Leandro Soriano Marcolino
Advisor: Hitoshi Matsubara

Multi-Agent Monte Carlo Go

Thesis presented to the School of Systems Information Science at Future University Hakodate as a partial requirement for the attainment of the degree of Master in Systems Information Science.

Future University Hakodate

Hakodate, August, 2011

To my parents.

Acknowledgments

Life is like a Go board, where the stones interact in complex ways. One stone is weak and powerless and cannot do anything else beyond standing alone and clueless in the universe. But when the stones interact, when they dance across the board, something magic appears. Some stones connect to us, giving us power to live. Other stones are more distant, but permit us to reach places that we could never imagine we would be able to reach. Some stones seem to be enemies and hit us in the head, or in the shoulder, but by doing so they force us to extend, to connect, to seek for places to live and grow. The stones are nameless in a Go board, and we merely give names to how they connect and relate to each other: one-point jump, bamboo, two-point jump, knight, long knight... In life each stone has an unique name, but we also have names to their special connections: family, teachers, friends, girlfriends, sponsors... And here am I, a small stone floating aimlessly in the Universe, I cannot even know if I am Black or White, but I am given this unique opportunity in time to look above and contemplate all the stones that have been so important to me in these two years, almost like I could for a single instant contemplate the thoughts of The Great Player.

A battle happened in these two years, and I had to grow in order to live. A battle happened in these two years, and here I am now, a small stone contemplating perplexed the product of its own research, almost like it was not really done by me, and indeed I would never be able to do it alone. I do not know if I am Black or White, if I will win or lose. Maybe in life there is not a loser nor a winner, maybe life is more like The Great Player playing against Himself, watching amazed how His stones can dance with harmony and beauty. And me, a small stone floating aimlessly in the Universe, cannot do anything else beyond feeling gratitude and appreciation for all the ones that are so important to me: my family, my teachers, my friends, my girlfriend, my sponsor...

The end is nothing; the road is all. (Willa Cather)

Abstract

Go is a strategic board game that is considered one of the greatest challenges for Artificial Intelligence. Many algorithms have been proposed, trying to tackle this problem, but generally all of them generated players that could be easily defeated by a strong human opponent. UCT Monte Carlo Go is one of the most successful algorithms. The basic idea is to associate a tree search with pseudo-random simulations, used to evaluate the leaves. Nowadays, the literature is more focused on how to parallelize the UCT algorithm, producing a stronger player by increasing the computational power. However, there is a limit in the improvement that can be obtained. In this thesis we focus on how to improve the algorithm itself, producing a stronger player with the same amount of computation. We propose a Multi-Agent version of UCT Monte Carlo Go. The emergent behavior of a great number of agents have been successfully applied in the literature to deal with a great variety of problems. In this thesis, we use emergence and diversity 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. However, we found that using all possible agents leads to a weaker player. Therefore, we developed two learning algorithms, in order to find a set of agents that can effectively generate a strong player. The first learning algorithm is a simple hill-climbing approach, it tries each agent one time and the agents that can improve the solution remain. The second learning algorithm is a simulated annealing approach. At each iteration, it decides randomly if it is going to add a new agent, or remove one from the database. The agent to be added/removed is selected randomly. Modifications that decrease the current solution might also be accepted. With either learning algorithm, we could significantly overcome Fuego, a top Computer Go software. Emergence seems to be the next step of Computer Go development.

Keywords: Emergent Behavior, Collective Intelligence, Multi-Agent Systems, UCT Monte Carlo Go

概要

囲碁は戦略的なボードゲームである。現在、囲碁は人工知能の難題である。この問題を解くためのたくさんアルゴリズムが存在するが簡単に強い人間に負ける。「UCTモンテカルロ碁」と言うアルゴリズムは現在の一番強いアルゴリズムである。このアルゴリズムは探索木に擬似乱数を用いて葉の評価を行う。現在、多くの研究は「UCTモンテカルロ碁」の並列について集中している。コンピュータの処理能力が上がれば、強いプレーヤーを作ることが可能であるが並列処理には限りがある。そこで、本論文は「UCTモンテカルロ碁」アルゴリズムの改良について注目している。同じ処理能力用いると、もっと強いプレーヤーを作る事ができた。本論文はマルチエージェント的な「UCTモンテカルロ碁」を示しマルチエージェントの創発的な行動を使い「UCTモンテカルロ碁」のシミュレーションの質を改良、人工知能のプレーヤーの強さを強化した。シミュレーションを行う際は、一人のエージェントが打つだけではなく様々なエージェントが打つようにした。こうすることで、処理に負荷を大きくかけることなく探索空間を探すの効率を上げる事できる。しかし、実験的にすべてのエージェントに適応した場合、プレーヤーが弱くなると分かった。そのため、強いプレーヤーを作られるエージェントセットを見付けるために二つ学習アルゴリズムを作った。一番目は簡単な山登り法で、各エージェントを1回加えて見る。結果が増加させたエージェントだけエージェントデータベースに残る。2番目は焼き鈍し方である。各繰り返し毎に確率的にエージェントを加えるとか外すとか決める。加えられた／外さられたエージェントは確率的に決定される。結果を減少さられた変化も確率によって受け入れることができる。どちらの学習アルゴリズムでも、「Fuego」に有意に勝つことができた。「Fuego」は優秀コンピューター碁のソフトである。創発的な行動はコンピューター碁の次のステップであると考えている。

キーワード：創発的な行動、集団的知性、マルチエージェントシステム、UCTモンテカルロ碁

添削者：佐々木啓太

Abstrakto

Goo estas strategia tabulludo tio estas konsiderita unu el la plej grandaj defioj por Artefarita Inteligenteco. Multaj algoritmoj estas proponita en la literaturo, provis pritrakti kun ĉi tiu problemo, sed ĝenerale ili ĉiuj produktis ludistojn ke povus esti facile venkita de forta homa oponanto. Unu el la plej sukcesa algoritmo nomiĝas UCT-a Montekarla Goo. La baza ideo estas unuiĝi arbon serĉon kun pseŭdohazardaj simuloj, uzata por taksi la foliojn. Nuntempe, la literaturo estas pli koncentrata en la maniero laŭ kiu paraleli la UCT-a algoritmo, por produkti pli fortan ludiston apud pli komputada potenco. Tamen, en ĉi tiu tezo ni decidis koncentriĝi pri kiel plibonigi la algoritmon sin, produktantas pli fortan ludiston kun la sama kvanto de komputado. Ni proponas Multagentan version de UCT-a Montekarla Goo. La elapereca konduto de granda nombro de agentoj estas sukcese apliki en la literaturo al pritrakti grandan varion de problemojn. En ĉi tiu tezo, ni uzas elaperecon kaj diversecon pliigi la kvaliton de la Montekarla simuloj, pliiganta la forton de la artefarita ludisto. Anstataŭ unu agento ludi kontraŭ sin, malsamaj agentoj ludas en la simula fazo de la algoritmo, kiu estigi pli bona esplorado de la serĉa spaco. Tamen, ni eltrovis kiu uzanta ĉiuj eblaj agentoj estigas pli malforta ludisto. Sekve, ni disvolviĝis du lernantaj algoritmoj, por trovi aron da agentoj kiu povas efike produkti fortan ludiston. La unua lernanta algoritmo estas simpla montetogrimpa metodo, ĝi provas ĉiu agento unun fojon kaj la agentoj kiu povas plibonigi la solvon postrestas. La dua lernanta algoritmo estas Simulata Malharda metodo. Ĝi elektas agenton provi hazarde, kaj ĝi akceptas agentojn ke malpliigi la solvon kun iu procento. Kun iu ajn el la lernantaj algoritmoj, ni povus signifoplene venki Fuegon, pintan Komputilan Goan softvaron. Elapereca konduto ŝajne estas la sekvonta paŝo de Komputila Goa evoluado.

Ŝlosilvorto: Elapereca Konduto, Kolektiva Inteligenteco, Multagenta Sistemo, UCT-a Montekarla Goo

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"I look at a Go board, and I see the whole universe"

1.1 Motivation

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 on 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 [Wolfe 2002]. According to Allis [1994], 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.

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

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. As can be seen in Chapter 2, the benefits of the parallelization saturate fast in the works that can be found in the literature. The increase of strength is far from linear and it stops with a small number of server or threads.

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.

1.2 Monte Carlo Techniques

The strength of Computer Go algorithms improved significantly with the development of Monte Carlo Techniques. The basic idea is to perform a great number of game simulations in order to evaluate a given board state. The simulations start at the current game position, that we want to evaluate, and end in the final game position. They are random, but are heuristic-driven, in order to simulate realistic Go games. Nowadays, this approach is

combined with a tree search phase. In a tree search, many different move options are evaluated, and their possible countermoves, and the moves that follow, etc. However, it is not possible to evaluate all the possible moves, from the current position to the end of the game, because the number of possibilities is too large. Therefore, the tree search stops in certain positions, called leaves. When a leaf is visited for the first time, it is evaluated by one (or more) Monte Carlo simulation(s). This idea led to significant improvements in the playing strength.

1.3 Emergence and Diversity

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* [Colorni et al. 1991], *Particle Swarm Optimization* [Kennedy and Eberhart 2002], etc, in order to solve difficult optimization problems. It is also notable how emergence can lead to complex and intricate group behavior [Marcolino and Chaimowicz 2008; 2009a;b, Reynolds 1987].

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 by chemical scent from larvae, other ants, intruders, food, waste, etc, and they leave chemical markers 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 1.1). However, emergence is generally not a clear concept. In this thesis we define emergence as a great number of simple interactions that occur in a system, leading to a complex result.

Figure 1.1: Water crystals, formed by a natural emergent process (taken from `www.wikipedia.org`).

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 [Page 2007].

1.4 Contributions

In this thesis, we significantly improve the strength of the current state of the art Computer Go algorithms. We do not focus on parallelization, therefore the improvement is obtained by better algorithms, and not by higher computational power. We modify the state of the art algorithm with a Multi-Agent System approach, exploring the concepts of emergence and diversity. Therefore, we offer a new paradigm for the exploration of Computer Go.

We can visualize the Monte Carlo evaluations as one agent that repetitively plays against itself using a playout strategy (heuristics). 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 thesis 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 describe our approach. Each agent has a different playing style. Therefore, by using many agents during the simulation process, we are also exploring the concept of diversity, but in a multiagent context.

We modify Fuego [Enzenberger and Müller 2009a], 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 improve Fuego, and produce a stronger Computer Go program. A shorter version of this work was published in Marcolino and Matsubara [2011], and was selected as a best paper nominee for the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011).

1.5 Organization of this Work

This work is organized as follows: in the next chapter we present some related work. We will talk about the classical approaches for Computer Go, the Monte Carlo method and the parallelization of the current algorithms. Next, we present important concepts, such as Emergence, Stigmergy and Diversity, and we comment on how they can be applied for improving the current approaches. We also give a quick overview of the Hill-Climbing and the Simulated Annealing algorithms, as they are going to be used later in our methodology. In Chapter 3, we present our proposed algorithm. In our approach, it is necessary to select a set of agents, in order to obtain a good solution. Therefore, a learning process is necessary, and we will present two different approaches: a Hill-Climbing and a Simulated-Annealing

based one. In Chapter 4 we present the results obtained by the two approaches. Chapter 5 discusses the significance of our results, as we believe that this work introduces a new paradigm for Computer Go development. The context in which we use a multi-agent system can be polemical, and we try to address the most common arguments that could be developed against our ideas. We also present interesting possibilities for future research. Finally, Chapter 6 presents the conclusion of this thesis.

