Multi-Agent Monte Carlo Go

Leandro Soriano Marcolino
Matsubara Laboratory – Intelligence Information Science Department
Future University of Hakodate
Hakodate, Japan
g2209001@fun.ac.jp

Hitoshi Matsubara
Matsubara Laboratory – Intelligence Information Science Department
Future University of Hakodate
Hakodate, Japan
matsubar@fun.ac.jp

ABSTRACT

In this paper we propose a Multi-Agent version of UCT Monte Carlo Go. We use the emergent behavior of a great number of simple agents to increase the quality of the Monte Carlo simulations, increasing the strength of the artificial player as a whole. Instead of one agent playing against itself, different agents play in the simulation phase of the algorithm, leading to a better exploration of the search space. We could significantly overcome Fuego, a top Computer Go software. Emergent behavior seems to be the next step of Computer Go development.

Categories and Subject Descriptors

I.2.1 [**Artificial Intelligence**]: Applications and Expert Systems—*Games*

General Terms

Algorithms, Experimentation

Keywords

Emergent Behaviour, Collective Intelligence

1. INTRODUCTION

Go is a two-player turn-based strategy board game, that is famous for being one of the main challenges in Artificial Intelligence. A small set of simple rules[1] leads to a game amazingly complex for a human being and a search tree that is unbearably large for a computer. There are many reasons for this difficulty of developing a strong artificial player. First, Go is played in a large board, 19x19, with 361 intersections, creating difficulties for tree search based algorithms. Second, generally most of the intersections are valid movements, increasing the number of possible states from a given state of the board. Third, the stones interact in complex ways during the game; one stone may influence a distant group, for example in situations where there is a *ladder*. Besides, building an evaluation function is not trivial. Even end of game situations, that intuitively should be simpler, were proved to be PSPACE-hard [31]. According to [1], compared to the complexity of Chess (10^{50}), the complexity of Go (10^{160}) is bigger by a factor of 10^{110}. We can see, therefore, how challenging it is to create an artificial player of Go.

However, recently, with the development of evaluations of the board state based on simulations (known as Monte Carlo techniques), the strength of Computer Go players improved significantly. Thanks to artificial players like MoGo, Crazy Stone, Fuego, Many Faces of Go, and Zen, the best Go programs are now considered amateur level 2 dan. Further improvement was achieved by parallelization, as it increases the computational power, allowing a deeper exploration of the possible movements. In February 2009, Many Faces of Go, running on a 32-core Xeon cluster, beat the professional player James Kerwin, in a 19x19 board with a handicap of 7 stones. Many recent works are now investing in the parallelization of Monte Carlo techniques. However, there is always a limit in the amount of speed-up that can be gained in a parallelization design.

Generally, there are two ways to increase the strength of an artificial player: advances in computational power, which can be achieved by parallelization, and advances in the theory, which can be achieved by new algorithms and methods. Nowadays, the research in Monte Carlo techniques seems to be focused on the parallelization of the current approaches. However, it is always desirable to advance the theory with the creation of better algorithms, that lead to stronger players even when the computational power has not necessarily increased. We believe that the next theoretical step lies in the investigation of Multi-Agent methodologies.

Multi-Agent systems have been used to solve a great range of problems in Artificial Intelligence. The emergent behavior of a great number of simple agents have been applied in algorithms like *Ant Colony Optimization* [11], *Particle Swarm Optimization* [20], etc, in order to solve difficult optimization problems. It is also notable how emergence can lead to complex and intricate group behavior [21, 22, 23, 28].

Emergence is a powerful concept, not only in Computer Science, but also in a variety of disciplines, like philosophy, systems theory and art. The stock market and the Internet are important systems to modern life that arise thanks to the emergence of simple components. Emergence is also fundamental in biological systems. A notable example is an ant colony. It is known that the queen does not order directly the ants. Each ant is always reacting to stimuli generated

[1]Available at many places, for example: http://www.pandanet.co.jp/English

Cite as: Multi-Agent Monte Carlo Go, Leandro Soriano Marcolino, and Hitoshi Matsubara, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. 21-28.

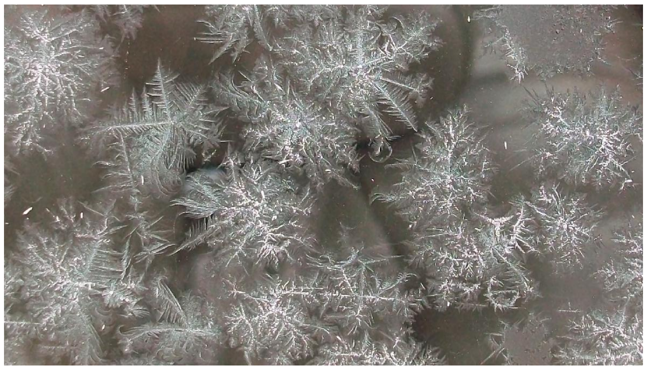

Figure 1: Water crystals, formed by a natural emergent process (taken from www.wikipedia.org).

by chemical scent from larvae, other ants, intruders, food, waste, etc, and they leave chemical that will be used as stimuli to other ants. Therefore, there is no centralized control, but the ant colonies exhibit complex behavior and are able to solve complex problems. Another example is the formation of water crystals on glass, a natural emergent process created by the random motion of water molecules, that leads to a highly-organized structure (Figure 1).

