Game-Tree Exploration using Stochastic Diffusion Search

Thomas Tanay
M.Sc. in Cognitive Computing
Goldsmiths, University of London

September 2012

Abstract

In this thesis, a method inspired from the two existing techniques Monte-Carlo Tree Search (MCTS) and Coupled Stochastic Diffusion Search (CSDS) is proposed to address the problem of computer game-playing.

Similarly to MCTS, it is a best-first search method based on the selective sampling of moves in the simulation of a great number of random games.

Similarly to CSDS, it relies on the collaboration of several populations of agents dynamically influencing each others' behaviour.

The proposed method is characterised by the use of one distinct population of agents per node of the studied game-tree, and the dynamical reallocation of agents between the different populations during operation.

The method has been applied to practical game-playing on Hex and is shown to lead to a slightly better tactical play than vanilla Monte-Carlo methods.

Contents

Chapter 1

Introduction

Stochastic Diffusion Search (SDS) was introduced by Bishop in his Ph.D. thesis as a method for pattern classification [Bishop, 1989b] and has taken since then a central place in his research program. In particular, it is currently being investigated as a possible instantiation of the proposed new Interaction paradigm for Cognitive Sciences, in a project involving M. Bishop, S. Nasuto, M. Spencer and E. Roesch and supported by the John Templeton Foundation. The entire body of work on SDS relies on strong convictions about the nature of Cognition. Naturally the share of most of these convictions was an important condition for the decision to undertake the present thesis.

1.1 Conceptual Framework

One of the fundamental motivations for the study of Stochastic Diffusion Search is the belief that Computationalism—the paradigm that gave birth to the Cognitive Sciences and that still strongly influences their development—is the wrong framework to address the problem of human cognition.

1.1.1 The Computational Metaphor

The idea that the brain is a computing device can be traced back to 1943[1] and the publication of Warren McCulloch and Walter Pitts paper *A Logical Calculus of Ideas Immanent in Nervous Activity*. With this paper indeed emerged the view that:

1. The brain is fundamentally performing 'logical calculus'.
2. The logical calculus performed by the brain is embedded in the 'nervous activity'.

This view found particular resonance in the young science of computing machines and in particular influenced John von Neumann in his design of the modern digital computer [von Neumann, 1945]. Since then, the metaphor of the brain as a computing device has become ubiquitous and is most represented through the cognitivist (or computationalist) paradigm.

[1] In fact some roots of the idea can be found back to Hume or Kant [Dreyfus, 1992, p. 156]

In *The Embodied Mind*, Varela, Thompson and Rosch propose to summarize the cognitivist research program as answers to three fundamental questions:

Question 1: What is cognition?

Answer: Information processing as symbolic computation—rule-based manipulation of symbols.

Question 2: How does it work?

Answer: Through any device that can support and manipulate discrete functional elements—the symbols. The system interacts only with the form of the symbols (their physical attributes), not their meaning.

Question 3: How do I know when a cognitive system is functioning adequately?

Answer: When the symbols appropriately represent some aspect of the real world, and the information processing leads to a successful solution of the problem given to the system.

The computationalist paradigm was unchallenged from its birth in the 40s until the late 70s, and remained deeply influential until today in Artificial Intelligence (AI) as well as in philosophy of mind. Yet it is subject to strong philosophical criticisms. In the paper *A Cognitive Computing Fallacy? Cognition, Computations and Panpsychism* (2009), Bishop summarises three of the main arguments against computationalism. They are rapidely presented here:

The Chinese Room Argument exposed by Searle [Searle, 1980]. This argument was initially conceived as an answer to the work of Shank and Abelson [Schank and Abelson, 1977] who claimed to have conceived a machine that can genuinely understand stories. In brief, the Chinese Room argument proposes to imagine what would happen if Searle himself was following the instructions of the machine's program (written in English, the only language he can speak) to answer questions about a story written in another language (say Chinese) while being locked in a room. The thought experiment suggests that although Searle would be able to answer the questions correctly (since he is provided with an appropriate program), the situation would be totally different from what happens when he answers questions about a story directly written in English: it seems that Searle cannot be said to truly understand the story written in Chinese. The Chinese Room argument is usually said to support the premise that "syntax is not enough for semantics".

Gödelian Argument initially proposed by Lucas [Lucas, 1961] and then developed by Penrose [Penrose, 1994]. This argument is based on Gödel's incompleteness theorems and maintains that mathematical intuition cannot be grasped by the functioning of a formal system. Simply put, it consists in saying that "for any such formal system F, humans can find the Gödel sentence [the true but unprovable statement in F given by the theorem] whilst the computation/machine (being itself bounded by F) cannot." [Bishop, 2009]

Dancing with Pixies argument formulated by Bishop [Bishop, 2002] as an extension of a previous argument from Putnam [Putnam, 1988]. The dancing with Pixies argument is a reductio ad absurdum that attempts to show that if an appropriately programmed computer really has genuine cognitive states, then panpsychism holds. The crux of the argument is the observation that computational states are not intrinsically material, but rather are mapped on physical states in an arbitrary way. Hence, "every open system implements every Finite State Automaton (FSA)" [Putnam, 1988]. This result can then be applied (on a finite time window with fixed input—a condition which should not be considered as restrictive for the proponents of strong AI) on any program supposedly instantiating consciousness and hence every open system is conscious.

These arguments constitute powerful criticisms of the computational metaphor. Although they often got a mixed reception [Haugeland, 2002], [Chalmers, 1995], [Chalmers, 1996] or are often simply ignored by the AI community, they strongly suggest that there is something wrong about computationalism.

1.1.2 The nature of Cognition

The first major challenge to the cognitivist approach emerged in the rediscovery of connectionist ideas in the late 70s. Interestingly, connectionism find its roots in the same paper as cognitivism, i.e. McCulloch and Pitts paper *A Logical Calculus of Ideas Immanent in Nervous Activity*. However, it seems that the development of efficient serial computers was a necessary prerequisite to explore and modelise more complex distributed systems, which might explain why cognitivism developed first. With connectionism appeared two important ideas:

1. There is no need for a symbolic level of representation. Neurons only perform numerical operations based on their electrical activity.
2. The behaviour of the system emerges from the application of simple and local rules, rather than from the execution of a program in a central unit.

Then in the early 90s emerged from the foundational work of Maturana and Varela on the concept of autopoiesis a new paradigm christened enactivism. In analogy to the description of the cognitivist program given earlier, Varela, Thompson and Rosch summarise the new research program as answers to the same three questions [Varela et al., 1992]:

Question 1: What is cognition?