Chapter 2

Background

"To see a world in a grain of sand
And a heaven in a wild flower,
Hold infinity in the palm of your hand
And eternity in an hour."
(*William Blake*)

Generally, the basic idea for board games is to evaluate all possible moves, and all possible responses for all possible moves, and all the possible counter-responses, and so forth. This process is called a tree search. However, it is generally not possible to do this evaluation from a certain position until all possible ends of a game, because the different possibilities grow exponentially. Therefore, the tree search has to stop in certain positions, called leaves. In order to evaluate the leaf, it is necessary to design an evaluation function, because it generally will not be an end-game situation. When evaluating the tree in order to choose a movement, the computer assumes that its opponent will always choose his best possible move ("minimax assumption" [Von Neumman 1928]). One of the most successful versions of this algorithm is called an alpha-beta search [Hart and Edwards 1963]. However, it does not work well for Computer Go, because of mainly two problems: the number of different move options in a given board position is huge; and it is difficult to design a good evaluation function [Bouzy and Cazenave 2001]. Therefore, the game is very difficult for

computers, and current software is weak. In this chapter, we are first going to take a look at the classical approaches for Computer Go. Then, we are going to describe a new algorithm that was able to increase dramatically the strength of the state of the art: Monte Carlo Go. We will introduce the current focus of the literature, which is how to parallelize the Monte Carlo method. Then, we are going to talk about how the strength of Computer Go could be improved without increasing computational power, by introducing three concepts: emergence, stigmergy and diversity. We will then take a quick look at two optimization algorithms, as they will be useful later: Hill-Climbing and Simulated Annealing.

2.1 Classical Approaches for Go

A great variety of approaches have been proposed in the literature in order to tackle the complexity of Go. The problem is too difficult for a conventional alpha-beta search, forcing the researchers to try many different methods. An interesting survey of the literature can be found in Bouzy and Cazenave [2001]. According to the survey, the history of Computer Go can be divided in three generations. The first Go programs used the idea of influence: a stone has an influence in its intersections, which decreases with distance [Zobrist 1969]. The second generation developed abstract representations for the board, and reasoned using group of stones [Friedenbach 1980, Wilcox 1985]. This generation was the first to play better than an absolute beginner. The third generation moved on to apply patterns intensively in order to recognize typical situations [Boon 1990]. Generally a classical Go program uses a combination of all these techniques. Other important approaches that have been explored include learning [Cazenave 1996], cognitive modeling [Bouzy 1995] or combinatorial game theory [Müller 1995].

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 [Benson 1976]. Some programs appeared that, instead of actually playing the full game, focused on how to solve local problems. One of the most famous is Thomas Wolf's Gotools, that solves life and death problems (*Tsume-Go*) [Wolf 1994; 2000].

Other works reduced the complexity not by focusing on local situations, but by reducing the size of the board itself. The game is solved for a 5x5 board in van der Werf et al. [2003]. However, the complexity increases exponentially with the size of the board, and the 9x9 board is far from being solved.

2.2 Monte Carlo Go

As we saw, many approaches have been tried for Go, but they all failed to build a strong player. 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 [Brugmann 1993]. Later, 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 Coulom [2006]. The state of the art was further advanced by the UCT Monte Carlo algorithm [Gelly et al. 2006], which contributed with significant improvements in playing strength. The proposed program, MoGo, won all the tournaments on the international Kiseido Go Server[1] in October and November 2006.

In this section, we are going to introduce UCT Monte Carlo Go, as our contribution is an improvement of this approach. 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 Auer et al. [2002], a simple algorithm is proposed, 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

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

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 Auer et al. [2002], a slightly more complicated algorithm is also introduced, called UCB1-TUNED, that had better experimental results. First, they calculate an estimation of the upper bound on the variance of arm j, by:

$$V_j = \left(\frac{1}{T_j(n)}\sum_{y=1}^{T_j(n)} X_{j,y}^2\right) - \bar{X}_j^2 + \sqrt{\frac{2\log n}{T_j(n)}} \tag{2.1}$$

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

$$\bar{X}_j + \sqrt{\frac{\log n}{T_j(n)} min\{1/4, V_j\}} \tag{2.2}$$

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 2.2 (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 [Yen et al. 2009].

2.3 Parallelization of Monte Carlo Go

In order to achieve further enhancements, parallel and distributed versions of the game started to appear in the literature. Generally, the idea is to use a great number of machines or processors to increase computation power. According to Chaslot et al. [2008], 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 a single thread. In tree parallelization, many threads execute in a single, shared tree. In Gelly et al. [2008], a straight algorithm for multi-core parallelization is proposed, 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 against the non-parallel version 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 [Kato and Takeuchi 2008]. Recently, a top Computer Go program, Zen, was run on a large cluster of computers [Kato and Takeuchi 2009]. A similar approach is also investigated by Cazenave and Jouandeau [2008], 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 Soejima et al. [2009], 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 on the possible improvement in game strength. In Kato

and Takeuchi [2008], 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 Chaslot et al. [2008], tree parallelization only scaled well up to 4 threads. A lock-free parallelization was proposed by Enzenberger and Müller [2009b], but it could not scale beyond 7 threads.

2.4 Next Steps of Monte Carlo Go

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 Obata et al. [2009], 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 Sugiyama et al. [2009] 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 Oguri and Kotani [2009], 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.

2.5 Emergence, Stigmergy and Diversity

We will start by defining an agent, as it is the most fundamental concept to the following theories. According to Russell and Norvig [2003], an *agent* is anything that senses its environment and, after some computation (that can be very simple, like a look-up table), generates an action in that environment. The most concrete example is a robot, that uses its sensors to perceive (or estimate) the state of the world, and, based on its running program, generates a command to its actuators, that are going to generate an action in the real world. However, we can also think about virtual agents, sensing and acting on

a virtual world; software agents, that receive inputs from keystrokes or files, and act by writing on the screen, sending data to the Internet, etc. An agent can interact with other agents, in a *Multi-Agent System*, and sometimes these interactions make a certain behavior *emerge* in the group. There is no central actor, or coordinator, controlling the emergent behavior, it is simply a product of the decentralized interactions between the agents.

Emergence is a very powerful concept, but it is still not well established and understood. For example, even though Russell and Norvig [2003] is one of the most important books for modern Artificial Intelligence, it does not dwell much on emergence. The concept does appear in a multi-agent context, but tangentially, when the authors give the example of birds that, by following simple rules, fly like a pseudo-rigid body, a behavior known as flocking. It also appears when the authors are talking about robotics, but with a single-agent emphasis. The interplay of simple controllers and a complex environment, can make a desired behavior of a robot emerge. But how can we actually solve problems using emergence?

However, Russell and Norvig [2003] is more focused on single-agent Artificial Intelligence. Important follow-ups, for Multi-Agent Systems and Distributed Artificial Intelligence, are Weiss [1999] and Wooldridge [2001]. Therefore, we would expect that they would dedicate long pages on explaining emergence and how to solve problems using it. That is not what happens, however. They focus more on protocols for agent communication and interaction, on how to build negotiation mechanisms like contracts or bids, how to distribute tasks, etc. The concept of emergence appears in Weiss [1999] only when explaining the combinations of simple controllers in the reactive agent architectures, a single-agent explanation. Wooldridge [2001] does not have "emergence" by itself in its index, but does discuss emergence in a multi-agent context, when explaining about the emergence of social laws. However, it is not yet very clear how to actually solve problems with emergence.

We can narrow-down even more and look at books that talk about Swarm Intelligence, like Engelbrecht [2006b]. The focus is more on large scale multi-agent systems, called *swarms*. Emergence is one of the main concepts in those systems, and the book starts talking about it right in the beginning, in the Introduction. The authors say that the

behavior of the swarm comes from the interactions of the individuals of the swarm over time, and it is usually not easy to know how the swarm will behave, given a set of behaviors for each agent. They define this process as "emergence". They also have a more formal definition, saying that "emergence" is properties (or behaviors) of a given system, that come to existence not because of a coordinated control system, but that appear because of the interactions of the individuals of this system with their local environment. Of course, we can consider the other individuals as part of their local environment. The term *stigmergy* also appears for the first time, being defined as an indirect form of communication between individuals. However, the book then starts to focus more on Optimization, and the authors dwell on Evolutionary Algorithms, Particle Swarm Optimization [Kennedy and Eberhart 2002], and Ant Algorithms [Colorni et al. 1991]. Optimization problems are very important, and a lot of situations in real life can be modeled using them. Therefore, it is very good to see how emergence can be an important concept in solving this kind of problem. However, we should not focus our attention only on optimization problems. Many situations cannot be easily defined as an optimization problem, for example the ones of swarm coordination that we deal with in Marcolino and Chaimowicz [2008; 2009a;b], and some board games, like Go. This does not mean that emergence cannot be used to solve these problems, as we did use it in our proposed solutions for swarm robotics, and, in this thesis, we are using it to propose a way of making Computer Go stronger. But it seems that the literature still tends to look at emergence under the cover of optimization theory.

Some might argue that board games are an optimization problem, where, given a state of the board, we want to find the best possible move. However, given the impossibility of a perfect evaluation of the value of each move, as we cannot evaluate the game until the end in the general case, combined with the strong time constraints of the problem, it is generally not useful to look at it in this way. It is better to apply search algorithms, like the ones that we explained in the beginning of this chapter, instead of optimization algorithms, like *Particle Swarm Optimization*, etc.

We can narrow-down even more, and look at a book about emergence itself [Johnson 2001]. Its target reader is a more general audience. The book is full of examples of

emergence: in biology, like ant colonies and mold; in the organization of our cities and in our brain. However, when it goes to computer software, it does not go far beyond the evolutionary and optimization algorithms that the other works already covered. It presents some examples of softwares that learn through a massive number of users, and "artificial life" programs, like mold simulations made in StarLogo [Colella et al. 2001]. That example makes us remember Conway's Game of Life [Gardner 1970], where the cells follow simple evolution rules, based on the state of their neighboring cells, and complex patterns emerge out of the system. Actually, it seems like John Coway actually used a Go board when he was testing his game for the first time (though we have no reliable source for this information, beyond the Wikipedia article at `http://en.wikipedia.org/wiki/Conway's_Game_of_Life`).

Emergence is a very powerful concept to solve complex problems, and, as we said, we should not restrict our attention only to those problems that can be described in terms of an optimization problem. It is important to go beyond that. However, it is not yet well understood how to use emergence to solve these kinds of problems. We hope this thesis can shed some light in this direction, by showing how an emergent process can improve Monte Carlo Go.

Another important concept is the idea of *stigmergy*, that we briefly mentioned before. Many works on Multi-Agent Systems develop protocols for communication, negotiation and many forms of complex interactions. However, it is sometimes possible to solve complex problems without direct communication between the agents. They might interact in an indirect way, by modifying the environment. As the environment is the input for the agents, and therefore affects their behavior, modifications in the environment are also a way to interact with other agents, and influence their behavior. The term was defined formally for the first time in Grasse [1959]. In the algorithm that we propose in this thesis, there is no direct communication between the agents. The agents perform movements in a Go board, and each agent acts in the Go board that was left by the previous agent. The act of one agent depends on the state of the Go board, and this state was modified by the previous agent. The act the agent selects modifies the state of the Go board (as it

performs a movement), and the state will then influence the act of the next agent, and so on. Therefore, our algorithm is clearly a stigmergic process.

Some social scientists and economists currently believe that teams of diverse people can have strong characteristics for solving difficult problems [Page 2007]. 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 Page's model, each person has a set of heuristics to solve problems. The heuristics are built based on the different perspectives that each person uses to look at problems. By combining different persons, we can have a greater number of perspectives, and heuristics, and therefore we have access to more tools to solve complex problems. In our model, as will be clear in Chapter 3, each agent has the same set of heuristics, but they give to their heuristics a different priority order, which makes them behave in a different way given the same situation. We believe that this is close to Page's model, and that his results might apply in our situation. We can consider, for example, that our set of heuristics, given one specific priority order, is one heuristic in Page's model, and then each of our agents has a different heuristic, but only one. Thinking about ways to approximate our model to Page's is a good resource for future work, as we will discuss in Chapter 5.

Page is able to prove many interesting properties in his model. First, he proves that Diversity Trumps Homogenicity: "If two collections of problem solvers contain problem solvers of equal individual ability, and if those problem solvers in the first collection are homogeneous and those in the second collection are diverse, that is, they have some differences in their local optima, then the collection of diverse problem solvers, on average, outperforms the collection of homogenous problem solvers" (Page [2007]). Next, given some conditions on the problem solvers population, he is able to prove that Diversity Trumps

Ability: "A randomly selected collection of problem solvers outperforms a collection of the best individual problem solvers"(Page [2007]). When talking about predictions, he also proves that "Given any collection of diverse predictive models, the collective prediction is more accurate than the average individual predictions" (Page [2007]). Therefore, Diversity is proven to be a powerful tool, and we should explore it to solve complex problems, for example, to build a strong Computer Go player.