However, emergence is generally not a clear concept. In this paper we define emergence as a great number of simple iterations that occur in a system, leading to a complex result. We can model the Monte Carlo evaluations as one agent that repetitively plays against itself using a playout strategy. Although the playout strategy might be simple, the combination of a great number of games with a tree exploration phase makes intelligent game play emerge in a Monte Carlo Tree Search algorithm (MCTS). In this paper we explore this further, by evaluating the effects of having not only one, but many different agents at the playout phase of a MCTS.

At each stage of the Go board, one agent is selected to generate a movement, leading the board to the next stage. The agents act in turn, therefore there is no spatial organization, but a temporal organization. However, each agent acts in the environment that was left by the previous one, and this interaction seems to lead to a higher playing strength. As the interactions are simple, but they lead to something complex (high-level go), we believe emergence is a good concept to define our idea.

Our proposed algorithm is also inspired by the advantages of diversity. It is currently believed by some social scientists and economists that the best teams are not necessarily composed of the best individuals. In order to build a team that is effective in solving problems, it is also important to look for diversity, to bring together people with different perspectives and solution strategies [27]. By using different agents during the simulation process, we are also exploring this concept, but in a Multi-Agent context.

We modify Fuego [13], an open source implementation of a powerful MCTS algorithm: UCT Monte Carlo Go. Therefore, the contribution of this paper is to offer a new paradigm for the exploration of Monte Carlo Go. Our experimental analysis show that we could significantly overcome Fuego, and produce a stronger Computer Go program.

2. RELATED WORK

A great variety of approaches have been proposed in the literature in order to tackle with the complexity of Go. The problem is too difficult for a conventional α-β search, forcing the researchers to try many different methods. An interesting survey of the literature can be found in [6]. Classical works used abstractions [15] and patterns [4]. Other important approaches that have been explored include learning [8], cognitive modeling [5] or combinatorial game theory [24].

Generally, in the classical way to develop a Go program, specific game knowledge has to be implemented. Therefore, many algorithms were proposed to resolve specific subproblems of the game [3]. However, the Monte Carlo approach appeared, which originally used only the simple Go rules to perform random simulations in order to discover good positions to play [7]. Later, the Monte Carlo simulations were used to evaluate leafs in tree search algorithms, and the simulations started to use heuristics, which included some Go knowledge, in order to improve their realism, as in [12]. The state of the art was further advanced by the UCT Monte Carlo algorithm [17], which contributed with significant improvements in playing strength. The proposed program, MoGo, won all the tournaments on the international Kiseido Go Server[2] on October and November 2006.

In order to achieve further enhancements, parallel and distributed versions of the game started to appear on the literature. Generally, the idea is to use a great number of machines or processors to increase computation power. According to [10], three different parallelization approaches are possible in UCT Monte Carlo: root parallelization, leaf parallelization and tree parallelization. In root parallelization each thread is responsible for one tree, and when the time is finished, the results are merged. In leaf parallelization, many simulations are executed to evaluate a single leaf, each one by one thread. In tree parallelization, many threads execute in a single, shared tree. In [16] it is proposed a straight algorithm for multi-core parallelization, based on shared memory, and an algorithm for cluster parallelization that uses less messages than a simple generalization of the multi-core algorithm. The multi-core algorithm achieved a 63% percentage of victory by doubling the computational power and the cluster algorithm achieved 83.8% percentage of victory by using 9 machines. Some works propose distributed systems based on a client/server architecture in order to increase the number of available playouts [18]. Recently, a top Computer Go program, Zen, was run in a large cluster of computers [19]. A similar approach is also investigated by [9], where a percentage of victory of 70.50% could be achieved against GnuGo, using 16 slaves. However, the results do not improve with a higher number of slaves, and even decreased in some cases. Root parallelization in the Fuego system was studied by [29], where experiments with 64 cores demonstrated that although the program gets stronger, there are limitations in the possible performance gain.

Recently, distributed versions of the top Computer Go programs have won against professional players in handicap games. However, it is known that the overhead of the parallelization imposes a limit to the possible improvement in game strength. In [18], for example, in 9x9 boards the system saturated with 7 servers, and the use of 4 servers brought a speed-up factor of only 1.55. In [10], tree par-

[2]http://www.weddslist.com/kgs/past/index.html

allelization only scaled well up to 4 threads. A lock-free parallelization was proposed by [14], but it could not scale beyond 7 threads.