Answer: Enaction: A history of structural coupling that brings forth a world.

Question 2: How does it work?

Answer: Through a network consisting of multiple levels of interconnected, sensorimotor subnetworks.

Question 3: How do I know when a cognitive system is functioning adequately?

Answer: When it becomes part of an ongoing existing world (as the young of every species do) or shapes a new one (as happens in evolutionary history).

A set of new important ideas comes with the enactive view: mainly the emphasis that a cognitive system is foundamentally a *dynamic system*, *coupled* with its environment both at the cell's level with *autopoietic self-regulation* and the nervous system's level with the presence of *sensorimotor loops*.

In *Second order cybernetics and enactive perception* (2005), Bishop and Nasuto develop a position that shares several ideas with the enactive view. In particular they argue that computationalist approaches are subject to the "external observer fallacy" and fundamentally lack of coupling with the environment (they relate to first order cybernetic), while a cognitive system can only be understood in terms of second order cybernetic involving dynamic system theory and sensorimotor theory of perception [O'Regan and Noë, 2001]. This position constitutes the theoretical background for the study of Stochastic Diffusion Search.

1.2 Objectives

The present thesis find its roots in two ideas:

1. It has been argued [Nasuto et al., 2009] that one of the strengths of SDS as a model for neural activity is its ability to support multiplexing (by the coding of information in inter-spike intervals instead of the simple mean firing rate of neurons) hence allowing non-linear knowledge representation (through the formation of predicates of arity superior to zero). However until now, the multiplexing ability of SDS has not been fully exploited. The first idea was thus to experiment a version of SDS using composite-hypotheses made of arrays of values.
2. Computer game-playing is historically the field of study of choice for AI. The application of SDS to this problem is conceivable given that it essentially constitutes a search problem in the space of possible games. Also, the combinatorial structure of the problem (search through a game-tree) is well-suited to apply the first idea: each agent's hypothesis could be composed of a combination of moves. The second-idea was thus to apply SDS to computer game-playing.

These two ideas were at first simple intuitions, but they led to the formulation of a concrete research question: "Can Stochastic Diffusion Search perform non-trivial game-playing?" A positive answer would lead to the application of SDS to a new range of problems, potentially providing a new variant of the heuristic. Conversely, a negative answer would give new insights on the limitations of SDS.

The methodology adopted has consisted, first, in the extensive study of the two fields concerned: computer game-playing (chapter 2), and Stochastic Diffusion Search (chapter 3). This led to the conviction that although the problem is more complex than suggested by the first intuitions, a method inspired from existing techniques both in computer game-playing (Monte-Carlo Tree Search) and in SDS (Coupled SDS) can perform game-tree exploration (chapter 4). Finally the proposed method has been applied to practical game-playing on the game of Hex (chapter 5).

Chapter 2

The problem addressed: Computer Game-Playing

In the presentation of the objective of the thesis in the introduction, the term of computer game-playing was used without further precision. Of course not all types of games are targeted here; for example video-games are not concerned. The archetypes games that are of interest are Chess, Checkers, Othello, Go, etc. They can be fully characterised by the name of *finite two-person zero-sum games with perfect information* and will be reffered to as *combinatorial games* in the following.

2.1 Combinatorial Games

Before introducing the existing methods in computer game-playing, the present section exposes a fundamental property of the combinatorial games—namely their determinateness—in a weak sense first with Zermelo's theorem and in a stronger sense then with the Minimax algorithm.

2.1.1 Zermelo's Theorem

An important result about finite two-person zero-sum games with perfect information is Zermelo's theorem, generally formulated as follows:

Zermelo's theorem: Every finite two-person zero-sum game with perfect information is determined.

In the original paper [Zermelo, 1913], Zermelo was considering Chess and "all similar games of reason, in which two opponents play against each other with the exclusion of chance events" (translation from [Schwalbe and Walker, 2001]). In particular, he was interested in the question: "can the value of an arbitrary position, which could possibly occur during the play of a game as well as the best possible move for one of the playing parties be determined or at least defined in a mathematically-objective manner?" This interrogation was justified by the observation that there definitely exists situations in

which the value of the position can be determined: "the so called Chess problems, i.e. examples of positions in which it can be proved that the player whose turn it is to move can force checkmate in a prescribed number of moves." The reasoning he employed in the paper is based on set theory, and is outlined in the following quote from [Schwalbe and Walker, 2001]:

> "To answer the (...) question, he states that a necessary and sufficient condition is the nonemptiness of a certain set, containing all possible sequences of moves such that a player (say White) wins independently of how the other player (Black) plays. But should this set be empty, the best a player could achieve would be a draw. So he defines another set containing all possible sequences of moves such that a player can postpone his loss for an infinite number of moves, which implies a draw. This set may also be empty, i.e., the player can avoid his loss for only finitely many moves if his opponent plays correctly. But this is equivalent to the opponent being able to force a win."

In other words, as summarised by [Hart, 1992, p. 30]: "In Chess, either White can force a win, or Black can force a win, or both sides can force a draw." Here it is important to realise that the theorem is not only saying that the possible outcomes in chess are a win for White, a win for Black or a draw. What it says is that the outcome is *determined* from the beginning (given that players play optimally). This result is conceptually important because it implies that what distinguishes Chess from, for example tic-tac-toe, is simply the size of the space of possible games. A result that led Zermelo to conclude his article with the remark: "The question as to whether the starting position p_0 is already a winning position for one of the parties is still open. Would it be answered exactly, Chess would of course lose the character of a game at all."

2.1.2 Minimax Theorem

Schwalbe and Walker argue that although Zermelo's theorem is commonly referred to as "the historically first theorem of Game Theory" [Hart, 1992], it is often surrounded by much confusion. In particular, a misconception commonly held (e.g. [Binmore, 1992]) is that Zermelo used backward induction to prove it. This mistaken belief reveals an underlying confusion between Zermelo's and von Neumann's respective contributions to Game Theory: "Note that in Zermelo's paper, contrary to what is often claimed, no use is made of backward induction. The first time a proof by backward induction is used seems to be in von Neumann and Morgenstern (1953)".