2.6 Optimization Algorithms

In one of the stages of this work, we developed optimization algorithms for selecting a set of agents. Our algorithms can be defined by basically two ideas: *Hill-Climbing* and *Simulated Annealing*. Therefore, as a matter of completeness, in this section we are going to explain and discuss these two ideas.

A hill-climbing algorithm (also called by other names, like local search, gradient descent, etc) is a very simple and efficient algorithm. Any book or reference about optimization or numerical methods should talk or at least mention this algorithm, therefore it is even difficult to choose the best reference. One of the places that introduces the algorithm is Engelbrecht [2006a]. The basic idea is that given a point in the space of possible solutions, we look for a direction that improves our current solution. Generally this direction is the opposite of the gradient of a function that we wish to optimize, so this algorithm is also known as gradient descent. We then make a step (whose length has to be calculated or set previously as a parameter) in that direction, and reach a new point in the space of possible solutions. We apply this iteratively, reaching a better solution at each new iteration. The problem is that in some positions of the solution space, we cannot find a direction to improve the current solution anymore. Any direction that we look only leads to a worse solution. If we are following the gradient, these will be places where the gradient is zero. These places are known as *local minima*. However, these are not, necessarily, the best solution that we can find for the problem. A better solution might exist, but we cannot reach it without decreasing the current solution first. The best possible solution is known

as *global minima*. But how can we find it? Using hill-climbing algorithms, we tend to get stuck in local minima, and not find the best possible solution. However, they are very simple and efficient, and therefore, very useful. Even complex solutions for optimization tend to use a hill-climbing algorithm as the final step to improve a certain solution.

A way to escape local minima is to accept solutions that are worse than the current one. That allows us a wider exploration of the solution space, and we might reach into better solutions in the end. One of the most famous algorithms that follows this approach is the *Simulated Annealing* [Dreo et al. 2005]. The basic idea is to accept solutions that decrease the current one, based on a certain probability, and to decrease this probability with time. The algorithm is inspired by a physical phenomena, called *annealing*. When we want to solidify a material in an organized state (a state with a minimal of energy), the basic physical process is to heat the material, reaching a disorganized liquid state, and to gradually decrease the temperature in a controlled manner. We can simulate this process in order to solve optimization problems, leading to the simulated annealing algorithm. As we accept solutions that decrease the current one, we might find better solutions than with a hill-climbing algorithm. Note that, as the temperature decreases, simulated annealing gradually transforms into a hill-climbing, and in the end, with a very low temperature, it will not accept solutions that decrease the current one anymore, turning itself completely into a normal hill-climbing algorithm.

2.7 Contributions

As we saw in this chapter, Go is a very hard problem, and many different algorithms were proposed in order to deal with it. Currently, the best algorithm is UCT Monte Carlo Go, based on the execution of simulations in order to evaluate a given board position. Nowadays, the literature is emphasizing how to parallelize UCT Monte Carlo Go, in order to obtain better solutions. However, parallelization always has limits, and we believe it is possible to achieve a stronger player without using parallelization, by improving UCT Monte Carlo Go.

We saw then how emergence, stigmergy and diversity can be used to solve difficult problems, although the literature still emphasizes emergence as a technique for optimization algorithms (like Particle Swarm Optimization). 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.

Chapter 3

Methodology

"When many work together for a goal, great things may be accomplished. It is said a lion cub was killed by a single colony of ants." (*Saskya Pandita*)

In this chapter we are going to explain our modifications of UCT Monte Carlo Go. As will be clear later, it is necessary to select a good set of agents, so we will also present two learning algorithms for solving that problem.

We can model the random simulations as one agent playing against itself using its available heuristics (Figure 3.1(a)). In this work, we investigate the effects of having not only one, but several agents playing against each other (Figure 3.1(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 in the hierarchy. The process repeats until a heuristic generates a movement. If no

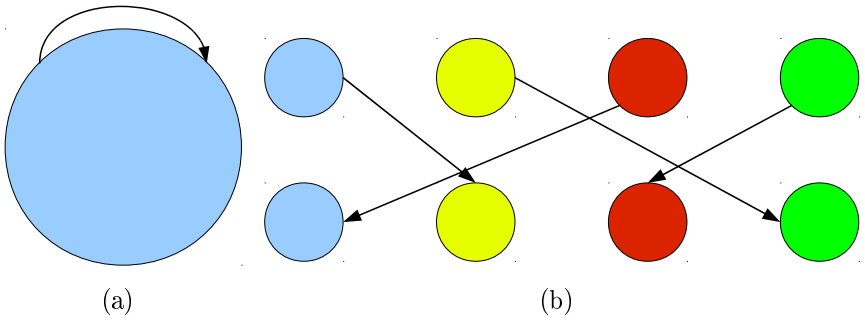
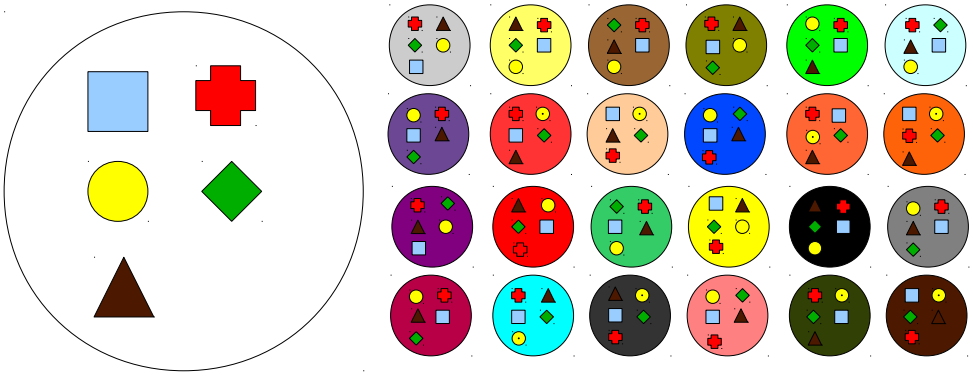


Figure 3.1: Original single-agent Monte Carlo (a) and proposed Multi-Agent Monte Carlo (b). The colors represent different agents, and the arrows represent interaction.

heuristic can generate a movement, a global capture move is attempted. If no move could be generated still, 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 five heuristics: **Nakade** If there is a region of three empty points, generates a movement 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 3.2(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 3.2(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 decides the next movement on the board (Figure 3.3). This approach seems to allow 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,



(a) (b)

Figure 3.2: Original Fuego agent (a) and new *agent database* (b).

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 Chapter 4). It is necessary to select a set of agents that effectively lead to better playing abilities.

3.1 Learning Algorithms

In this section we are going to introduce two learning algorithms that we implemented, in order to find a good set of agents: hill-climbing and simulated annealing. The hill-climbing algorithm is simple, fast and straight-forward, while the simulated annealing algorithm is more complex and slow, in an attempt to obtain better solutions by escaping local minima.

3.1.1 Hill-Climbing

As testing all possible combinations of agents is very expensive, we executed a simple greedy learning algorithm. We start with only the original Fuego agent in the database. Then, we perform a series of games against Fuego. The result (percentage of victory) is

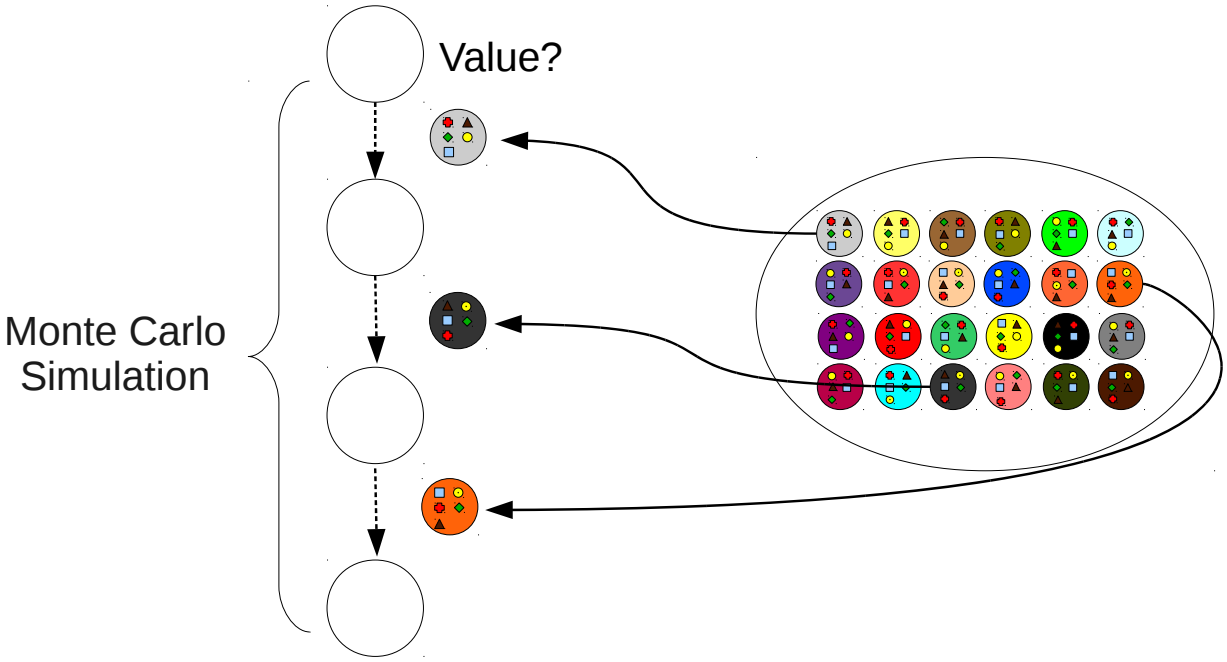


Figure 3.3: Agent selection in the Monte Carlo simulation process.

saved. We then add one more agent to 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. A representation of this algorithm can be seen in Figure 3.4.

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 chapter, 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

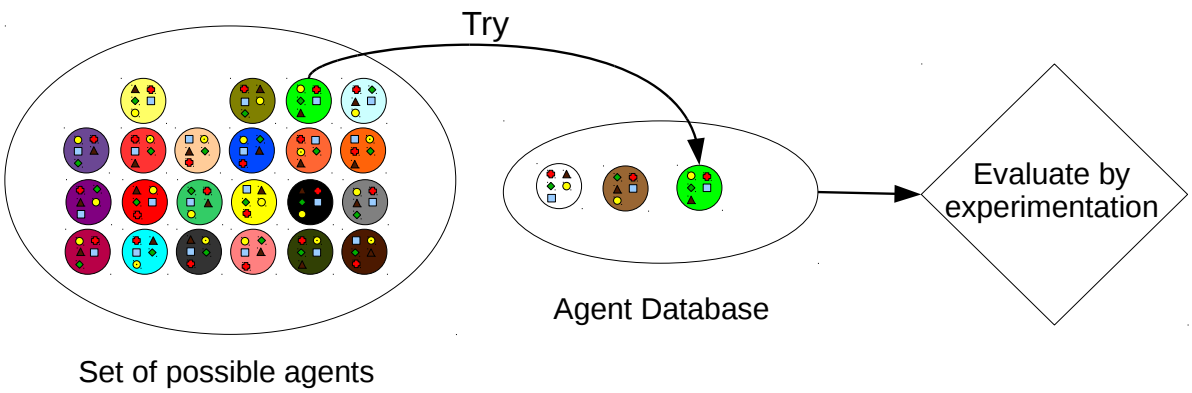


Figure 3.4: A greedy hill-climbing learning algorithm.

stronger players.