The next step seems to be converging into Multi-Agent System paradigms. Some works started to apply this idea, but in order to play other games. In [25], a consultation system to play Shogi is proposed. A set of players send their opinion about what should be the next movement, and one of the opinions is selected as the official movement. The authors show that a consultation system composed of three famous Shogi programs plays better than each software individually. In [30] the authors extend the last approach, but this time they use the position evaluation of different players in order to select a single movement. The number of agents in these works was limited, though, with at most 6 agents. In [26], the authors explore a Swarm Intelligence Algorithm, Stochastic Diffusion Search, to build an artificial Othello player. We believe that the use of Multi-Agent Systems has to be further explored, and it can be the next cornerstone in Computer Go development.

Some social scientists and economists currently believe that teams of diverse people can have strong characteristics for solving difficult problems [27]. By combining different perspectives and solution strategies, a diverse team can explore a greater range of possible solutions for a problem; while a team with high-talented but similar individuals might not be able to explore so many different solutions, as each member will tend to have similar results as the other members of the group. Therefore, a team of diverse members might perform better than a team with the best individuals. This concept is also an important point to be explored in the development of Multi-Agent paradigms for Computer Go.

In this work we are going to extend the top MCTS algorithm, UCT Monte Carlo Go, with a Multi-Agent System paradigm. Instead of showing the computational power gains that can be obtained by parallelization or distribution, we are going to show how the emergent properties of a great number of simple (and diverse) agents, by itself, can enhance the strength of an artificial Go player.

3. METHODOLOGY

First, we are going to introduce UCT Monte Carlo Go. The algorithm is based on the multi-arm bandit problem. A multi-arm bandit is like a traditional slot machine, but with many arms. Each arm has a reward drawn from an unknown probability distribution. The objective is to maximize the total sum of iterative plays. When choosing an arm to play, there is a balance between selecting the best arm found so far, or exploring other arms. In [2], it is proposed a simple algorithm, called UCB1 in order to solve the selection problem. Let's define the K-armed bandit problem by the random variables $X_{i,n}$, for $1 \leq i \leq K$ and $n \geq 1$. Each variable is the reward of arm i when it is played at time n. Given a certain arm i, the rewards $X_{i,n}$ are independent for all n, and are identically distributed according to an unknown probability distribution. The rewards across arms are also independent, but they might not be identically distributed.

The algorithm selects the arm j, that maximizes $\bar{X}_j + \sqrt{\frac{2\log n}{T_j(n)}}$, where n is the overall number of plays up to the current iteration, $T_j(n)$ is the number of times arm j has been played after the first n plays, and $\bar{X}_j$ is the mean of the values obtained so far when arm j was selected . In [2], it is also introduced a slightly more complicated algorithm, called UCB1-TUNED, that had better experimental results. First, they calculate an estimation of the upper bound on the variance of arm j, by:

$$V_j = \left(\frac{1}{T_j(n)}\sum_{y=1}^{T_j(n)} X_{j,y}^2\right) - \bar{X}_j^2 + \sqrt{\frac{2\log n}{T_j(n)}}$$

Then, they select the arm j that maximizes the following equation:

$$\bar{X}_j + \sqrt{\frac{\log n}{T_j(n)} min\{1/4, V_j\}} \qquad (1)$$

In UCT Monte Carlo Go, each Go board situation is seen as a bandit, and each possible move is seen as an arm with unknown reward of a certain distribution. Generally, the algorithm can be defined by two phases: tree search and leaf evaluation (also known as playout). The tree search phase starts at the root of the tree. At each node (Go board situation), the child-node (possible move) that maximizes Equation 1 (UCB1-TUNED) is selected as the next node to be visited. This is executed recursively, always choosing the child-node according to UCB1-TUNED. When a node is selected that has never been visited before, the next phase is executed: score estimation by Monte Carlo simulations, where heuristic-driven random games are executed from the state of the leaf until the end of the game. Generally the heuristics are designed in a way that the end game can be easily recognized, and the final score easily calculated. The final score is used to estimate the value of the leaf. The value of the nodes in the path are then updated iteratively, from the father-node of the selected leaf to the root. Note that the Go board states created during the Monte Carlo simulations will not become part of the tree, they are used only to estimate the value of the leaf. Improving the quality of the simulations will improve the estimation of the score, leading to a stronger player [32].

We can model the random simulations as one agent playing against itself using its available heuristics (Figure 2(a)). In this work, we investigate the effects of having not only one, but several agents playing against each other (Figure 2(b)). Each agent has a different playing style, increasing the range of exploration of the search space. As will be further explained, at every stage of the simulation process, a different agent will be selected in an *agent database*, and this agent will be responsible for selecting the next movement. Note, therefore, that (contrary to our first idea) in our approach we are not executing a tournament between different agents, as one agent does not play a full game against another.

We based our implementation on Fuego, an open source UCT Monte Carlo Go algorithm. The Fuego system executes several heuristics hierarchically. It starts by selecting the first heuristic. In case it cannot generate a movement, it proceeds by selecting the next one on the hierarchy. The process repeats until a heuristic generates a movement. If no heuristic can generate a movement, a random move is selected from the board. Generally the heuristics are applied in the neighborhood of the last movement. The current version of Fuego (0.4) has mainly five heuristics: **Nakade** If there is a region of three empty points, generates a move-

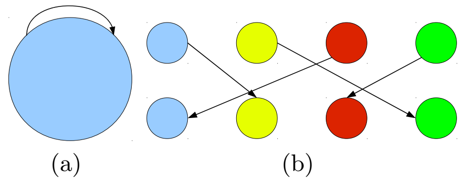

Figure 2: Original single-agent Monte Carlo (a) and proposed Multi-Agent Monte Carlo (b). The colors represent different agents, and the arrows represent interaction.