Von Neumann's first main contribution to Game Theory was his 1928's paper *Zur Theorie der Gesellschaftsspiele* in which he established the Minimax theorem[1]:

[1] von Neumann is quoted as saying: "As far as I can see, there could be no theory of games (...) without that theorem (...) I thought there was nothing worth publishing until the Minimax Theorem was proved" [Casti, 1996]

Minimax theorem: A minimax equilibrium exists in pure or mixed strategies for every two-person zero-sum game. [Rasmusen, 2007]

What Zermelo's theorem and the Minimax theorem have in common—together with the more general and important concept of equilibrium in Game Theory—is that they all address the same original question: "Is there a natural way to behave for a player involved into a game?" What differentiates them however, is the kind of games they focus on. On the one side, Zermelo was interested in Chess and other combinatorial games, the complexity of which stems from the extremely large number of possible combinations of moves. On the other side, von Neumann was interested in the formalisation of conflictual problems where two persons have to take a simultaneous decision while being conscious that their actions will affect each other. In this case the difficulty does not originates from the combinatorial structure (since each player only performs one action), rather it comes from the simultaneity of the actions. The minimax theorem states that there exists an equilibrium in the sense that *if both players play well*[2], they will tend to stabilize the game by always choosing the same strategy (or mixed strategy: an assignment of probability over the possible actions). The theorem is called Minimax because the strategy in question is the one that minimize each player's maximum loss, or equivalently that minimize their opponent's maximum payoff.

The link between the problems addressed by Zermelo and von Neumann was made only in [Von Neumann and Morgenstern, 1953], where background induction was introduced to extend the Minimax theorem to games with several moves. The general idea is that a game with several moves (Zermelo's concept of a game) can be treated as a succession of one-move games (von Neumann's concept of a game) and the Minimax strategy to adopt for each move can be determined recursively from the end. When applied to combinatorial games such as chess, this idea is even more powerful than Zermelo's theorem because it determines explicitly (in theory) what is the best move to play for each player at each stage of the game (although this information is inaccessible in practice due to combinatorial explosion). This remark is pushed further in section 2.1.4 where the Minimax algorithm is presented, and after the notion of game-tree is introduced in section 2.1.3.

2.1.3 Game-Tree Representation

A convenient way to represent combinatorial games is to build a game-tree in which:

- Each node represents a game position. There are nodes corresponding to the first player's turn (called Max) and nodes corresponding to the second player's turn (called Min).
- Each branch represents an admissible move from the player whose turn it is, to transit from the parent position to the child position.

[2] If one of the player plays poorly, there might be a strategy for the other player that is better for him than his minimax strategy

The tree is build recursively from the root node (labelled Max) by associating to each node a branch for every admissible move from the corresponding player. The construction terminates for a branch when a terminal state is reached, in which case the leaf takes the value:

- 1 if Max wins the game
- -1 if Min wins the game
- 0 if the outcome is a draw

In the case of games where the outcome is determined by counting points (such as Go), it is possible to associate the points corresponding to Max to each leaf instead of the values $\{1, -1, 0\}$. The construction is assured to terminate because the games considered are finite and because only a finite number of moves are available for each player at each turn. In games where an infinite succession of moves is possible (such as chess where it is possible to do and undo the same couple of actions endlessly), a stopping rule is supposed to be defined. An example of a part of a game-tree is shown in Figure 2.1.

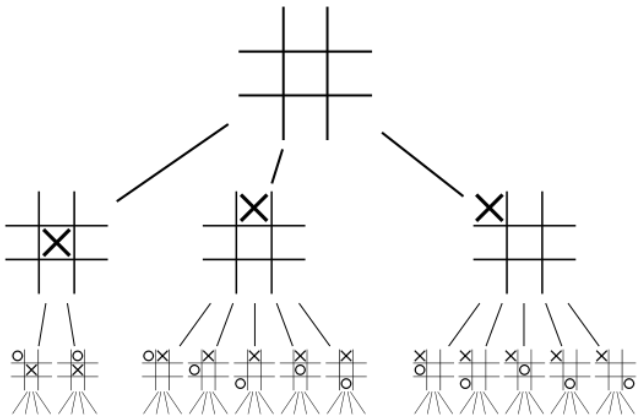


Figure 2.1: The first two plies of the game-tree for tic-tac-toe. The rotations and reflections are eliminated. (*From Wikimedia Commons*).

2.1.4 Minimax Algorithm

The Minimax algorithm is a direct application of the idea of backward induction introduced by von Neumann and Morgenstern to combinatorial games. It leads to the full resolution of such games (in theory) in the sense that it associates a value to every possible position (the outcome from that position if the players play optimally) and indicates for every position the best move(s) to play for the current player.

Minimax algorithm: Method to entirely determine a combinatorial game.

The validity of the algorithm can be proven by mathematical induction on the number of moves of the game as sketched in the following.

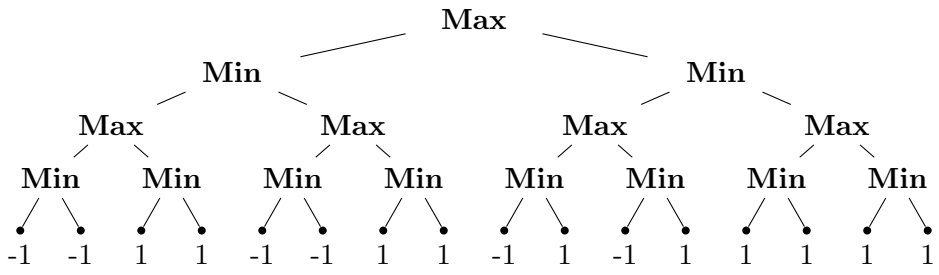
Basis Show that the Minimax Algorithm holds for n=1.
The base case of the reasoning is simply the minimax theorem applied to one-move games with perfect information[3], i.e. games in which each player can perform only one move (from a finite set of possibilities) and in which the second player knows what move the first player chose. In such a situation, the second player always chooses the move that maximises his own outcome (rational agent hypothesis). However the best move the second player can perform is dependant on the move chosen by the first player. Since the game is zero-sum, the intuition suggests (and the minimax theorem confirms) that the best move the first player can perform is the one that minimizes the best possible outcome for the second player. For example in the situation illustrated by the game-tree in Figure 2.2a, if Max plays move a, Min will play move c which leads to the outcome -1 (win for Min), whereas if Max plays move b, whatever Min plays leads to a draw. Hence Max's best move is b. The game can thus be totally determined as shown in Figure 2.2b.