3.1.2 Simulated Annealing

As hill-climbing approaches are known to have a local minima problem, we also explored a simulated annealing algorithm. The basic idea is to accept modifications that decrease the percentage of victory, with a certain probability; and to decrease this probability each time a new modification is accepted. Therefore, we can escape local minima by accepting a modification that decreases the current solution, but in the final iterations we want to converge to the best possible solution in the neighborhood. The acceptance of a bad agent might also compromise the solution, so we also desire regressibility in our algorithm. Thus, we consider as a modification not only the addition of a randomly selected new agent, but also the removal of one of the agents that are already in the database. The algorithm, in detail, can be described as follows:

1. Start with an empty *agent database*, temperature t, and a percentage of victory v.
2. Choose an agent randomly to add in the database
3. Evaluate the solution by performing a series of games against Fuego, in order to obtain a new percentage of victory v'

4. If the solution improves, accept the modification.

5. If the solution decreases, accept the modification with the following probability:

$$e^{\frac{v'-v}{t}} \tag{3.1}$$

6. If a solution is accepted, decrease the temperature by:

$$t := \alpha * t \tag{3.2}$$

   And also update the percentage of victory, by $v := v'$.

7. Choose to either add a new agent, with probability ρ, or to remove an agent, with probability $1 - \rho$. If there is only one agent in the *agent database*, it is automatically chosen to add a new agent, not to remove.

   (a) If it was chosen to add an agent, select randomly an agent to add that was not in the *agent database*.

   (b) If it was chosen to remove an agent, select randomly an agent to remove that is in the *agent database*.

8. Go back to Step 3

A representation of this algorithm can be seen in Figure 3.5. In the next chapter, we are going to see the results obtained by the two learning algorithms proposed.

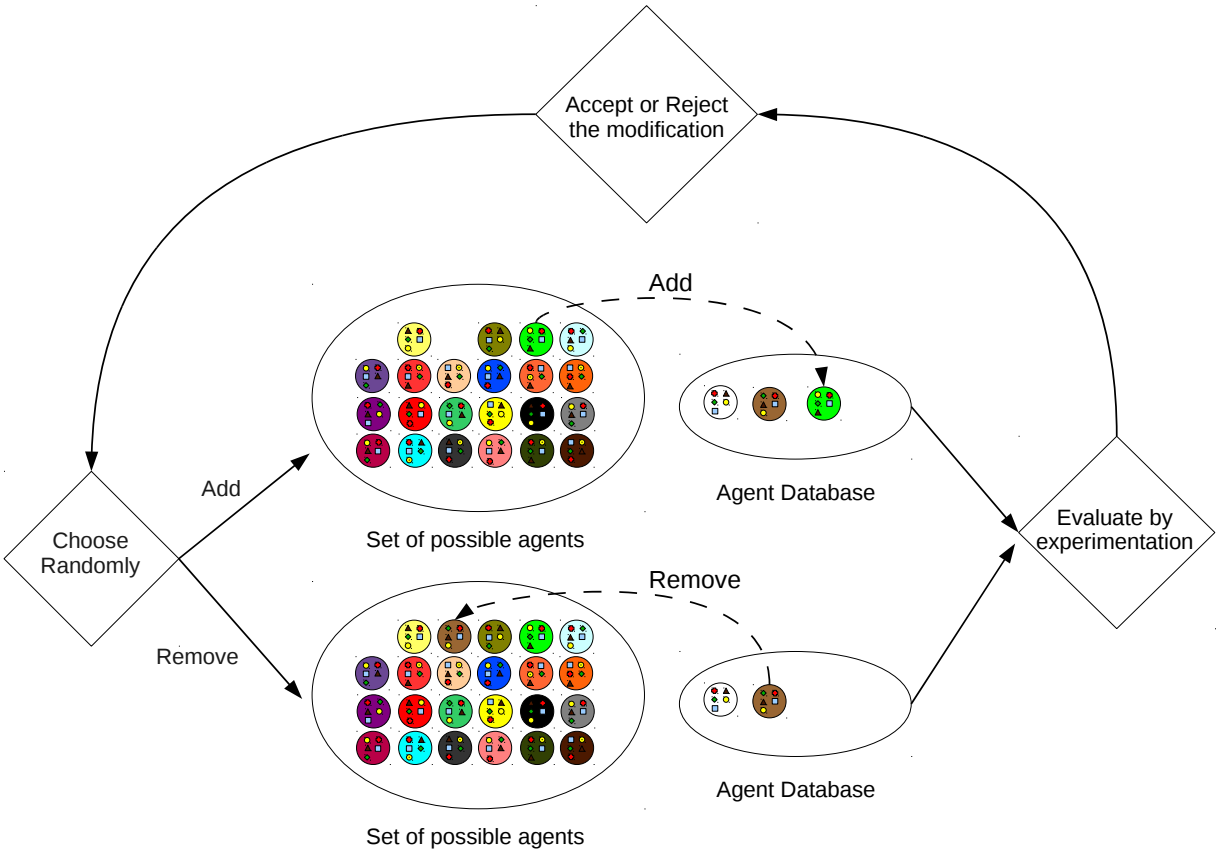


Figure 3.5: A simulated annealing learning algorithm.

Chapter 4

Results

"Then of their session ended they bid cry
With trumpet's regal sound the great result:
Toward the four winds four speedy Cherubim
Put to their mouths the sounding alchemy,
By herald's voice explained; the hollow Abyss
Heard far and wide, and all the host of Hell
With deafening shout returned them loud acclaim."
(*John Milton*, in *Paradise Lost*)

4.1 Hill-Climbing

In this section we are going to present the experiments performed to validate our approach, using the hill-climbing learning algorithm. The experiments performed with the simulated annealing algorithm are going to be presented in the next section. All experiments were executed on 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 [1] platform, a group 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 processing researchers. For reference, the number of Monte Carlo simulations executed were in the order of 10000 games per second.

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 chapter.

First, we are going to show our results when the order in which each agent is tested is random. The *agent database* selected by the learning algorithm is represented in Table 4.1, 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 4.1. 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 at 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% 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

[1] `http://www.intrigger.jp`

Nakade	Atari Capture	Atari Defend	Lowlib	Pattern
Atari Defend	Nakade	Atari Capture	Pattern	Lowlib
Atari Defend	Nakade	Pattern	Atari Capture	Lowlib
Atari Defend	Atari Capture	Pattern	Nakade	Lowlib
Nakade	Atari Capture	Pattern	Lowlib	Atari Defend

Table 4.1: Selected *agent database*.

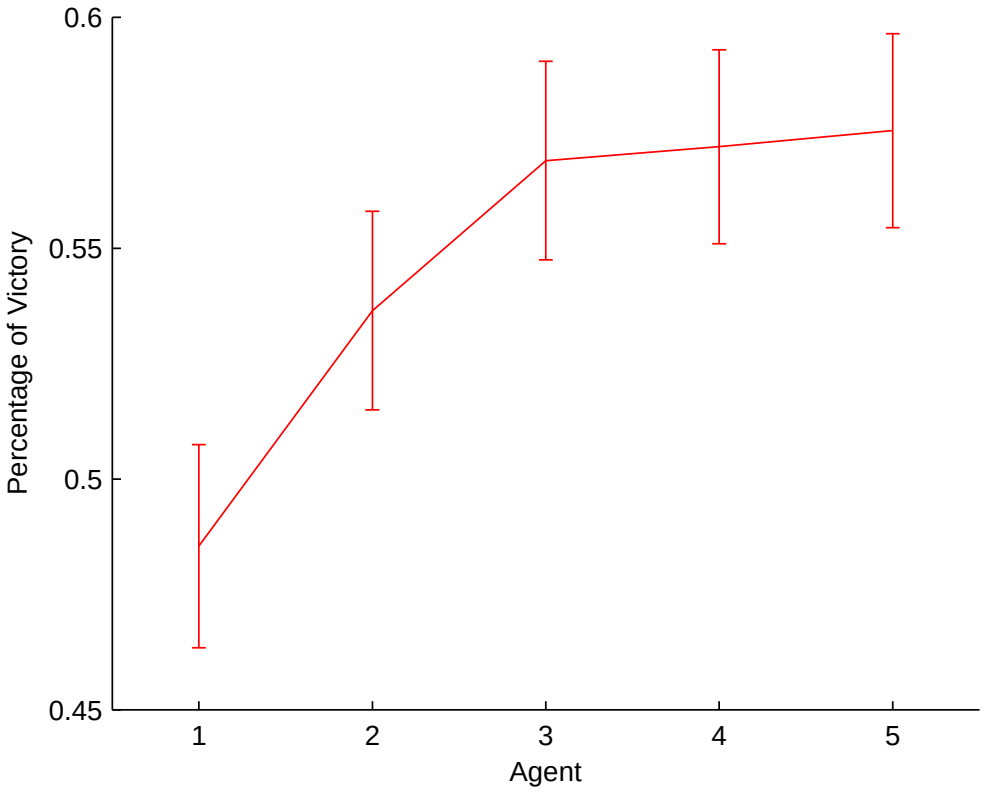


Figure 4.1: Percentage of victory for the selected *agent database*.

Agent Number	Percentage of Victory
0	$48.50\% \pm 2.20\%$
5	$52.85\% \pm 2.15\%$
6	$53.60\% \pm 2.15\%$
64	$57.30\% \pm 2.15\%$
70	$29.60\% \pm 1.90\%$

Table 4.2: Percentage of victory for each individual agent.

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 run on 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%.

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 4.2, 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).

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 phenomenon that makes the algorithm stronger.

The learning graph of our algorithm can be seen in Figure 4.2. After adding agent 5 and 6, the system fluctuates, and is able to escape from the local minimum (lack of

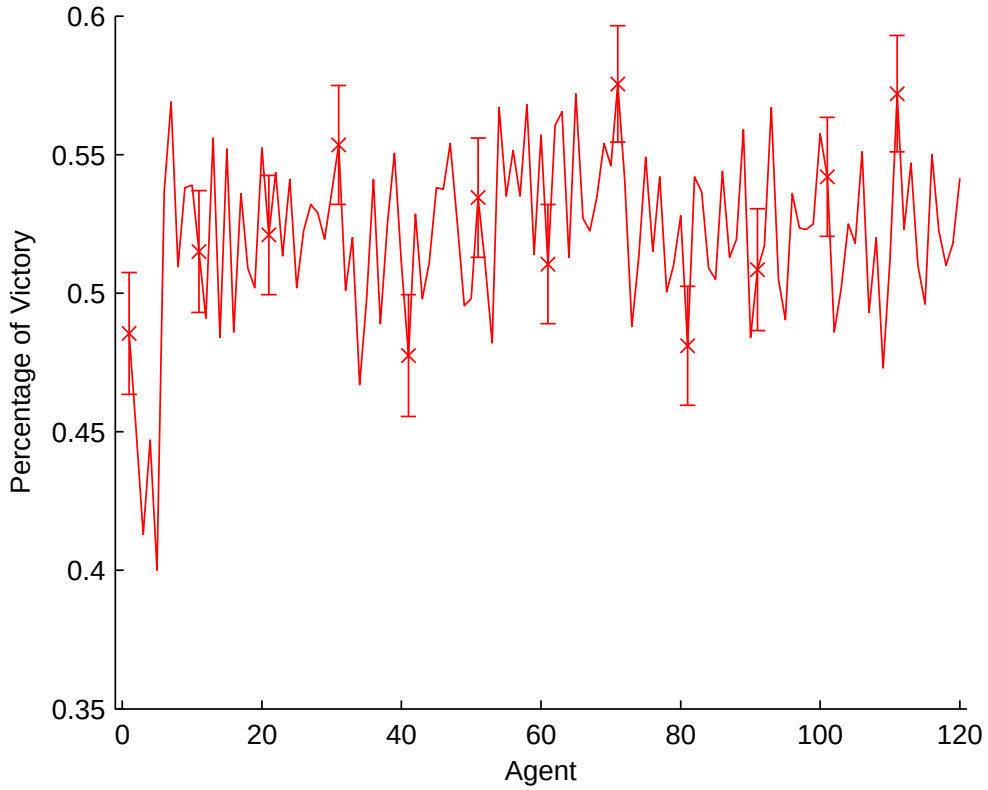


Figure 4.2: Learning graph, as the algorithm tries to add each agent in the database.