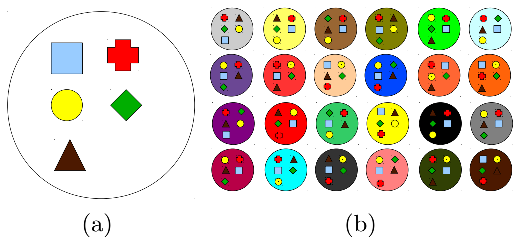

Figure 3: Original Fuego agent (a) and new *agent database* (b).

ment in the center of this region; **Atari Capture** Captures an Atari; **Atari Defend** Defends an Atari; **Lowlib** Move generator for 2-liberty blocks; **Pattern** Uses a set of 3x3 patterns, this heuristic is applied in the neighborhood of the two last moves.

The hierarchical order of the heuristics is fixed. A representation of the Fuego original agent can be seen in Figure 3(a), where each symbol represents a different heuristic, and the order of the symbols represent the order that each heuristic will be applied. We created several new agents in the Fuego system by changing the order of the default heuristics of the original agent. Therefore, each agent will give a different priority to the heuristics; which will make each agent have a different playing style (Figure 3(b)). The set of all agents implemented in the system form an *agent database*.

Every time one movement will be generated during the Monte Carlo simulations, one agent is randomly selected in the *agent database* and this agent will be responsible for selecting the movement. Therefore, at each step in the simulation process, a different agent is going to decide the next movement on the board (Figure 4). This approach allows the Monte Carlo method to explore better the search space, using the same amount of computation time. The intuition behind this idea is simple. Although some Go movements, such as the capture of a stone, can seem to be quite strong for a beginner, an experienced player knows that preferring apparently "strong" movements all the time will lead to a poor and unnatural game. Therefore, in order to simulate more realistic Go games, it is necessary to diversify the movement generation process.

However, although we can use 120 different agents, we empirically found out that using all of them does not lead to a stronger player (see Section 4). It is necessary to select a set of agents that effectively lead to better playing abilities. As testing all possible combinations of agents is very expensive, we executed a simple greedy learning algorithm. We start

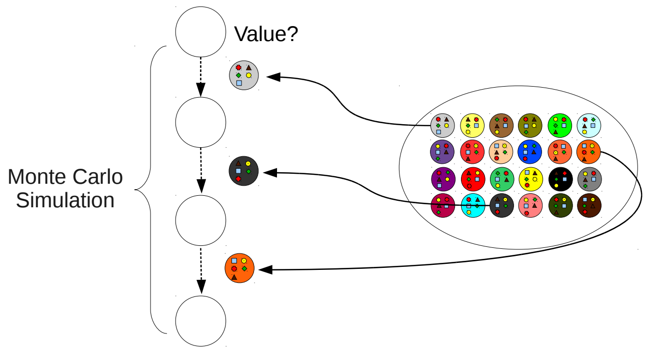

Figure 4: Agent selection in the Monte Carlo simulation process.

with only the original Fuego agent in the database. Then, we perform a series of games against Fuego. The result (percentage of victory) is saved. We then add one more agent in the database. A series of games is again performed against Fuego. If the result is better than the best result found so far, the agent will remain in the database. If the result gets worse, the agent will be removed from the database, and will not be tested anymore. The algorithm proceeds by testing all the remaining possible agents. Note that every time a "good agent" is found, it will be permanently inserted in the *agent database*, and it will be used in all the following iterations of the learning process. Also note that the original Fuego agent will always be in the *agent database*, because it is used in the first iteration.

Therefore, our algorithm is a hill climbing in the space of agent sets: we add one agent to the set and greedily keep it if the new set performs better. We test each agent exactly one time. The most natural way is to generate a random list in which all agents appear exactly once, and follow the order of the list. However, we also manually changed the list in one of our experiments, in order to try to achieve a better solution. As will be seen in the next section, our simple learning algorithm led to a significant percentage of victory against Fuego, showing that Multi-Agent Monte Carlo Go can effectively be used to create stronger players.

4. RESULTS

In this section we are going to present the experiments performed to validate our approach. They were all executed in a 9x9 board, with the same time limit for both our system and Fuego. We used Fuego's default time limit and default configuration for the number of playouts per leaf (1 playout per leaf, for a 9x9 board). We executed 500 games with our system playing as White, and 500 games with our system playing as Black, giving a total of 1000 games per configuration. The experiments were executed in a cluster of Intel(R) Xeon(R) CPU E5530, at 2.4GHz and with 24GB of RAM. Note that our algorithm is not parallel, but we used a cluster in order to distribute the execution of the 1000 games, decreasing significantly the time necessary to run the experiments. The cluster used is part of the InTrigger [3] platform, a cluster of more than 13 clusters distributed across Japan. They are intended to be used for information technology research, both for system software and for large scale data

[3]http://www.intrigger.jp

N	AC	AD	L	P
AD	N	AC	P	L
AD	N	P	AC	L
AD	AC	P	N	L
N	AC	P	L	AD

Table 1: Selected *agent database*.

