can be merged and produces a complete incentive order, for example, $L = \langle a, b, c, d, e, f \rangle$. If the results of the previous runs are not sufficient to determine a complete ordered list, one can continue additional runs. The divide-and-conquer feature of incentive opens a door for further research into more efficient incentive learning algorithms.

Although our incentive learning scheme can be applied to general MCTS, further studies are needed to testify as to its performance in various problems. One way of measuring the feasibility of incentive learning for a problem is to check whether the match rate between the output action sequence L of incentive learning and the output action sequence of MCTS increases with increased simulations. The extension of incentive learning to other general problems deserves further research.

References

[1] J. H. Conway, *On Numbers and Games*. New York, NY, USA: Academic, 1976.

[2] E. R. Berlekamp, J. H. Conway, and R. K. Guy, *Winning Ways*. New York, NY, USA: Academic, 1982.

[3] J. Milnor, "Sums of positional games," in *Contributions to the Theory of Games*, Kuhn and Tucker, Eds. Princeton, NJ, USA: Princeton Univ. Press, 1953, pp. 291–301.

[4] O. Hanner, "Mean play for sums of positional games," *Pacific J. Math.*, vol. 9, pp. 81–99, 1959.

[5] K. Kao, "Mean and temperature search for Go endgame," *Inf. Sci.*, vol. 122, no. 1, pp. 77–90, Jan. 2000.

[6] R. Coulom, "Efficient selectivity and backup operators in Monte-Carlo tree search," in *Proc. 5th Int. Conf. Comput. Games*, 2006, pp. 72–83.

[7] S. Gelly and D. Silver, "Achieving master level play in 9 × 9 computer Go," in *Proc. 23rd Conf. Artif. Intell.*, 2013, pp. 1537–1540.

[8] H. Finnsson and Y. Björnsson, "Simulation-based approach to general game playing," in *Proc. 23rd Conf. Artif. Intell.*, 2013, pp. 259–264.

[9] R. Lorentz, "Amazons discover Monte-Carlo," in *Proc. 6th Int. Conf. Comput. Games*, 2013, pp. 13–24.

[10] M. Winands and Y. Björnsson, "Evaluation function based Monte-Carlo LOA," in *Proc. 12th Adv. Comput. Games Conf.*, 2009, pp. 33–44.

[11] J. Schäfer, "The UCT algorithm applied to games with imperfect information," Diploma thesis, Institut für Simulation und Graphik, Otto-von-Guericke-Universität Magdeburg, Magdeburg, Germany, 2008.

[12] N. Sturtevant, "An analysis of UCT in multi-player games," in *Proc. 6th Int. Conf. Comput. Games*, 2013, pp. 37–49.

[13] R. Balla and A. Fern, "UCT for tactical assault planning in real-time strategy games," in *Proc. 21st Int. Joint Conf. Artif. Intell.*, 2009, pp. 40–45.

[14] F. Teytaud and O. Teytaud, "Creating an upper confidence tree program for Havannah," in *Proc. 12th Adv. Comput. Games Conf.*, 2009, pp. 65–74.

[15] S. Gelly, "A contribution to reinforcement learning; Application to Computer Go," Ph.D. dissertation, Laboratoire de recherche en informatique, Univ. South Paris, Paris, France, 2007.

[16] S. Gelly and D. Silver, "Monte-Carlo tree search and rapid action value estimation in Computer Go," *Artif. Intell.*, vol. 175, no. 11, pp. 1856–1875.

[17] P. Drake, "The last-good-reply policy for Monte-Carlo Go," *Int. Comput. Games Assoc. J.*, vol. 32, no. 4, pp. 221–227, Dec. 2009.

[18] K. Kao, "On sums of hot and tepid combinatorial games," Ph.D. dissertation, Dept. Math., Univ. North Carolina at Charlotte, Charlotte, NC, USA, 1997.

[19] C. Browne, E. Powley, D. Whitehouse, S. Lucas, P. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton, "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intell. AI Games*, vol. 4, no. 1, pp. 1–43, Mar. 2012.

[20] B. Bouzy and G. M. J.-B. Chaslot, "Monte-Carlo Go reinforcement learning experiments," in *Proc. IEEE Symp. Comput. Intell. Games*, 2006, pp. 187–194.

Kuo-Yuan Kao received the B.S. degree in information engineering from the National Taiwan University, Taipei, Taiwan, in 1989 and the Ph.D. degree in mathematics from the University of North Carolina at Charlotte, Charlotte, NC, USA, in 1997.

He is one the early pioneers of *Computer Go*. Since 1995, he has engaged in the research of combinatorial game theory and published several papers in this field. He is a 6-dan amateur *Go* player.

Prof. Kao won the U.S. Computer Go Championship in 1994. He is a member of the Taiwan AI association. Since 2007, he has served as the Director of the Chinese Go Association, Taiwan.

I-Chen Wu (M'10) received the B.S. degree in electronic engineering and the M.S. degree in computer science from the National Taiwan University (NTU), Taipei, Taiwan, in 1982 and 1984, respectively, and the Ph.D. degree in computer science from Carnegie Mellon University, Pittsburgh, PA, USA, in 1993.

He is currently a Professor of the Department of Computer Science, and the Director of the Institute of Multimedia Engineering at the National Chiao Tung University, Hsinchu, Taiwan. He introduced the new game, *Connect 6*, a kind of six-in-a-row game, in 2005. Since then, *Connect6* has become a tournament item at the Computer Olympiad. He led a team developing various game-playing programs, winning over 20 gold medals in international tournaments, including the Computer Olympiad. His research interests include artificial intelligence, Internet gaming, volunteer computing, and cloud computing. He wrote over 80 papers.

Dr. Wu served as a chair and a committee member in over 30 academic conferences and organizations, including the Games Technical Committee of the IEEE Computational Intelligence Society.

Shi-Jim Yen (M'10) received the B.S. degree in computer science and information engineering from Tamkang University, New Taipei, Taiwan, in 1991, the M.S. degree in electrical engineering from the National Central University, Zhongli City, Taiwan, in 1993, and the Ph.D. degree in computer science and information engineering from the National Taiwan University, Taipei, Taiwan, in 1999.

He is currently a Professor in the Department of Computer Science and Information Engineering, National Dong Hwa University, Hualien, Taiwan. He has specialized in artificial intelligence and computer games. In these areas, he has published over 100 papers in international journals and conference proceedings. He is a 6-dan *Go* player.

Prof. Yen received the Excellent Junior Researcher Project Award from the Taiwan National Science Council in 2012 and 2013. He served as a Program Chair of the 2015 IEEE Conference on Computational Intelligence and Games, and a Workshop Chair of 2010–2013 Conferences on Technologies and Applications of Artificial Intelligence. He served as a Workshop Cochair of the 2011 IEEE International Conference on Fuzzy Systems. He has been the Chair of the IEEE Computational Intelligence Society (CIS) Emergent Technologies Technical Committee (ETTC) Task Force on Emerging Technologies for Computer Go since 2009. His team develops many strong board game programs including *Go*, *Chinese Chess*, *Dark Chess*, *Connest6*, and many puzzle games. These programs have won the gold and other medals numerous times at the Computer Olympiad, TAAI tournaments, and TCGA tournaments since 2001.

Yi-Chang Shan received the B.S. and M.S. degrees in computer science from the National Taiwan Normal University, Taipei, Taiwan, in 1991 and 2002, respectively, and the Ph.D. degree in computer science from the National Chiao Tung University, Hsinchu, Taiwan, in 2013.

His research interests are computer game, combinatorial game theory, volunteer computing and cloud computing.