(a) Unsolved game (b) Solved game

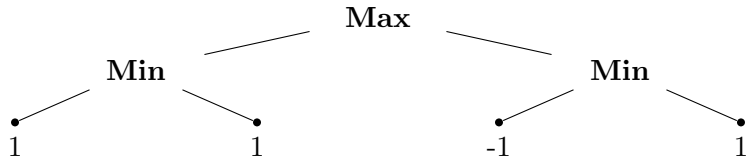
Figure 2.2: Resolution of a one-move game with perfect information

Inductive step Show that if the Minimax Algorithm holds for n-move games, then it holds for (n+1)-move games.
Let one assume that every n-move combinatorial game can be entirely determined and consider G, an (n+1)-move game. G's game-tree can be seen as a n-move game-tree, the leafs of which have been replaced by 1-move game-trees. But since all the corresponding 1-move games can be determined by applying the Minimax theorem as it is done in the base case, their game-trees can be replaced by the value of their root node. This is equivalent to writing G as a n-move game and since it is assumed that every n-move combinatorial game can be entirely determined, G can be entirely determined. For example, the 2-move game shown in Figure 2.3a can be reduced to a 1-move game by applying the Minimax theorem to every possible second moves games as shown in Figure 2.3b, which lead to the complete resolution shown in Figure 2.3c.

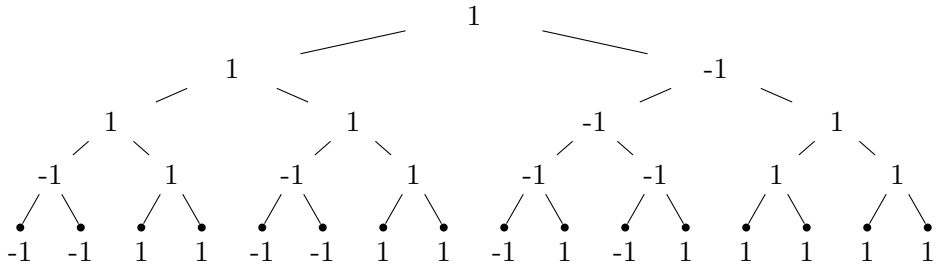
[3]In fact, the proof also holds with the base-case n=0 (zero-move game) as developed in [Von Neumann and Morgenstern, 1953] but the notion of zero-move game is less intuitive than the notion of one-move game.



(a) Unsolved game



(b) Reduced game (after one step)



(c) Solved game

Figure 2.3: Resolution of a combinatorial game: if the players play optimally, Max is the winner (value 1 at the root node).

Zermelo concluded his paper with the remark that chess is determined, and that if the question as to whether the starting position is a winning position for one of the two player were to be answered, chess would lose the character of a game at all. It now appears that the Minimax theorem proves that combinatorial games are determined in an even stronger sense than suggested by Zermelo's theorem because it gives an explicit method to determine what is the best move to play in every situation of the game. Not surprisingly then, this result led von Neumann and Morgenstern to similar observations than Zermelo about chess:

> "... if the theory of Chess were really fully known there would be nothing left to play. The theory would show which of the three possibilities [win for white, win for black, or draw] actually holds, and accordingly *the play would be decided before it starts.*" (emphasis added)

However, von Neumann and Morgenstern note that their result is purely theoretical, and that the interest of the game lies elsewhere than simply in the question as to whether the first of the second player is in a winning position at the beginning[4]:

> "But our proof, which guarantees the validity of one (and only one) of these three alternatives, gives no practically usable method to determine the true one. This relative, human difficulty necessitates the use of those incomplete, heuristic methods of playing, which constitute 'good' Chess; and without it there would be no element of 'struggle' and 'surprise' in that game."

2.2 The classical approach

The problem of computer game playing was historically one of the first studied in the field of Artificial Intelligence. This observation is not surprising if one reminds that AI was born under the dominant paradigm of Computationalism: combinatorial games such as chess are at the same time relatively straightforward to address in computational terms, and the perfect example of human intelligence as an abstract symbol-manipulation. However with the passing of time and the switch of paradigm from Computationalism to Enactivism and Embodiement, the ability to play chess for a computer is no-longer seen as a proof of intelligence. The last Human-Computer chess competition held in 2006 between World Champion Vladimir Kramnik and the program Deep Fritz (ran on a computer containing two Intel Core 2 Duo CPUs) ended once more in a victory for the machine, and the superiority of machines over humans in this domain is now generally agreed as Pr. Newborn from McGill University in Montreal asserted to the New York Times by declaring: "The science is done". Sections 2.2.1 and 2.2.2 offer a brief review of how this feat was achieved.

2.2.1 Historical notes

In their introductory book to AI, *Artificial Intelligence: A Modern Approach* (2010), S. Russel and P. Norvig write:

> "Chess was one of the first tasks undertaken in AI, with early efforts by many of the pioneers of computing, including Konrad Zuse in 1945, Norbert Wiener in his book *Cybernetics* (1948), and Alan Turing in 1950 (see Turing et al., 1953). But it was Claude Shannon's article *Programming a Computer for Playing Chess* (1950) that had the most complete set of ideas..."

What is striking when reading Shannon's article is that it indeed introduced most of the questions computer-chess playing has been facing since these early years. In *The Embodied Mind*, Varela, Thompson and Rosch make the observation that "virtually all the themes in present-day debates were already introduced in the formative years of

[4]For example in Hex, it is proven that the first player is in a winning position and yet the game is still being played by a lot of enthusiasts.

cognitive science from 1943 to 1953" (that they call "the Foundational Cloud"). It appears that Shannon's article *Programming a Computer for Playing Chess* definitely belongs to this foundational cloud. In particular, Shannon introduced a distinction between two classes of algorithms that he labelled type A and type B. A type A algorithm is a direct application of the Minimax algorithm presented in section 2.1.4 in a brute force manner, although since the full exploration of the game-tree is impossible due to combinatorial explosion (even for very fast computers), the exploration is cut to a certain depth and all the possible intermediary positions at this depth are evaluated by using an evaluation function. Shannon argues that such an algorithm would suffer from two main weaknesses. First it would still be subject to combinatorial explosion: Shannon estimates the time necessary to explore the tree at a depth of three moves for each side to 16 minutes, even in the "very optimistic" hypothesis that each position is evaluated in one microsecond. Second it does not include any condition about what he calls "quiescent positions": he explains that the evaluations should be performed in a stable (or quiescent) phase of the game, or all the consequences of the chosen move might not be taken into account (the simplest example being the evaluation of the game after a move that gives an advantage of a queen, while it would in fact be directly followed by the other queen being taken back and thus constitute a simple queen's exchange.) He thus introduces the notion of type B algorithm inspired from human play. A type B algorithm would only evaluate "reasonable positions" and examine them "as far as possible (...) where some quasi-stability has been established." As section 2.2.2 attempts to show, this different questions are at the heart of every modern chess-playing program (and many programs for other games).