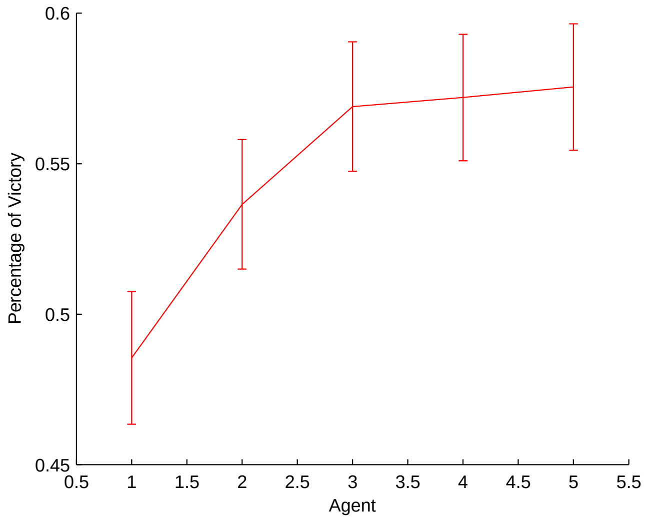

Figure 5: Percentage of victory for the selected *agent database*.

processing researchers.

We first ran our algorithm with all the possible 120 agents. It led to a relatively low percentage of victory: 41.20% (±2.10%). After performing several experiments with the database, we found out that some agents seemed to decrease, while other agents seemed to increase the percentage of victory. Therefore, we created a simple learning algorithm, that tries to add each agent in the database, and tests if it increases or decreases the strength, as described in the previous section.

First, we are going to show our results when the order in which each agent is tested is random. Let N, AC, AD, L and P be the Nakade, Atari Capture, Atari Defense, Lowlib and Pattern heuristics, respectively. The *agent database* selected by the learning algorithm is represented in Table 1, where each line defines one agent and the columns defines the order in which each heuristic is attempted. The first line corresponds to the original Fuego agent. Our algorithm was able to find a set of 5 agents that seems to increase playing strength.

The result obtained with the addition of each new agent can be seen in Figure 5. As can be observed, from a 48.55% (±2.20%) percentage of victory with only Fuego's original agent, with 5 agents we could achieve 57.55% (±2.10%), a gain of 9.00%. Therefore, our strategy seems to be effective into improving the strength of Computer Go algorithms. We performed a $t-test$ analysis that showed that the result with 5 agents is better than the result with only Fuego's original agent with 99% of confidence.

The result of about 48% when our system has only the Fuego original agent is a little bit different from the theoretically expected 50%. We believe this might happen because the game with only one agent is not really "Fuego" vs. "Fuego", it is "Fuego" vs. "Fuego with a small overhead", as the algorithm for agent selection and agent execution is still there, and it is ran in every step of the Monte Carlo simulations. As the number of simulations is very high, we believe this overhead might be responsible for the 48.55% result, instead of 50%.

Agent Number	Percentage of Victory
0	48.50% ± 2.20%
5 (α)	52.85% ± 2.15%
6 (β)	53.60% ± 2.15%
64	57.30% ± 2.15%
70	29.60% ± 1.90%

Table 2: Percentage of victory for each individual agent.

We also executed games with our system running with a single agent (again, against Fuego). In each execution, we used one of the agents that were selected for the *agent database*, but only that one. The objective of these experiments is to see if the result of the agents as a group is better than the result of each individual agent. We can see the percentage of victory obtained for each agent in Table 2, where the Agent Number represents the position of the agent in the list (or, in other words, the number of the iteration in which the agent was tested). We called agent 5 as α and agent 6 as β because they are going to appear again in our next experiment.

Many interesting observations can be drawn from these experiments. First, as can be seen, the result of the group (57.55%) was better than the result of each individual agent, though the difference between the group and the agent 64 is quite small. However, even before adding agent 64, the group already performed quite well (56.90%), a percentage of victory higher than each member. Second, agent 70 is clearly much weaker than the other agents, but when it was added in the *agent database* the result improved 0.35%, instead of decreasing. Therefore, it seems that there is a group phenomena that makes the algorithm stronger.

The learning graph of our algorithm can be seen in Figure 6. After adding agent 5 and 6, the system fluctuates, and is able to escape from the local minimum (lack of improvement) only with the addition of agent 64. After adding agent 70, the system fluctuates again and is not able to find a better solution.

We ran our algorithm a second time, but now we tested the agents in a different order. Before we developed our learning algorithm, we had a list of 15 agents that we believed to be strong (by intuition and trial an error experiments), and we moved those agents to the beginning of the list. Our original intention, when we developed the learning algorithm, was to test this set of agents. The rest of the agents followed the same order as the previous experiment. The agents that compose the new solution found by the learning algorithm can be seen in Table 3. The result obtained with the addition of each new agent is represented in Figure 7, and the learning graph can be seen in Figure 8. This time, we found a slightly better result, of 59.15% (±2.10%).