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 (see Table 4.3), by intuition and trial and 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 4.4. The result obtained with the addition of each new agent is represented in Figure 4.3, and the learning graph can be seen in Figure 4.4. This time, we found a slightly better result, of 59.15% ($\pm$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 4.5. The Agent Number of each agent

Nakade	Atari Capture	Atari Defend	Lowlib	Pattern
Atari Defend	Nakade	Atari Capture	Pattern	Lowlib
Atari Defend	Nakade	Pattern	Atari Capture	Lowlib
Atari Defend	Pattern	Lowlib	Nakade	Atari Capture
Atari Defend	Lowlib	Pattern	Atari Capture	Nakade
Atari Defend	Pattern	Nakade	Atari Capture	Lowlib
Atari Capture	Atari Defend	Nakade	Lowlib	Pattern
Atari Capture	Atari Defend	Pattern	Lowlib	Nakade
Atari Defend	Atari Capture	Pattern	Nakade	Lowlib
Lowlib	Atari Defend	Pattern	Nakade	Atari Capture
Atari Defend	Atari Capture	Nakade	Lowlib	Pattern
Lowlib	Atari Defend	Nakade	Pattern	Atari Capture
Pattern	Atari Defend	Atari Capture	Nakade	Lowlib
Atari Defend	Nakade	Lowlib	Pattern	Atari Capture
Atari Defend	Atari Capture	Lowlib	Pattern	Nakade

Table 4.3: Set of 15 agents that we believed to be strong.

Nakade	Atari Capture	Atari Defend	Lowlib	Pattern
Atari Defend	Nakade	Atari Capture	Pattern	Lowlib
Atari Defend	Nakade	Pattern	Atari Capture	Lowlib
Atari Defend	Pattern	Lowlib	Nakade	Atari Capture
Atari Capture	Nakade	Atari Defend	Lowlib	Pattern

Table 4.4: Selected *agent database*, in the not random order.

is 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. 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.

However, we were not satisfied that our best result needed manual intervention in the

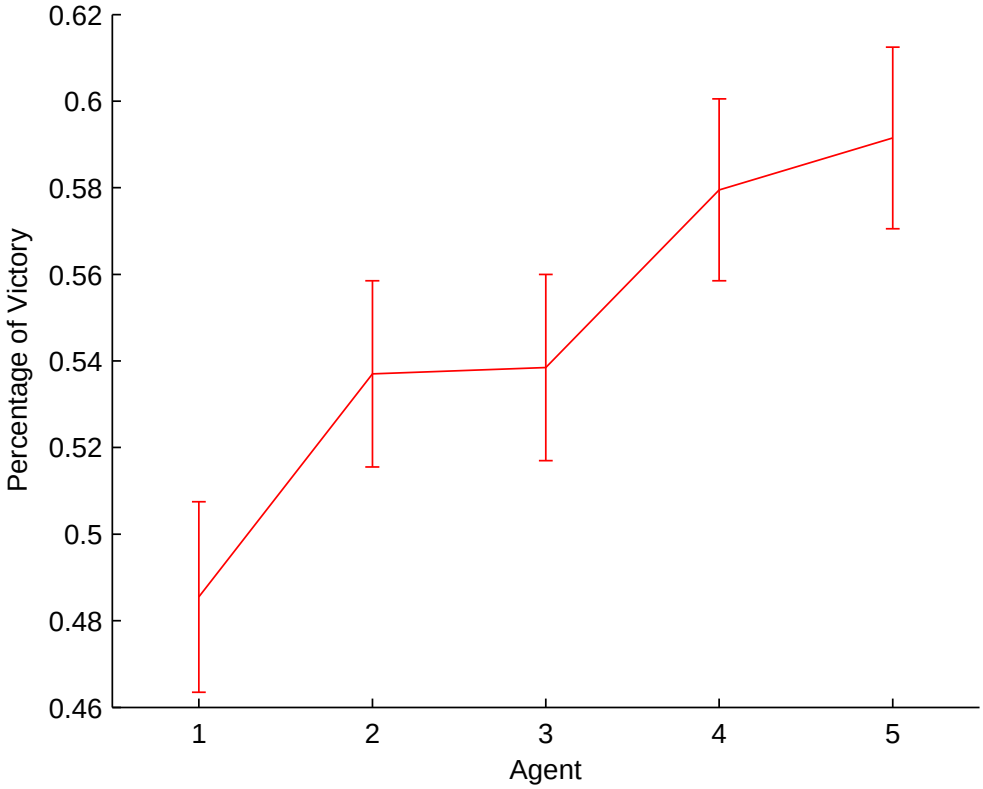


Figure 4.3: Percentage of victory for the selected *agent database*, in the not random order.

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

Table 4.5: Percentage of victory for each individual agent, in the not random order.

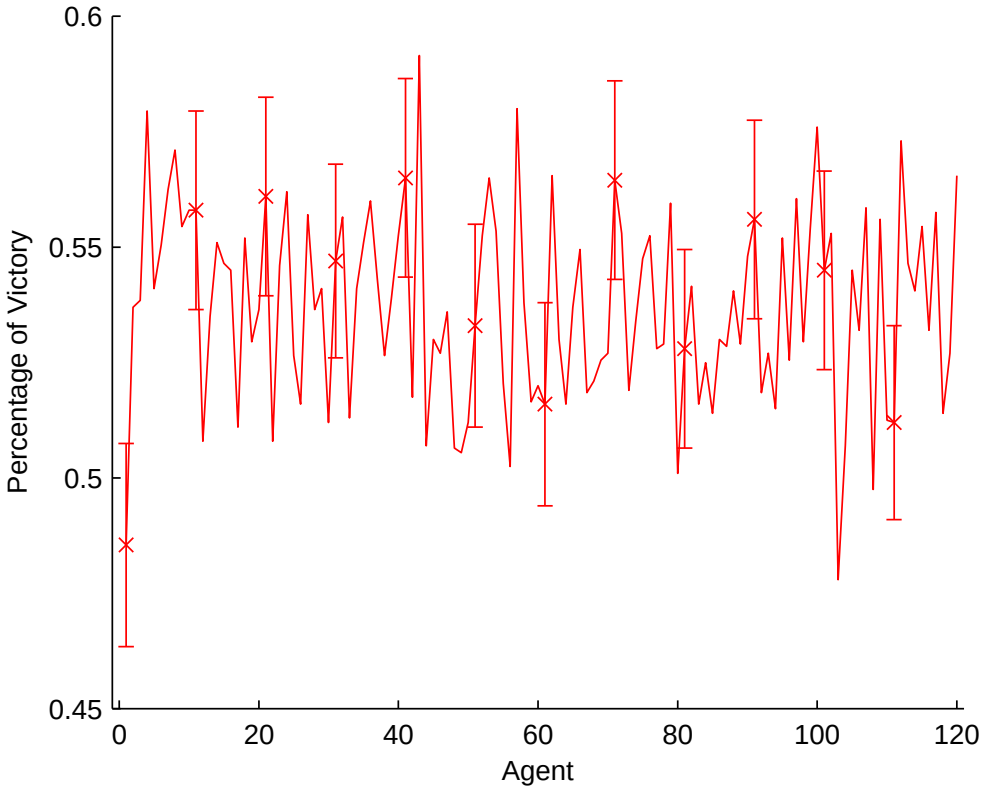


Figure 4.4: Learning graph in the not random order, as the algorithm tries to add each agent in the database.

Nakade	Atari Capture	Atari Defend	Lowlib	Pattern
Atari Capture	Nakade	Atari Defend	Lowlib	Pattern
Atari Capture	Nakade	Atari Defend	Pattern	Lowlib
Atari Defend	Pattern	Lowlib	Atari Capture	Nakade
Atari Defend	Nakade	Pattern	Lowlib	Atari Capture
Atari Defend	Atari Capture	Pattern	Nakade	Lowlib
Atari Defend	Pattern	Atari Capture	Nakade	Lowlib

Table 4.6: Selected *agent database*, in the second random order.

Agent Number	Percentage of Victory
0	48.55% ± 2.20%
6	50.15% ±2.20%
7	53.90% ±2.15%
12	53.70% ±2.15%
15	52.65% ±2.15%
31	54.20% ±2.15%
41	54.90% ±2.15%

Table 4.7: Percentage of victory for each individual agent, in the second random order.

order that the agents were tried. Therefore, we generated again another random order to test each agent, and we ran our learning algorithm a second time. This time we could obtain a very good result without manual intervention. The agents that compose the new solution found by the learning algorithm can be seen in Table 4.6. The result obtained with the addition of each new agent is represented in Figure 4.5, and the learning graph can be seen in Figure 4.6. This time, we found a slightly better result, of 59.5% (±2.10%). We also executed games with our system running with a single agent. The percentage of victory obtained for each agent can be seen in Table 4.7. The difference between the group and each agent seems to be higher than in the previous experiments, but it does not seem to be big enough yet to prove that the group is better than the best agents. We will return to this point in Chapter 5.

As can be seen, we could find three 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 lead to higher quality simulations, creating stronger players. It is notable that we could obtain a percentage

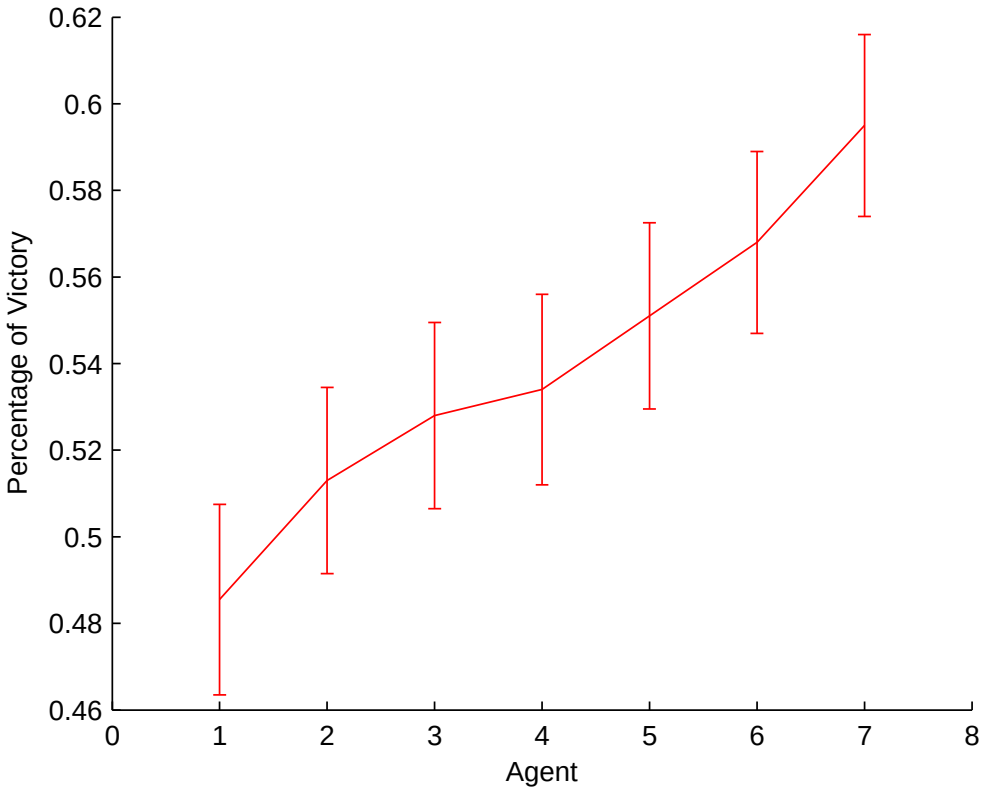


Figure 4.5: Percentage of victory for the selected *agent database*, in the second random order.