2.2.2 Modern Game-Playing

For most of the combinatorial games (notably Chess, Checkers or Othello), the main framework for computer playing consists in the combined use of the Minimax algorithm with a domain dependant evaluation function for intermediary positions as characterised by Shannon's type A class of algorithms. Then, the historically first main improvement added to this basis was the Alpha-Beta pruning technique[5] "that appears to have been reinvented a number of times" between 1956 and 1972 [Newell and Simon, 1976]. Alpha-Beta pruning returns the same move as the Minimax algorithm would, but prunes away branches that cannot possibly influence the final decision. For a game-tree with a branching factor of b, and a search to a depth of d plies, the complexity of the Minimax algorithm is in $O(b^d)$ while the complexity of Alpha-Beta pruning is in $O(b^{3d/4})$ for a moderate b in the general case.

The next improvements that can be made to the algorithm while still returning the same move as Minimax are based on the fact that Alpha-Beta pruning is more efficient when the best move at a node is explored first, because it is the most likely to produce

[5]The material in the following paragraphs is based on [Russell and Norvig, 2010] and [Schaeffer, 1989]

a cutoff (in the ideal case the complexity of the algorithm is in $O(b^{d/2})$). This idea is used in several heuristics, such as:

The killer heuristic: It consists in trying in first a move that produced a cutoff in another branch of the game-tree at the same depth (called a "killer move"), assuming that it is likely to produce a cutoff in other branches as well.

Ordering functions: It consists in choosing the move to evaluate first according to some domain knowledge. For example Russel and Norvig mention a fairly simple ordering function for chess which consists in trying captures first, then threats, then forward moves, and then backward moves and state that it leads to a performance within about a factor of 2 of the best-case result.

Iterative deepening approach: It consists in repeating Alpha-Beta pruning on a search-tree whose depth is incremented at each iteration. It seems to be highly redundant but in fact it can lead to better results than typical depth-first search Alpha-Beta pruning because the knowledge acquired in previous iterations about possible cutoffs can be used in subsequent iterations. It is also a good solution to the problem of time management in a limited time context.

These heuristics combined with other techniques such as the use of transposition tables (that keep in memory the positions already visited during the search in order to avoid to re-examine an entire subtree when a position is encountered a second time) reduce efficiently the size of the search tree. The experience showed that contrary to what Shannon speculated with his first critic of type A algorithms, the use of such search extensions is preferable to performing early cutoffs simply based on an evaluation function—a technique referred to as "forward pruning"—because it is much more robust. However, Shannon's second critic based on the notion of quiescent positions appeared to be well-founded and many game-playing programs (and in particular every good chess-playing program) address it with the concept of "quiescence search" that extend of a few moves the "noisy" positions (i.e. unstable positions). This problem also appeared to be related to another one called the "horizon effect" that arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable. In such a case the program would have a tendency to delay the damage behind the horizon of its search (sometimes at the expense of other sacrifices), while the best strategy might be to receive the damage quickly.

Beyond the core principles described in the previous paragraphs, most of the top-rated game-playing programs also include a lot more domain specific knowledge. For example in many games the evaluation functions perform poorly in the beginning phase which is better handled with the use of opening books (that dictate the move to play according to the opponent's move until a certain depth based on the observation of Grand Masters games). Similarly in some games such as Chess or Go, the endgame is a delicate phase where the risk to perform a blunder is high; there the use of endgame tablebases (either programmed manually or determined by exhaustive search) is often made. Among the

different tools provided by the field of AI, Machine Learning is also commonly used to determine parameters that might be used in the algorithm (for example in the evaluation function) or to refine their value according to the phase of the game (for example in Othello, corners are more important in the opening and early midgame than in the endgame). This can be done either by reviewing a large database of professional games or by playing many games against itself or other people or programs.

2.2.3 Limits

The approach described in the previous section was historically developed for Chess, but was straightforward to extend to other games since the concepts of Minimax algorithm and evaluation function are very general. Hence it was successfully applied to various games, most notably to Checkers and Othello. However, two main limits to this classical approach appeared while extending it to other games.

The first limit to the classical approach is that it requires a lot of domain knowledge; principally in the conception of the evaluation function but not only (for example in the move ordering function if this heuristic is used to speed up the Alpha-Beta search). The quality of the domain knowledge integrated appears to be a crucial factor in the strength of a program, and thus the conception of a competitive program necessitates the collaboration of experts of the game. Another related problem is that well-performing evaluation functions are sometimes highly time consuming—which is in direct contradiction with the idea to evaluate a great number of possible positions. For example in the game of Hex (see chapter 5), an interesting evaluation function proposed originally by [Shannon, 1953] consists in representing Hex positions by an electrical circuit, and determining the good positions for a player as the ones that minimize the resistance between his two sides of the board. This idea can then be improved by integrating a concept of *virtual connection* [Anshelevich, 2000], but the resulting evaluation function is too slow to be applied to deep Minimax search. Fortunately in the case of Hex, this led to a new approach based on Automatic Theorem Proving which resulted in the realisation of the best Hex playing program in 2000 called Hexy: "We offer a new approach, which results in superior playing strength. This approach emphasizes deep analysis of relatively few game positions." The second limit to the classical approach is that its performance drops down for games with high branching factors. The archetype game suffering from this problem is Go: the regular board size for professional playing is 19*19 and stones can be placed everywhere; this makes a starting branching factor of 361 which is prohibitive for the use of Minimax.

The two main limits presented here have been partially solved in several games (in Go in particular) by the use of a new range of methods based on Monte-Carlo ideas. Section 2.3 introduces these methods.

2.3 Monte-Carlo methods

Monte-Carlo methods are a class of computational algorithms that rely on repeated random sampling to compute their results [Hubbard, 2010]. An elegant illustration of this idea can be seen in the Buffon's needle experiment. The French naturalist Buffon posed in 1733 the following question: "Suppose we have a floor made of parallel strips of wood, each the same width, and we drop a needle onto the floor. What is the probability that the needle will lie across a line between two strips?" A bit of calculation show that if the length l of the needle is smaller than the width d of the strips of wood, the probability that it lies across a line is $P = \frac{2l}{d\Pi}$ from which it follows that $\Pi = \frac{2l}{dP}$. Hence, a Monte-Carlo method to estimate Π could consist in dropping a large number of times a needle on a floor made of wood strips and using the evaluated probability that it crosses a line between two strips in the previous formula. Sections 2.3.1 and 2.3.2 show how this kind of idea can be used in computer game-playing.