We executed games with our system running with a single agent. The percentage of victory obtained for each agent can be seen in Table 4. The Agent Number of each agent is

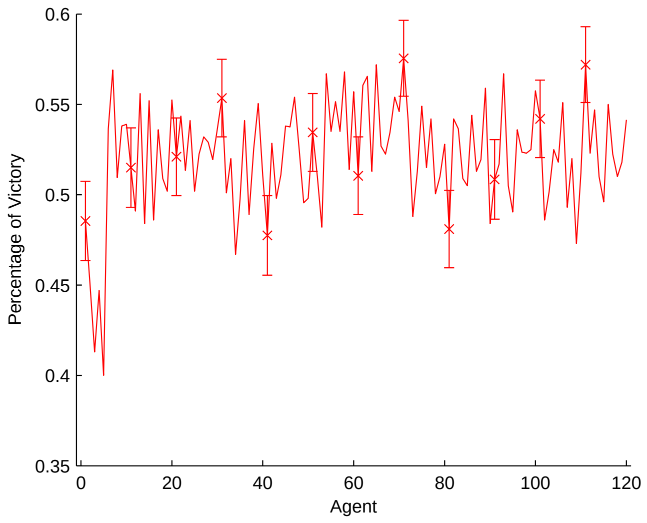

Figure 6: Learning graph, as the algorithm tries to add each agent in the database.

N	AC	AD	L	P
AD	N	AC	P	L
AD	N	P	AC	L
AD	P	L	N	AC
AC	N	AD	L	P

Table 3: Selected *agent database*, in the not random order.

different than last time, as the order changed, but agent 1 and 2 are the same as agent 5 and 6 of the last experiment, respectively. Therefore, we named them α and β. Again, the result of the group was better than the result of each individual agent (although the difference between the group and agent 3 is small). This time, the difference between the group and the best agent seems to be higher than in the previous experiment. And, for the second time, agents that are weaker were able to increase the percentage of victory when they were added to the group. Agent 42 had a percentage of victory of only 50.90%, but was able to increase the percentage of victory of the system in 1.20% when it was added in the group. Therefore, with this new agent order, we were also able to show that we can increase the strength of Monte Carlo Go using the emergent behavior of a group of agents, this time with a slightly better result.

As can be seen, we could find two agent sets that perform quite well against the original Fuego. After analyzing the result of our experiments, we think we have strong indications that the emergent behavior of a group of agents can

Agent Number	Percentage of Victory
0	$48.55\% \pm 2.20\%$
1 (α)	$54.40\% \pm 2.15\%$
2 (β)	$54.55\% \pm 2.15\%$
3	$57.05\% \pm 2.15\%$
42	$50.90\% \pm 2.20\%$

Table 4: Percentage of victory for each individual agent, in the not random order.

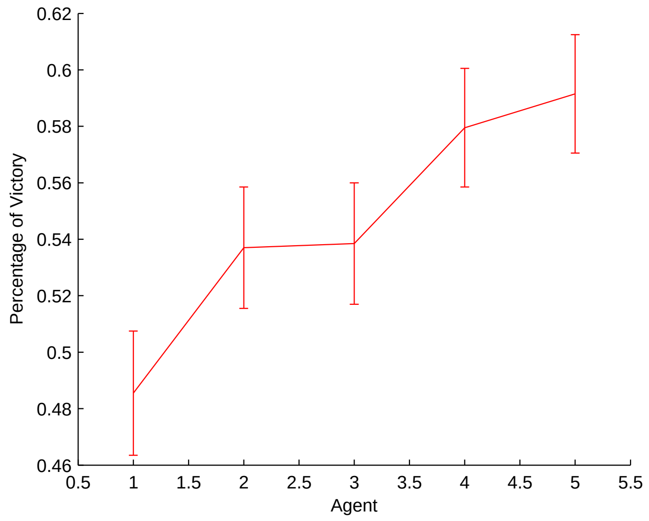

Figure 7: Percentage of victory for the selected *agent database*, in the not random order.

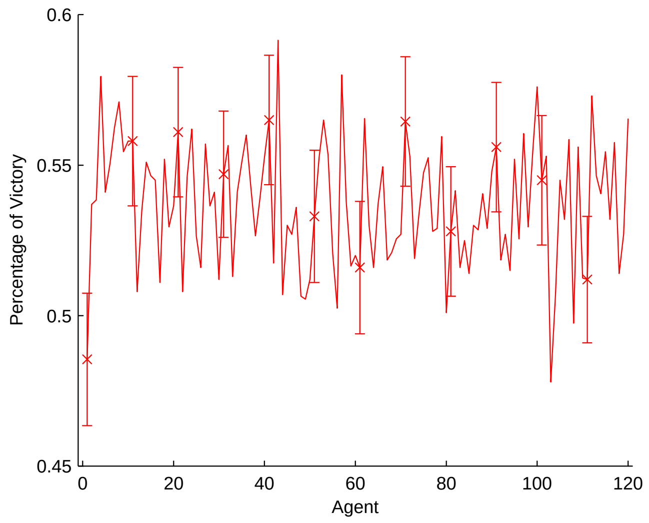

Figure 8: Learning graph in the not random order, as the algorithm tries to add each agent in the database.

lead to higher quality simulations, creating stronger players. It is notable that we could obtain a percentage of victory of around 59% against Fuego, in its default configurations for time limit and number of playouts per leaf.