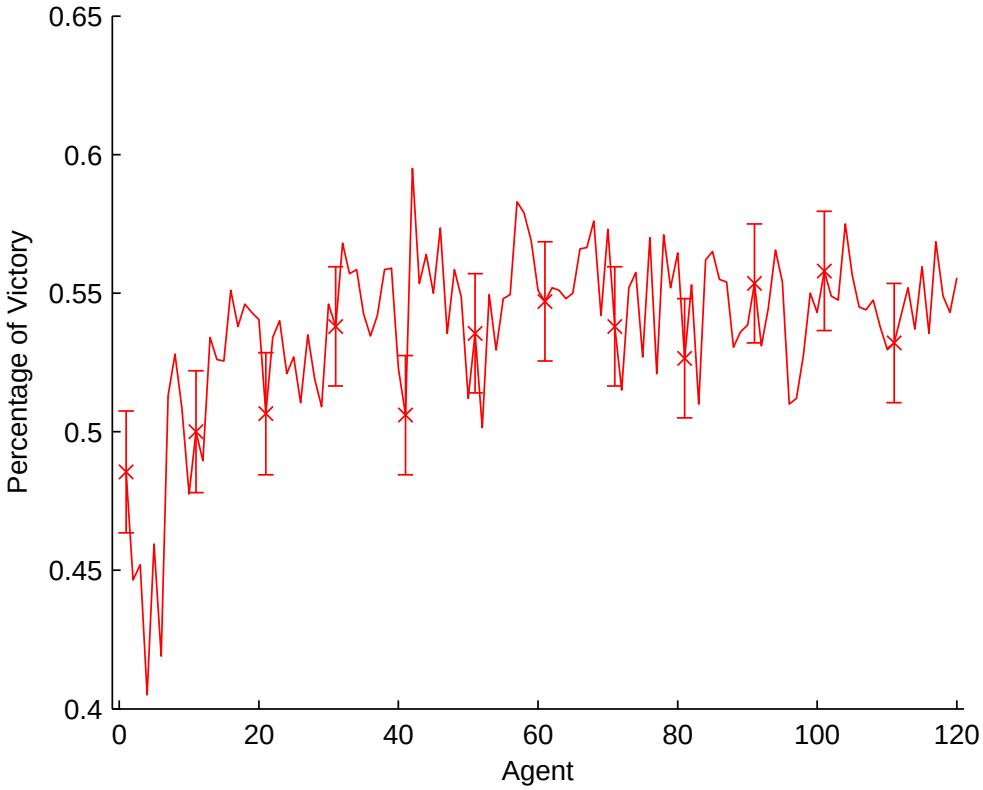


Figure 4.6: Learning graph in the second random order, as the algorithm tries to add each agent in the database.

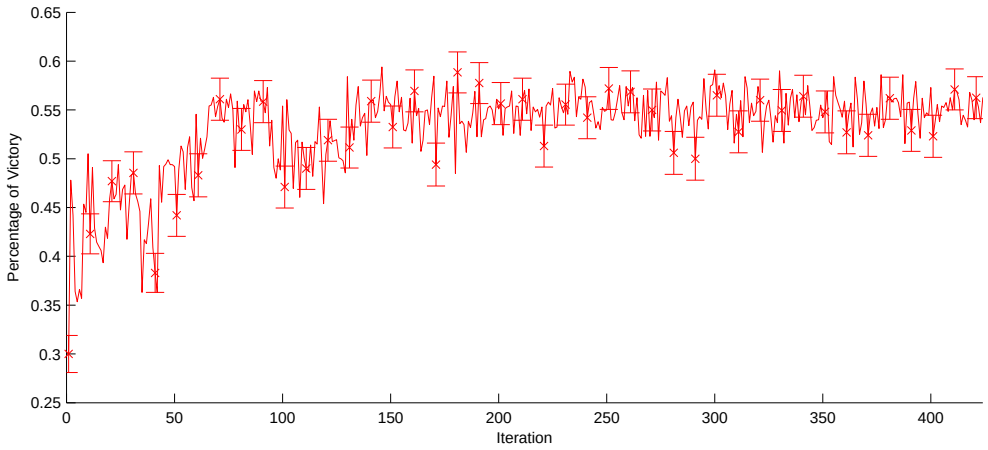


Figure 4.7: Learning graph with all iterations.

of victory of around 59% against Fuego, in its default configurations for time limit and number of playouts per leaf.

4.2 Simulated Annealing

In this section we are going to present the experiments performed using the simulated annealing technique. The experiments were executed with the same configurations as the previous one: 9x9 Go, with Fuego's default time limit and number of playouts per leaf. We also executed 1000 games per configuration, 500 with our system playing as White, and 500 as Black. The same cluster was used, of Intel(R) Xeon(R) CPU E5530, at 2.4GHz and with 24GB of RAM. We used the initial temperature t as 0.5, the initial percentage of victory v as 0.4855, the temperature decrease constant α as 0.9, and the probability of adding an agent ρ as 0.5.

The learning graph with all iterations can be seen in Figure 4.7. The result obtained after each accepted modification can be seen in Figure 4.8, and the final *agent database* is represented in Table 4.8. In Figure 4.9, we can see the result with only the agents that remained after the final iteration. As can be seen, the result was not better than the one found by the hill-climbing algorithm. In the final iteration, we still had a percentage of victory of 59.40% (±2.10%).

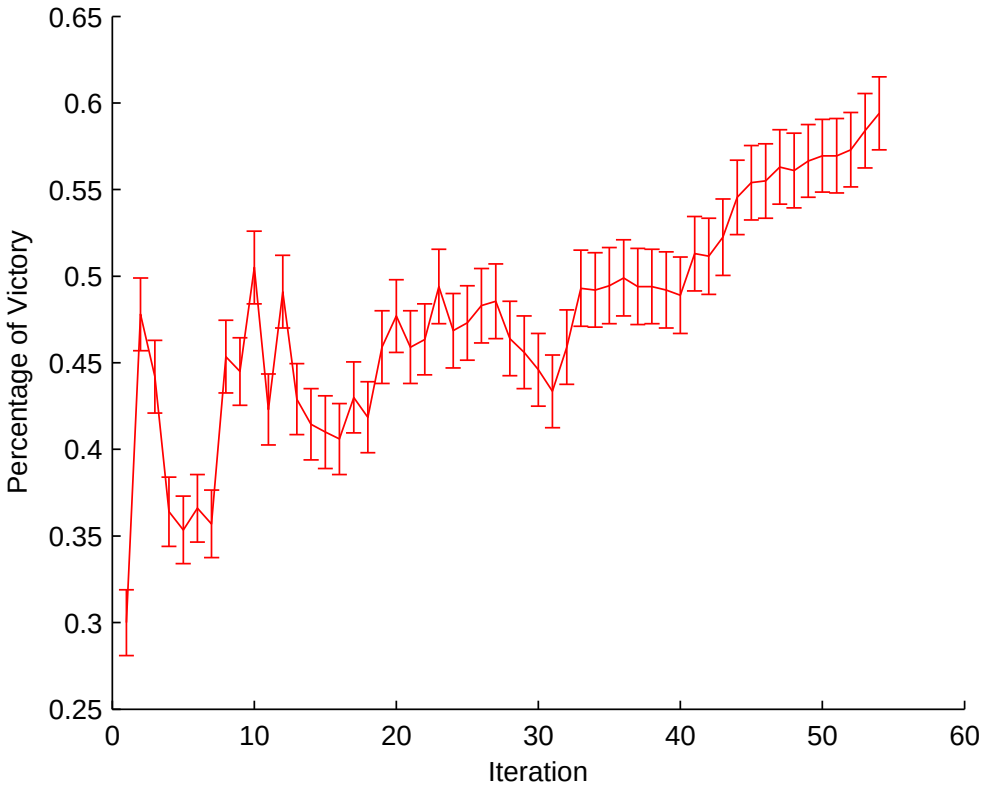


Figure 4.8: Learning graph with only accepted iterations.

Atari Defend	Pattern	Nakade	Atari Capture	Lowlib
Atari Defend	Lowlib	Atari Capture	Nakade	Pattern
Atari Defend	Pattern	Lowlib	Atari Capture	Nakade
Atari Defend	Nakade	Pattern	Atari Capture	Lowlib

Table 4.8: Selected *agent database*, by the Simulated Annealing learning algorithm.

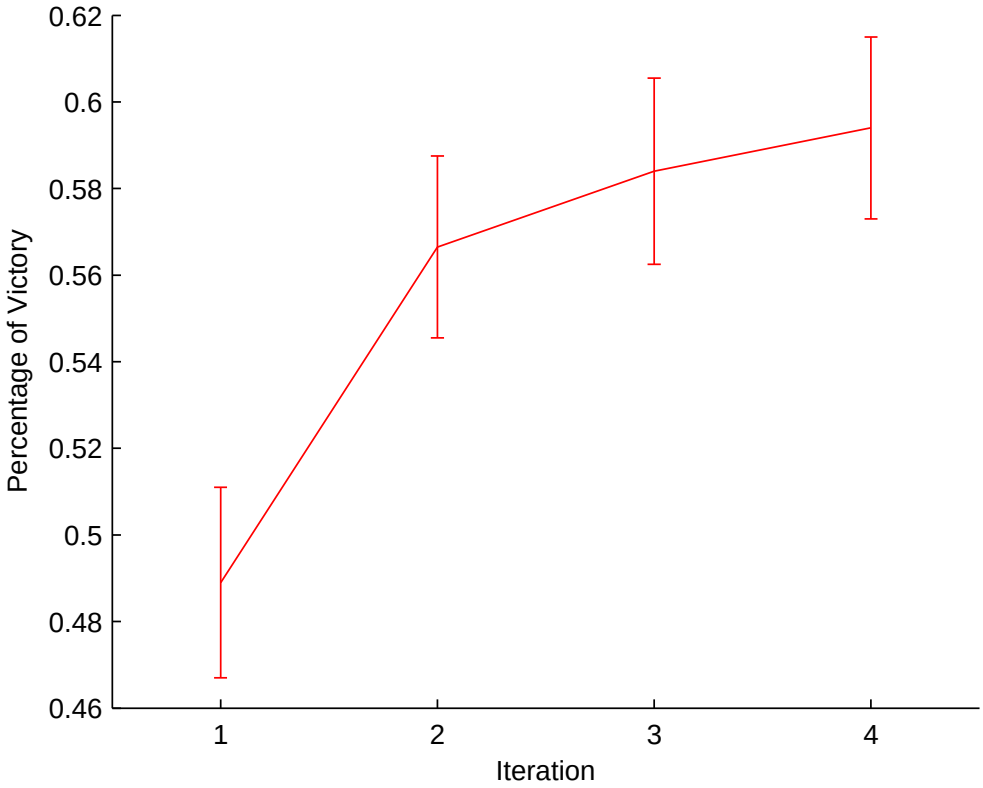


Figure 4.9: Agents that remained after the final iteration.

Agent Set	Percentage of Victory
29	0.3
29, 13	0.478
29, 13, 38	0.442
29, 38	0.364
29, 38, 56	0.3535
29, 38	0.366
29, 38, 86	0.357
29, 38, 86, 3	0.4535
29, 86, 3	0.445
29, 86, 3, 65	0.505
29, 86, 3, 65, 45	0.423
29, 86, 3, 65, 45, 83	0.491
29, 86, 3, 65, 45, 83, 24	0.429
29, 86, 3, 65, 45, 83, 24, 92	0.4145
29, 86, 65, 45, 83, 24, 92	0.41
29, 86, 65, 45, 83, 24, 92, 112	0.406
29, 86, 65, 45, 83, 92, 112	0.43
29, 86, 65, 45, 83, 112	0.4185
29, 86, 65, 45, 83, 112, 61	0.459
29, 65, 45, 83, 112, 61	0.477
29, 65, 45, 83, 112	0.459
65, 45, 83, 112	0.4635
65, 45, 83	0.494
65, 45, 83, 73	0.4685
65, 45, 83, 73, 63	0.473
65, 45, 83, 73	0.483
65, 45, 83, 73, 0	0.4855
65, 45, 83, 73	0.464
65, 45, 83	0.456
65, 45	0.446
65, 45, 44	0.4335
65, 45, 44, 62	0.459
65, 44, 62	0.493
65, 44	0.492
65, 44, 10	0.4945
65, 44, 10, 8	0.499
65, 44, 10	0.494
65, 44, 10, 49	0.494
44, 10, 49	0.492

Table 4.9: Iterations of the Simulated Annealing learning algorithm.