2.3.1 The Expected-Outcome model

The idea to use Monte-Carlo methods in computer game-playing was introduced in 1990 by B. Abramson in his paper *Expected-outcome: a general model of static evaluation*. According to the expected-outcome model, "the proper evaluation of a game-tree node is the expected value of the game's outcome given random play from that node on". Put differently, a good move is a move that lead often to a win if the rest of the game is played randomly. This method seems very crude at first but appeared to lead to interesting results. In particular, Abramson applied it to a 6*6 version of Othello and demonstrated its ability to outduel a standard Othello evaluator. After this first result, the idea to use Monte-Carlo methods in computer game-playing, and in particular in Go, started to spread. In 2004, B. Bouzy and B. Helmstetter presented two Go programs developed by a Monte-Carlo approach, OLGA and OLEG, in their paper *Monte-Carlo Go Developments*. Their conclusion were similar to Abramson's: "the results of [these programs] against knowledge-based programs on 9x9 boards are promising."

Another reason that makes the expected-outcome model very interesting is that it solves (at least partially) the first limit to the classical approach discussed in section 2.2.3. Contrary to the use of evaluation functions highly based on domain knowledge, the only requirement to perform an evaluation based on the expected-outcome is to know the rules of the game, in order to be able to simulate random play and determine the winner at the end. Both Abramson and Bouzy and Helmstetter were aware of this. Abramson's abstract states that: "overall, the expected-outcome model of two-player games is shown to be precise, accurate, easily estimable, efficiently calculable, and *domain-independent*" (emphasis added), and Bouzy and Helmstetter write in their conclusion that their program "uses very little domain-dependent knowledge (...). When compared to the knowledge-based approaches, [the Monte-Carlo based] approach is very easy to implement."

However, the expected outcome model still suffer from the second limit to the classical approach: it cannot be combined to deep Minimax search on large branching factor games. Hence it results in programs that have a good overall strategic sense (they tend to choose moves that have a good probability to lead to a win), but they are weak tactically (they are unable to predict the short term consequences of the moves). Section 2.3.2 introduces an improvement of the expected-outcome model that addresses this problem.

2.3.2 Monte-Carlo Tree Search

In the last decade, the strength of Go-playing programs has radically increased. [Russell and Norvig, 2010] states that "up to 1997 there were no competent Go programs. Now, the best programs play most of their moves at the master level". This revolution is partly due to the adoption of Monte-Carlo based evaluation as described in section 2.3.1, but also to the use of optimised game-tree exploration techniques exemplified in the Upper Confidence bounds applied to Tree (UCT) method.

UCT was introduced in [Kocsis and Szepesvári, 2006] as a method that "applies bandit ideas to guide Monte-Carlo planning". The crux of the idea in UCT is that contrary to the evaluation functions used in the classical approach (that estimate the value of a position in one calculation), Monte-Carlo based evaluation is performed by the repetition of rapid game simulations a great number of times. It is indispensable to perform many game simulations to get a good evaluation, however there is no requirement on the order in which to perform them. For example if two positions have to be evaluated, it is possible to perform 1000 game simulations for the first one and then 1000 game simulations for the second one, or it is possible to alternate: one for the first one, one for the second one, etc. until 1000 simulations have been performed for each. The two situations might seem identical, but they are not. In the second case, the information grabbed during the evaluation of each position can be used to orient the planning: for example it might appear after 100 simulations only that the first position gets much better results than the second one and thus that the planning should concentrate on this position. This is roughly what the UCT method does as explained in [Kocsis and Szepesvári, 2006]:

> "Recently, Monte-Carlo search proved to be competitive in deterministic games with large branching factors, viz. in Go. (...) Intriguingly, Monte-Carlo search algorithms used by today's games programs use either uniform sampling of actions or some heuristic biasing the action selection probabilities that come with no guarantees. The main idea of the algorithm proposed in this paper is to sample actions selectively."

To find a natural way to sample action selectively, Kocsis and Szepesvári draw an analogy between the situation of a gambler facing a row of slot machines in a casino (a multi-armed bandit) and the problem of move selection in a game-tree exploration

algorithm. When a gambler is facing a multi-armed bandit, he wishes to find the machine that has the maximum expected-reward in order to maximize his own reward. But since he knows nothing about the machines, the only solution he has is to try different ones to get information about them (exploration), and then focus on the machines that seem to give the maximum reward (exploitation). Similarly, when a game-tree exploration algorithm tries to determine what is the best move to perform for a player, it has no a priori information about the moves and need to *explore* them, but after several evaluations it starts to have some idea of what are the best moves and it can *exploit* them. In fact the multi-armed bandit problem constitutes the quintessence of the *exploration-exploitation* dilemma which is met in particular in the field of reinforcement learning. The UCT method uses UCB1 (Upper Confidence Bounds 1 studied in details in [Auer et al., 2002]) as "a simple, yet attractive algorithm that succeeds in resolving the exploration-exploitation tradeoff" [Kocsis and Szepesvári, 2006].

At this point, it is important to understand that the advantage of game-tree exploration techniques such as UCT is not only to speed up the selection of the best move at the root node of the tree. In fact, the balance between exploration and exploitation can be employed at every nodes and the gain is extremely important, as stated by Kocsis and Szepesvári:

> "In order to motivate our approach let us consider problems with a large number of actions and assume that the lookahead is carried out at a fixed depth D. If sampling can be restricted to say half of the actions at all stages then the overall work reduction is $(1/2)^D$. Hence, if one is able to identify a large subset of the suboptimal actions early in the sampling procedure then huge performance improvements can be expected."

Contrary to Minimax and Alpha-Beta pruning that have properties of depth-first search, or iterative deepening approaches that have properties similar to breadth-first search, UCT constitutes a *best-first* search algorithm. It allows much deeper exploration of the game-tree than the early computer-Go programs based on Monte-Carlo evaluation, and for that reason it is able to solve the weakness in tactic that characterized them.

Since the publication of [Kocsis and Szepesvári, 2006], the idea to balance exploration and exploitation in game-tree exploration has become more important and is now referred to under the general name of "Monte-Carlo Tree Search". In [Chaslot et al., 2008], it is proposed that Monte-Carlo tree search might constitute a new framework for Game AI for its domain-independent character. In particular, it is stated that it can be applied effectively to classic board-games (two-player, perfect information), modern board-games (discrete, turn-based, imperfect information, multi-player) and video games (environment in which AI is expected to behave realistically). Chaslot et al. also propose an outline of a Monte-Carlo Tree Search in four phases as illustrated in figure 2.4. The description of these phases is quoted below, as it constitutes an important source of inspiration for the main work of this thesis (chapter 4).