5. DISCUSSION

In this paper we opened a new path for Computer Go: emergent behavior. In our approach, different agents play in the simulation phase of UCT Monte Carlo Go, which allows a greater diversity, increasing the quality of the simulations, and of the artificial player as a whole. It is possible to argue that other MCTS programs also have emergent behavior, as intelligent game play emerges from a playout strategy executed repetitively by a single agent. However, this work is the first to put Multi-Agent Systems and emergent behavior into perspective, showing new paths that can be explored to improve the current algorithms.

We could not achieve a significant percentage of victory against Fuego using the set of all possible 120 agents. However, we noticed that a selected set of agents could effectively improve the solution, and overcome Fuego. This inspired us to create a simple greedy learning algorithm, that tests if the presence of each agent contributes to improve the strength or not. With this algorithm, we could find a set of agents that won about 59% of the games. In the not random order, the first agents that the algorithm tried were already known to be good, and they were immediately selected. However, we had a set of 15 agents that we believed to be strong (when all of them were in the *agent database*, we obtained a percentage of victory of about 54%), and we were surprised when the learning algorithm reduced this set to only 5 agents. And also, the learning process increased the percentage of victory of our system in about 5%, compared to the solution that we could find manually. Therefore, it had a significant impact in our results.

However, even though we could significantly overcome Fuego with our agent set, it is still not so clear if the group performs better than the best agents, as the difference between them was small. As the number of possible combinations of agent sets is quite high, we believe there might be agent sets that perform even better, and might clearly overcome the best agents. Therefore, it is necessary to develop better algorithms for finding strong agent sets.

We believe that our approach is in a good direction to improve MCTS. However, even our straight $O(n)$ learning algorithm, executing in 104 cores, takes about 120 hours to finish. This happens because it is necessary to perform a great number of games in order to reach stable results, with low standard deviations. With the problems of sharing a cluster, like system maintenances, queues, machine reservation schedules, jobs being killed, etc, the whole execution took about one week and a half. Therefore, finding good agents is a difficult, computationally intensive problem.

Even though, we believe that much can still be discovered in that direction. An immediate future work is to regenerate the random list of agents, and run the learning process again. Would it select a similar set of agents? Could it discover an even stronger group? Another question that should be answered is the effect of adding not one agent to the database, but a set of agents. In other words, does each agent by itself contribute to the solution or is there improvement only when a specific set of agents are all together in the database? If so, how can that set be found? It is impossible to test all combinations of agents. One idea could be to apply an algorithm like simulated annealing, and accept agents that decrease the solution, in order to escape local minima. In our experiments we could perceive that agents that perform bad individually are able to increase the quality of a certain set, so the effect of one agent might depend on the presence of other agents in the group. Unfortunately, it does not seem to be possible to apply learning algorithms like the evolutionary methods, due to the high cost of testing each solution.

Another possible future research path is to study how to apply Multi-Agent System paradigms in different ways. Our system employs a great number of agents during the simulations that are executed to evaluate the score of the leafs. It is possible to experiment with different applications of the paradigm. For example, what if different programs negotiate about a single move, like in [25]? How can we know which is the best movement among the ones suggested? In the case of Shogi the number of possible movements is more limited, and the convergence seems to be easier than in Go, allowing the application of simple majority voting algorithms. With the range of different possibilities allowed in a Go game, how can we solve the selection problem? Other possible direction is to try to apply Multi-Agent Systems in the tree search phase. Which algorithms could be applied? What benefits could we obtain? As can be seen, there is a great range of ideas and algorithms that can be inspired by this work.

6. CONCLUSION

In this paper we present a new paradigm to the state of the art of Go: Multi-Agent Monte Carlo Go. In our approach, different agents play in the simulation phase of UCT Monte Carlo Go, increasing the realism and the quality of the simulations by their emergent behavior. We could not achieve a significant result with all possible agents, but after selecting a good set of agents by a learning algorithm, we could significantly overcome the original system, Fuego. Therefore, we effectively increased the strength of UCT Monte Carlo Go. We present several discussions about our system, including directions for further improvement and points that should be better studied. We believe that our work presents a new paradigm for Monte Carlo Go, and it can be used as inspiration to a variety of different works.

We plan to further explore the research possibilities that were discussed in this paper. Therefore, our future work includes better exploring how to automatically learn good sets of agents. After running again our learning algorithm, but this time with different random agent lists, we plan to explore the effect of probabilistically accepting agents that decrease the solution, like simulated annealing algorithms. Other directions of research can be also explored, for example, discovering solutions of how to select the best move when different programs cooperatively play Go. We believe that much can still be researched, and Computer Go can be greatly improved by exploring Multi-Agent System techniques.