Agent Set	Percentage of Victory
44, 10, 49, 5	0.489
44, 10, 49, 5, 14	0.513
44, 10, 49, 5	0.5115
10, 49, 5	0.5225
10, 5	0.5455
10, 5, 44	0.554
10, 5	0.555
10, 5, 87	0.563
5, 87	0.561
5, 87, 36	0.5665
5, 87, 36, 1	0.5695
5, 87, 36	0.5695
5, 36	0.573
5, 36, 20	0.584
5, 36, 20, 2	0.594

Table 4.10: Iterations of the Simulated Annealing learning algorithm (continuation).

Chapter 5

Discussion

"The important thing is not to stop questioning" (*Albert Einstein*)

In this thesis 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. This might happen because it would be equivalent to simply choosing randomly one of the 5 heuristics at each simulation step. 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 by about 5%, compared to the solution that we could find manually. Therefore, it had a significant impact in our results. We were also able to find, without manual intervention, a solution that was as good as the one found with a not random order, a very good indication of the quality of our learning process.

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. In this thesis, we only presented single-agent executions with the agents that were part of a group, because our objective was to evaluate if the group is stronger than its components. However, as a future work, it is also interesting to show the percentage of victory of all possible agents. After finding the strongest agent, we could test one certain group or run again our learning algorithms against this agent, instead of Fuego's original agent.

We believe that our approach is in a good direction to improve MCTS. However, even our straight $O(n)$ learning algorithm, executing on 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.

Nevertheless, we believe that much can still be discovered in this direction. A 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. 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.

In order to escape local minima and explore those questions, we tried in this work to apply a simulated annealing learning algorithm, and accept agents even when they decrease the solution. However, the result that we found was still very similar to the one found by our simple hill-climbing approach. One of the disadvantages of Simulated Annealing techniques is the influence of a great number of parameters [Dreo et al. 2005]. Therefore, it might be possible to find better results by trying different combinations of them. We could also try other learning algorithms. Unfortunately, it does not seem to be possible to apply learning algorithms like evolutionary methods, due to the high cost of testing each solution.

Another interesting idea to continue this work is to approximate our agent model to Page's model ([Page 2007]). As we saw in Chapter 2, in Page's model each person has a collection of heuristics and, by having a diverse team, we can have a greater collection of heuristics and solve complex problems. However, in our agent model, every agent has the same set of heuristics, and we varied the prioritization order of them. Therefore, an interesting and important question is: What would happen with the result, if we have a greater number of heuristics, and each agent has a subset of them, like in Page's model?

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, as in Obata et al. [2009]? 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? Another possible direction

is to try to use Multi-Agent Systems ideas 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.

5.1 Why Agents?

"The heart of all major discoveries in the physical sciences is the discovery of novel methods of representation." (*Steven Toulmin*)

Some people might argue if we really have a set of "agents", and not simply a collection of heuristics, each one applied with a different probability. We can divide their argument in basically two questions: 1 - Is this really a multi-agent system?; and 2 - What is the importance of seeing this as a multi-agent system?

We will start by answering question number 1. In order to do it, we need to go back a bit, and ask ourselves if our set of heuristics is an agent. As we saw in Chapter 2, according to Russell and Norvig [2003], an agent is something that senses its environment and, based on some computation (that can be extremely simple), generates an action, that might change the state of the environment. It is very easy to see this in the case of a robot, for example. If we consider a robot that is moving in an environment, with sensors for obstacle detection, and when it perceives an obstacle, it generates an action to avoid it, we can clearly consider it an agent. But the definition is broader than this. We can also consider an artificial chess player, for example. It is an agent that, given a chess board, generates an action on that board, after a very complex and long computation, and this action will change the state of the board. The definition can go even broader than this. Any computer software can be seen as an agent, with its environment being the state of the available computational devices, the input given by the user or some arbitrary device, and, after the computation, the software has an action, the output, that will be shown to the user or will be used as input to some other device. Therefore "agent" is a very broad concept, and we can freely use it when it is convenient for our analysis.

In the case of our research, the environment of the agent is a Go board. The agent perceives the Go board and then, based on its heuristics, generates an action that is going to modify the environment: a movement in the Go board. Even though an agent that always perform the same action could still be considered an agent (though a very dull one), in our case the action of the agent will depend on the state of the Go board. The

heuristics have a hierarchical order, which does not mean that the first one will always be applied. The agent will try to apply the first one, and in case it cannot be applied, it will try to apply the second one, etc. Therefore, depending on the state of the environment, it might apply its first heuristic, its second heuristic, etc. Given a certain environment, it is possible to predict the behavior of the agent, so it is possible to argue that the agent is not really making a "decision". We agree that our agent is a simple one, but this does not mean that it is not an agent. We can compare it with a simple robot, that has sensors on its left and its right, and turns right when it perceives something on the left and turns left when it perceives something on the right. The behavior of this robot is perfectly predictable, but it is very clear that it is an agent.

Actually, our agent architecture, composed by a set of heuristics with a hierarchical order, can be directly related to the robot architecture called *Subsumption Architecture* [Brooks 1985]. In this architecture, a robot is composed by a series of modules, and they are organized over layers, following a hierarchical architecture. The high-level modules can suppress the output of lower level modules. Therefore, it is possible to say that our simple agent is following a *Subsumption Architecture*.

We hope it is clear now that our collection of heuristics (with an associated hierarchical order) can be seen as an agent. As we have many agents, each one with a different hierarchical order (and, therefore, a different playing style), it follows that we have a multi-agent system, and one with heterogeneous agents, at least if we consider the agent's internal algorithm. We know that this use of agents and multi-agent systems is not in the context that is generally seen in the literature, so it can be quite intriguing. We believe that this conceptual step, linking the game community and the multi-agent community is one of the most important contributions of this work.

It follows then our answer to the next question: What is the importance of seeing this as a multi-agent system? Giving an agent set, it is probably possible to achieve the same effect by building a probabilistic decision tree [Mitchell 1997]: there is a certain probability for each heuristic in the first level of the tree (that sum up to one), and as we move down one level of the tree, there will be a certain probability for the remaining heuristics, etc,

until we reach the final level of the tree, in which we will execute the final remaining heuristic with a probability of one. If we look at the problem in this way, we lose the conceptual step linking our research with the multi-agent community, though. The link brings many advantages. First, it allows us to discuss and justify our results based on the concepts of emergence, diversity and stigmergy. Second, it exposes to the general agent community the opportunities that Monte Carlo Go might offer to their techniques. We hope this opens a class of agent-oriented approaches for dealing with Computer Go, one of the main challenges for Artificial Intelligence. It is also a concrete, interesting, and easy to evaluate application for Multi-Agent Systems and, therefore, it was very well-received by the Multi-Agent community.

The metaphor also allows us to look at the problem in a different way, and try to find techniques for selecting agents sets, compare the result of the group with individual agents, think about influences between agents, etc. And, of course, we can explain and justify our results using theories not only from the Multi-Agent community, but also from the "wisdom of crowds" community (such as [Page 2007]). All this would be lost if we think about this problem only as a probabilistic decision tree.

One question that remains to be answered though, is if it is easier to learn a probabilistic decision tree than a set of agents. Could we obtain a better result with this alternative perspective? We still do not know the answer to this question. However, even if learning a probabilistic decision tree is easier, it seems to be possible to always convert it back to a set of agents, if we consider that we can have repetition in our set (what will actually transform it into an agent bag). The mapping between the two algorithms might not be so simple as it seems to be at first sight, as in our algorithm we make a random decision only in the beginning of the movement selection process, that would correspond to a full-path in a probabilistic decision tree, and in the tree the probability of each level will depend on what was chosen before. However, for the sake of the argument, let's assume there is a mapping. Therefore, we would be able to convert back the learned tree to a bag of agents, and we would not lose the link between our research and the Multi-Agent theories.

The problem is that the same argument can be used against us. We can say that we

have a good result, given by emergence, stigmergy and diversity, and one could say that we only have to look at the solution as a probabilistic decision tree to see that there is no agent interaction whatsoever. We believe that the best way to answer this argument is to use the concept of a perspective [Page 2007]. A perspective is a way to look at a particular event or problem. It is a method of representation, and it is often metaphorical. Based on a certain perspective, we can build solutions to a problem, what Page calls heuristics. Some solutions and ideas come easily when we use a certain perspective, and other solutions and ideas come in a different perspective. Therefore, by increasing the number of perspectives, we can increase the number of solutions that we are able to propose, increasing our number of available tools, and enabling us to reach better results in the end. That is exactly why Page argues that diversity is important in order to solve complex problems, as a team of diverse people have access to a great number of perspectives and heuristics.

Therefore, we believe that instead of discussing which perspective is right and which one is wrong, we should simply use the perspective that seems to be more appropriate to us in the moment, and try to have as many tools as possible to solve a certain problem. Therefore, new perspectives should always be welcome.

Even if the reader is still not satisfied with our perspective, we believe that this whole discussion about the definitions of agents and emergence, and the limits of these definitions is, by itself, already a major contribution of this work.

5.2 More on Emergence

In this section we are going to present two philosophical ideas about emergence and Go, but they are not related to the technical aspects of our work. The reader that is more interested in the technical work is welcome to skip this section.

First, it might be possible to think about the Go game itself as an emergent process. The Go stones interact in simple ways, by following simple rules, but the game progresses in a very complex system. In the case of Go, there is a central unity, the player, governing the progress of the system, as much as he can control it. However, as there is an opponent,

sometimes things go out of his control, and he has to stumble to the rules of the system, making movements that can be considered as almost obligatory (as not performing them is worse), but that will inevitably lead to big losses, or even the defeat of the game. Examples of these are abundant in the Go problems books, dealing with *tesuji* (local fighting problems) and *tsumego* (life and death problems).

Second, according to the ideas of *Situated Learning*, presented in Lave and Wenger [1991], we can see even the concept of Go itself as the result of an emergent process. According to the authors, knowledge and ideas are not something abstract, that are passed from an individual to the other. Knowledge is being constantly reconstructed (and maintained, somehow), by the social process that occurs between mentors and apprentices, in a community of practice. Therefore, the notion of what Go is, and the techniques for playing it well, exist and are constantly being modified and transformed by the interactions between Go players, the strong (masters) and weak ones (apprentices), happening in the Go saloons, schools, clubs, and professional leagues all over the world (communities of practice).

Chapter 6

Conclusion

"This World is not Conclusion.
A Species stands beyond—
Invisible, as Music—
But positive, as Sound—
It beckons, and it baffles—
Philosophy—don't know—"
(*Emily Dickinson*)

In this thesis we presented a new paradigm to the state of the art of Computer 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 for a variety of different works.

This work brings together the Computer Games community and the Multi-Agent com-

munity. We hope it serves as an example of how concepts like “emergence”, “stigmergy” and “diversity” can be easily applied to solve real problems. We can see a strong focus on the literature on using emergence only in optimization algorithms, and we hope this thesis can give some light in how to use emergence in problems that cannot be easily defined as an optimization situation. We also can see a strong focus on the computer games literature on how to parallelize the current algorithms, and we hope this thesis can show different ways to improve the strength, that do not need more computational power. Therefore, by making a bridge between the two communities, we hope we are bringing contributions to both.

This thesis might also be useful for scientists interested in the emergent process by itself, by giving an example of artificial emergence that could actually produce a complex result. Nature and society is full of examples of “natural” emergence, but examples of artificial emergence are not so abundant. As we said before, generally it is mostly seen only in optimization algorithms. Therefore, the example of emergence applied to Computer Go can be useful to scientists interested in emergence, and these ideas might also be transported to other complex systems.

We also believe that we present a nice example for the “intelligence of crowds” community, showing how a team of diverse agents could effectively solve a complex problem. Therefore, we hope that we provided empirical evidence of their theoretical work. However, we still could not prove that our team is better than the best agents. Our agent model is simple, though, it would be a nice future work to approximate our model to Page’s model.