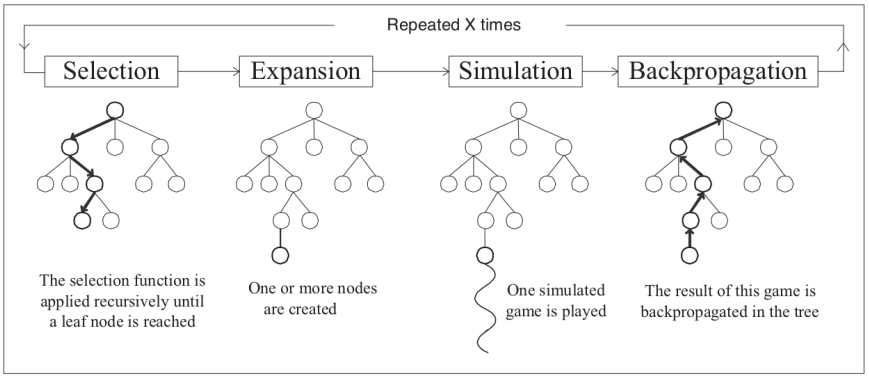


Figure 2.4: Outline of a Monte-Carlo Tree Search. (*From [Chaslot et al., 2008]*).

"**Selection** While the state is found in the tree, the next action is chosen according to the statistics stored, in a way that balances between exploitation and exploration. On the one hand, the task is often to select the game action that leads to the best results so far (exploitation). On the other hand, less promising actions still have to be explored, due to the uncertainty of the evaluation (exploration). Several effective strategies can be found in Chaslot et al. (2006) and Kocsis and Szepesvári (2006).

Expansion When the game reaches the first state that cannot be found in the tree, the state is added as a new node. This way, the tree is expanded by one node for each simulated game.

Simulation For the rest of the game, actions are selected at random until the end of the game. Naturally, the adequate weighting of action selection probabilities has a significant effect on the level of play. If all legal actions are selected with equal probability, then the strategy played is often weak, and the level of the Monte-Carlo program is suboptimal. We can use heuristic knowledge to give larger weights to actions that look more promising.

Backpropagation After reaching the end of the simulated game, we update each tree node that was traversed during that game. The visit counts are increased and the win/loss ratio is modified according to the outcome."

Chapter 3

The metaheuristic employed: Stochastic Diffusion Search

There are two main aspects that make Stochastic Diffusion Search an interesting object of study. First it is a well characterised search and optimisation method that has been applied successfully to many problems such as text search [Bishop, 1989a], object recognition [Bishop and Torr, 1992], feature tracking [Grech-Cini and McKee, 1993], mobile robot self-localisation [Beattie and Bishop, 1998] and site selection for wireless networks [Whitaker and Hurley, 2002]. Second, it offers an original and inspiring model for neural activity.

3.1 Search and Optimisation

In a very broad sense, formulating a problem consists in defining a goal and a set of available actions. In this context, solving a problem is equivalent to *searching* in the set of available actions the one that leads to the goal. Similarly if a function F is introduced to measure the value of the actions, solving a problem is equivalent to finding the action with the *optimal* value. In fact, an appropriately formulated problem can always be reduced to search in a space of possible solutions, or optimisation of an objective function.

3.1.1 Overview

The development of optimisation techniques constitutes a very large part of applied mathematics. One of the first historical example was Newton's method that can be applied to find the optimum of a reasonably well-behaved real-valued function whose second derivative is defined. The idea is to estimate the zero of the derivative function in an iterative way: the process starts by formulating a guess x_0 of the zero and then consists in calculating a better approximation by taking the intersection of the derivative's tangent in x_0 with the x-axis as illustrated in figure 3.1. More precisely, the optimum of a function f can be approximated by calculating iterated values of the sequence (x_n)

defined as: $x_{n+1} = x_n - \frac{f'(x_n)}{f''(x_n)}$. This constitutes a rather good optimisation process as it offers a quadratic convergence toward the optimum.

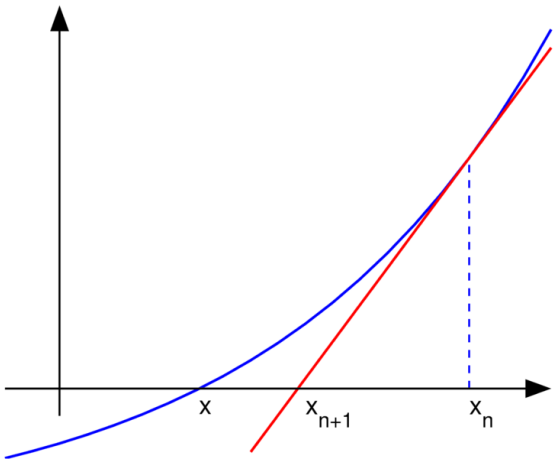


Figure 3.1: One iteration of Newton's Method: x_{n+1} constitutes a better approximation of the blue curve's zero x than x_n (*From Wikimedia Commons*).

Another important example that illustrates the diversity of the field of optimisation is Dantzig's simplex algorithm. This algorithm is concerned with problems of linear programming, i.e. problems that can be expressed in linear terms typically as follow:

- Maximise the objective function: $f(x) = c^T.x$ (a linear combination of the problem's variables $(x_1, x_2, ..., x_n)$)
- Under the constraints: $A.x < b$ (a set of linear inequalities on the problem's variables)

The simplex algorithm is based on the observation that the set of linear inequalities delimits a polytope of a feasible region, and that due to linearity the objective function will take its optimum value on one of the vertex of the polytope. Hence the algorithm operates by going through the vertices of the polytope in an increasing direction each time until it reaches the optimum solution.

A common point of the two methods just discussed is their iterative character; this constitutes indeed a general feature of most of the optimisation techniques. However they both appear to be very specific to their respective scope: Newton's method is restricted to problems with real-valued objective functions whose second derivative can be calculated and that are sufficiently well-behaved while the simplex algorithm can be applied only to linear problems. In fact, designing general optimisation methods is not easy as real-world problems have various and complex objective functions, with large and multidimensional domains of definition, either continuous or discrete. A solution adopted by the computer

science community is the use of heuristics, i.e. experience-based methods that do not necessarily return the absolute optimum but that are able to provide a good solution in a reasonable amount of time. When heuristics are expressed in sufficiently general terms to be applied to almost any problem, they are called metaheuristics; Stochastic Diffusion Search is one of them.