Acknowledgment

The authors would like to thank the Japanese Ministry of Education, Culture, Sports, Science and Technology for supporting this project.

7. REFERENCES

[1] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[2] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47:235–256, May 2002.

[3] D. B. Benson. Life in the game of go. *Inf. Sci.*, 10(2):17–29, 1976.

[4] M. Boon. A pattern matcher for goliath. *Computer Go*, 13:13–23, 1990.

[5] B. Bouzy. *Modelisation cognitive du joueur de Go*. PhD thesis, Université Paris, 1995. (in French).

[6] B. Bouzy and T. Cazenave. Computer go: an ai oriented survey. *Artificial Intelligence*, 132:39–103, 2001.

[7] B. Brugmann. Monte carlo go. Technical report, Physics Department, Syracuse University, 1993.

[8] T. Cazenave. *Systeme d'Apprentissage par Auto-Observation. Application au Jeu de Go*. PhD thesis, Universite Pierre et Marie Curie, 1996. (in French).

[9] T. Cazenave and N. Jouandeau. A parallel monte-carlo tree search algorithm. In *CG '08: Proceedings of the 6th international conference on Computers and Games*, pages 72–80, Berlin, Heidelberg, 2008. Springer-Verlag.

[10] G. M.-B. Chaslot, M. H. Winands, and H. J. van den Herik. Parallel monte-carlo tree search. In *Proceedings of the 6th International Conference on Computer and Games*. Springer, 2008.

[11] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. In *European Conference on Artificial Life*, pages 134–142, 1991.

[12] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. J. van den Herik, P. Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[13] M. Enzenberger and M. Müller. Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. Technical report, University of Alberta, Dept. of Computing Science, TR09-08, April 2009.

[14] M. Enzenberger and M. Müller. A lock-free multithreaded monte-carlo tree search algorithm. In *Advances in Computer Games 12*, 2009.

[15] K. J. Friedenbach, Jr. *Abstraction hierarchies: a model of perception and cognition in the game of go*. PhD thesis, University of California, Santa Cruz, 1980.

[16] S. Gelly, J.-B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. On the Parallelization of Monte-Carlo planning. In *ICINCO*, Madeira Portugal, 2008.

[17] S. Gelly, Y. Wang, R. Munos, and O. Teytaud. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA, France, November 2006.

[18] H. Kato and I. Takeuchi. Parallel monte-carlo tree search with simulation servers. In *13th Game Programming Workshop (GPW-08)*, November 2008.

[19] H. Kato and I. Takeuchi. Running "zen" on computer clusters. In *14th Game Programming Workshop (GPW-09)*, November 2009. (in Japanese).

[20] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, August 2002.

[21] L. S. Marcolino and L. Chaimowicz. No robot left behind: Coordination to overcome local minima in swarm navigation. In *Proceedings of the 2008 IEEE International Conference on Robotics and Automation*, pages 1904–1909, 2008.

[22] L. S. Marcolino and L. Chaimowicz. Traffic control for a swarm of robots: Avoiding group conflicts. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1949–1954, October 2009.

[23] L. S. Marcolino and L. Chaimowicz. Traffic control for a swarm of robots: Avoiding target congestion. In *Proceedings of the 2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1955–1961, October 2009.

[24] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory*. PhD thesis, Zürich, 1995. (in French).

[25] T. Obata, T. Sugiyama, K. Hoki, and T. Ito. Consultation algorithm in shogi: Can a set of players create a single strong player? In *14th Game Programming Workshop (GPW-09)*, November 2009. (in Japanese).

[26] T. Oguri and Y. Kotani. Move decision method based on sds. In *14th Game Programming Workshop (GPW-09)*, November 2009. (in Japanese).

[27] S. E. Page. *The Difference: How the Power of Diversity Creates Better Groups, Firms, Schools, and Societies*. Princeton University Press, January 2007.

[28] C. W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *Proceedings of the 14th annual conference on Computer graphics (SIGGRAPH 87)*, pages 25–34. ACM Press, 1987.

[29] Y. Soejima, A. Kishimoto, and O. Watanabe. Root parallelization of monte carlo tree search and its effectiveness in computer go. In *14th Game Programming Workshop (GPW-09)*, November 2009. (in Japanese).

[30] T. Sugiyama, T. Obata, H. Saito, K. Hoki, and T. Ito. Consultation algorithm in brain game - a move decision based on the positional evaluation value of each player. In *14th Game Programming Workshop (GPW-09)*, November 2009. (in Japanese).

[31] D. Wolfe. Go endgames are pspace-hard. *More Games of No Chance*, pages 125–136, 2002.

[32] S.-J. Yen, C.-W. Chou, S.-C. Hsu, J.-C. Chen, and T.-N. Yang. Improvement of mcts in computer go. In *14th Game Programming Workshop (GPW-09)*, November 2009.