A more concrete contribution of this work is the actual modification of the Fuego software, enabling it to be even stronger on 9x9 Go. We plan to let this modification be publicly available, so anyone can enjoy the benefits of a stronger player, and we hope that it could eventually be inserted in the official version of Fuego. A stronger player is useful not only for the popularity of Go and for the enjoyment of being able to play it on the computer, but also to increase Go comprehension and analysis.

There are many possibilities of research that can continue the work develop in this thesis, and we talked about many of them in Chapter 5. It is necessary to better study

the Simulated Annealing algorithm and actually find an agent set that can bring a better solution. It would also be interesting to explore other directions of research, for example, discovering solutions of how to select the best move when different programs cooperatively play Go.

One important practical aspect that needs to be solved is to bring those ideas for 19x19 Go. It is more difficult to execute experiments in the bigger board, due to the time requirements in order to play a full game. However, it is more useful for the game community, at least in practical aspects, to have a stronger player for the default Go board. The small one is used mainly for educational objectives, and not for professional Go playing. A simple experiment might be to test in the full-board an agent set that was learned in the 9x9 board. Would that agent set remain strong?

We believe that much can still be researched, and Computer Go can be greatly improved by exploring Multi-Agent System techniques. We hope this work is useful in opening new ideas, and in bringing even stronger Computer Go players for the near future.

Bibliography

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

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

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

Boon, M. (1990). A pattern matcher for Goliath. *Computer Go*, 13:13–23.

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

Bouzy, B. and Cazenave, T. (2001). Computer Go: an AI oriented survey. *Artificial Intelligence*, 132:39–103.

Brooks, R. A. (1985). A robust layered control system for a mobile robot. Technical report, Cambridge, MA, USA.

Brugmann, B. (1993). Monte Carlo Go. Technical report, Physics Department, Syracuse University.

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

Cazenave, T. and Jouandeau, N. (2008). 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. Springer-Verlag.

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

Colella, V. S., Klopfer, E., and Resnick, M. (2001). *Adventures in Modeling: Exploring Complex, Dynamic Systems with StarLogo*. Teachers College Press.

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

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

Dreo, Pétrowski, A., Siarry, P., and Taillard, E. (2005). Chapter 3 - simulated annealing. In *Metaheuristics for Hard Optimization: Methods and Case Studies*. Springer.

Engelbrecht, A. P. (2006a). Chapter 3 - optimization algorithms. In *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons.

Engelbrecht, A. P. (2006b). *Fundamentals of Computational Swarm Intelligence*. John Wiley & Sons.

Enzenberger, M., 0003, M. M., Arneson, B., and Segal, R. (2010). Fuego - an open-source framework for board games and Go engine based on Monte Carlo Tree Search. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):259–270.

Enzenberger, M. and Müller, M. (2009a). 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.

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

Farneback, G. (2008). GTP - Go text protocol. `http://www.lysator.liu.se/~gunnar/gtp/`.

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

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123.

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

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

Grasse, P. P. (1959). La reconstruction du nid et les coordinations interindividuelles chez bellicositermes natalensis et cubitermes sp. La theorie de la stigmergie: essai d'interpretation du comportement des termites constructeurs. *Insectes Sociaux*, 6:41–81.

Hart, T. and Edwards, D. (1963). The alpha-beta heuristic. Technical report, Cambridge, MA, USA.

Johnson, S. (2001). *Emergence - The Connected Lives of Ants, Brains, Cities, and Software*. Scribner, 1st edition edition.

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

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

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

Lave, J. and Wenger, E. (1991). *Situated Learning - Legitimate Peripheral Participation*. Cambridge: University of Cambridge Press.

Marcolino, L. S. and Chaimowicz, L. (2008). 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.

Marcolino, L. S. and Chaimowicz, L. (2009a). 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.

Marcolino, L. S. and Chaimowicz, L. (2009b). 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.

Marcolino, L. S. and Matsubara, H. (2011). Multi-agent Monte Carlo Go. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, pages 21–28.

Mitchell, T. (1997). Chapter 3 - decision tree learning. In *Machine Learning (Mcgraw-Hill International Edit)*. McGraw-Hill Education (ISE Editions), 1st edition.

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

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

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

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

Reynolds, C. W. (1987). 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.

Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition edition.

Soejima, Y., Kishimoto, A., and Watanabe, O. (2009). Root parallelization of Monte Carlo Tree Search and its effectiveness in Computer Go. In *14th Game Programming Workshop (GPW-09)*. (in Japanese).

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

van der Werf, E., van den Herik, H., and Uiterwijk, J. (2003). Solving Go on small boards. *Journal of the International Computer Games Association*, 26(2):92–107.

Von Neumman, J. (1928). Zur theorie der gesellschaftsspiele. *Math. Annalen.*, 100:295–320.

Weiss, G., editor (1999). *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT Press, Cambridge, MA, USA.

Wilcox, B. (1985). Reflections on building two Go programs. *SIGART Bull.*, pages 29–43.

Wolf, T. (1994). The program GoTools and its computer-generated tsume go database. In *Proceedings of the Game Programming Workshop in Japan'94*, pages 84–96.

Wolf, T. (2000). Forward pruning and other heuristic search techniques in Tsume Go. *Information Sciences*, 122:59–76.

Wolfe, D. (2002). Go endgames are PSPACE-hard. *More Games of No Chance*, pages 125–136.

Wooldridge, M. J. (2001). *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA.

Yen, S.-J., Chou, C.-W., Hsu, S.-C., Chen, J.-C., and Yang, T.-N. (2009). Improvement of MCTS in Computer Go. In *14th Game Programming Workshop (GPW-09)*.

Zobrist, A. L. (1969). A model of visual organization for the game of Go. In *Proceedings of the May 14-16, 1969, spring joint computer conference*, AFIPS '69 (Spring), pages 103–112, New York, NY, USA. ACM.

Appendix A

Modifications in Fuego

" 'Free software' is a matter of liberty, not price. To understand the concept, you should think of 'free' as in 'free speech' not as in 'free beer.' " (*Richard Stallman*)

In this appendix we are going to briefly introduce the Fuego architecture and the modifications that we made in order to implement our solution. The reader interested in Fuego architecture should refer to Enzenberger and Müller [2009a], Enzenberger et al. [2010] for more details.

According to its authors, Fuego is "an open-source software framework for developing game engines for full-information two-player board games, with a focus on the game of Go". Therefore, the objective of Fuego is not simply to be a software to play Go. It was created as a framework to be used for other games as well, but its greatest use is as a Go software. The main point of Fuego is not to present something new in terms of algorithms, but to provide an implementation of the state of the art algorithms that can be freely studied and modified by anyone. Therefore, it is a powerful tool for research, and we are very thankful to Fuego's authors for their work.

Fuego's interface is only handled through text messages, using the Go Text Protocol [Farneback 2008]. Therefore, the user has to install some graphic interface for the program. The recommended one in Fuego's website is GoGui (`http://gogui.sourceforge.net/`), which can be seen in Figure A.1.

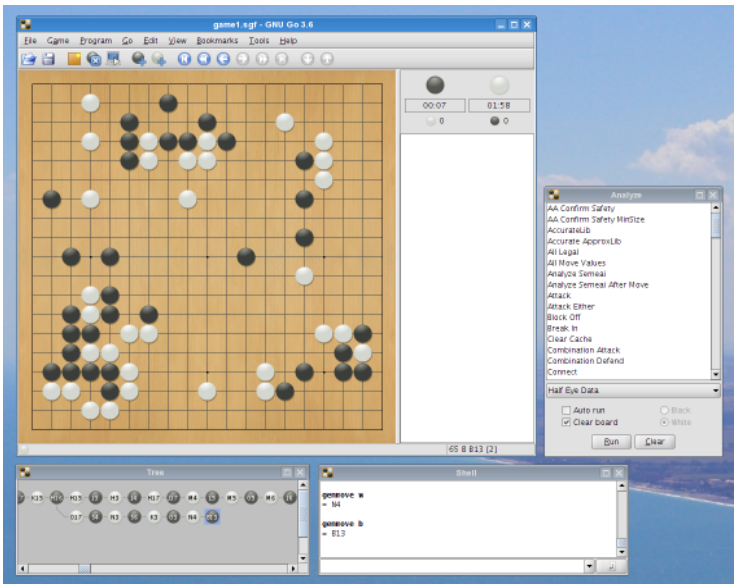


Figure A.1: GoGui screenshot, taken from the official website.

Fuego's architecture was built with care, trying to follow the principles of a good Software Architecture. As one of the objectives of Fuego is to provide a software to be freely studied and modified, it is important to build it in a clear and extendable way.

Fuego is organized into seven modules. The module *GtpEngine* is the one responsible for handling to Go Text Protocol, therefore it provides an interface for input/output based on text messages. *SmartGame* is responsible for platform-dependent operations (such as time measurement, process creation, etc). This module also provides general algorithms for board games programming. Therefore, it provides an implementation of the alpha-beta search and the UCT search algorithm. The module *Go* provides functionality specific for playing Go, such as abstractions for the Go board, etc. The module *SimplePlayers* is a collection of simple algorithms for playing Go, used for testing. The main Go engine is in the module *GoUct*, and *FuegoMain* provides the main application (while *FuegoTest* provides tests, for debugging). A representation of the modules and its dependences can be seen in Figure A.2.

Therefore, the core of our modification is in the module *GoUct*, that provides the UCT Monte Carlo algorithm for Computer Go. More specifically, in the file *GoUctPlay-*

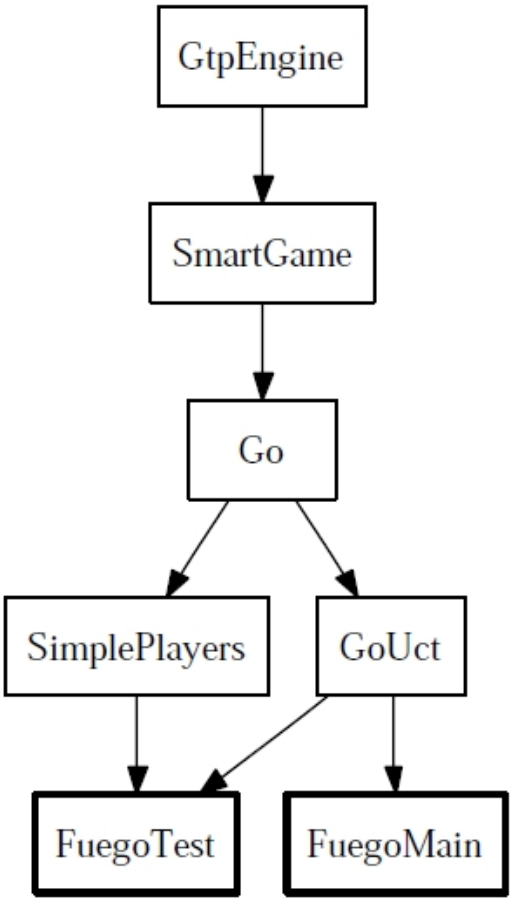


Figure A.2: Modules of Fuego, taken from [Enzenberger et al. 2010].

outPolicy.h, we can find in the function *GenerateMove* the algorithm for generating the pseudo-random moves during the Playout Phase (the simulations executed in order to evaluate a leaf). Therefore, we modified this function, in order to be able to use many agents during the playout execution, as we explain in Chapter 3.

Fuego allows the user to set many parameters using the Go Text Protocol. We also made a modification in the file *GoUctCommands.cpp*, to create a new parameter: *uct_param_policy number_agents*. This allow us to change the number of agents of the system, without the need to recompile the whole program.

The learning algorithms (hill-climbing and simulated annealing) are handled outside Fuego. We developed a collection of *python scripts* responsible for automatically generating a new version of Fuego, according to the rules of the learning algorithm, and testing it over a cluster.

All the source code of this project, including the modifications of Fuego and the learning algorithm scripts are going to be freely available on the Internet, on the website `http://www.leandromarcolino.com.br/academic`. We hope they can be useful for the community, and can be used as a basis for the development of even stronger programs.