3.1.2 Nature Inspired Heuristics

One of the main sources of inspiration for the conception of heuristics has been the observation of natural phenomenons. The best-known instances are briefly presented in the following.

Simulated Annealing

A first example of a nature inspired heuristic is the Simulated Annealing (SA). It was first proposed in [Kirkpatrick et al., 1983] and its main goal is to address the problem faced by local search methods. These methods rely on the assumption that the search spaces of real-world problems have some form of continuity such that a candidate solution must always be surrounded by a better solution unless it is already optimal. However the drawback of this approach is that the optimum found is not guaranteed to be the global optimum of the objective function.

In the SA the value of a solution through the objective function is called its energy, and the goal is to minimize it. The main contribution of the method is the introduction a concept of *temperature* that allow the current solution to escape local minima. During the iterated procedure, a neighbour of the current solution is selected and the two following situations are considered:

1. If the energy of the neighbouring solution is lower than the one of the current solution, then the neighbouring solution is adopted.
2. If the energy of the neighbouring solution is higher than the one of the current solution, then the neighbouring solution can still be adopted with a probability proportional to the temperature and inversely proportional to the difference of energy between the two solutions.

At the beginning of the procedure the temperature is set high and the current solution moves in the search space with an almost-uniform probability. However as the temperature is decreased, the ability of the current solution to climb high energy levels also decreases and it tends to stay in the low-energy areas of the search space. In the end if the temperature is decreased sufficiently slowly, the solution selected will be the global minimum or a good approximation of it.

Particle Swarm Optimisation

A second example of a nature inspired heuristic is the Particle Swarm Optimisation (PSO) introduced in [Kennedy and Eberhart, 1995] as an extension of Reynolds' work on flocking behaviours [Reynolds, 1987]. PSO considers a population of solutions randomly initialised in the search space and attributes them a "velocity". At each iteration, the particles are moved according to their velocity, and the particle's velocities are updated so that each particle is attracted toward both the population's and its own best visited solutions. The resulting behaviour is a flock of solutions that tend to progress toward good values of the search space. In particular, the small displacements of the solutions between each iteration are performing a form of local search while the global inertia of the flock over several iterations allow distant areas to be explored and local optima to be escaped.

Genetic Algorithms

A characteristic of the two heuristics presented before is that their problem-solving abilities are not directly present in the phenomenons modelised: the industrial process of annealing and the flocking behaviour of birds can be seen as performing optimisation only in a very metaphorical way. Conversely, when looking for real problem-solving abilities in nature, one of the most obvious response is the principle of evolution through natural selection. This idea is at the basis of the Genetic Algorithm (GA) heuristics where a population of solutions is evolved potentially toward a global optimum. There are many ways of implementing a GA and a good overview can be found in [Fogel, 2006]; a typical GA starts by initialising a population of random solutions and then iterates the two steps:

Selection The population of solutions is tested through the objective function (called the fitness function in this context) and only the best-fitted solutions are kept.

Reproduction New solutions are created by using different mechanisms. In analogy to the DNA-basis of natural selection, the solutions of the problem are commonly represented as chromosomes (strings of binary digits), and are submitted to mutation (the random modification of one digit of the string) or crossover (the swap of the digits at a certain point in the string between two parent solutions)

If a concept of distance is introduced on the binary representation of the solutions, it appears in fact that mutation of solutions constitutes local search (a solution close from the parent ones is considered) while crossover is a way to explore more distant areas of the search space (in order to escape local minima). Moreover, the use of the genetic operator crossover rely on the assumption—based on the analogy with natural selection but not necessarily true—that a solution halfway between two good solutions is likely to be good.

Ant Colony Optimisation

Another particularly inspiring example of problem-solving ability observable in nature is the swarming behaviour of social insects such as ants. Ant Colony Optimisation (ACO) aims at solving problems which can be reduced to finding a short path in a graph. The archetype problem that can be expressed in such terms is the travelling salesman problem (in which a travelling salesman wishes to find the shortest way to visit once and only once a given number of cities) but numerous planning problems are also concerned. The algorithm introduced in [Dorigo, 1992] implements artificial ants that wander randomly on the studied graph and that leave behind them a trail of pheromones when they reached their objective (for example a food source). The pheromone trails have two properties:

1. They attract other ants. Hence the paths that lead to the objective are followed more often than other paths and are reinforced.
2. They evaporate over time. Hence, short paths are more likely to be followed because they evaporate for a shorter time.

Similarly to the previous examples, ACO allows both local search (by the fact that ants never follow exactly the track of other ants) and global search (by the fact that some ants always explore new paths). A very interesting property of the ACO which was not present in any other example discussed so far is the fact that the best solution is not found by one individual of the population, but emerges from the global behaviour. The best solution is constituted by all the portions that are followed by the largest number of ants but it might be that no ant is following it entirely. This property is common to SDS and will be discussed in more details in the following sections.

3.1.3 No Free Lunch Theorem

Before investigating in details the SDS metaheuristic, an important theorem for the theory of search [Wolpert and Macready, 1995] and optimisation [Wolpert and Macready, 1997] need to be presented: the No Free Lunch (NFL) theorem.

No Free Lunch theorem: Any two algorithms are equivalent when their performance is averaged across all possible problems [Wolpert and Macready, 2005].

The NFL theorem is based on the observation that when no knowledge about the search space of the problem being solved is available, there is no better way to proceed than by performing successive evaluations of random candidate solutions. Of course when some knowledge is available, it can be used to improve the efficiency of the search procedure. This is the case of local search techniques which (as already stated) assume some form of continuity in the space of solutions and the presence of only one optimum. However, an algorithm using such knowledge becomes specialised to one type of problem at the expense of performing worse on other types of problems. In practice, the NFL theorem does not imply that it is useless to conceive other algorithms than the random search but simply nuances and put into perspective the concept of “good” search procedure.

3.2 SDS as a search and optimisation method

Since its introduction in 1989 as a method for pattern classification [Bishop, 1989b], SDS has become over the years "a well characterised robust swarm intelligence global metaheuristic, that can efficiently solve search and optimisation problems with compositional structure" [Bishop, 2007]. Several complete descriptions of SDS have already been given, in particular in [De Meyer, 2004] and [Williams, 2010]. An overview of the main results is presented in the following.

3.2.1 Prologue

The ACO heuristic is inspired from a form of stigmergic communication (indirect communication through the modification of the environment) present in many species of ants: successful ants secrete a trail of pheromones which indirectly indicates to other ants the way to the food source. Conversely, a particular species of ant called *Leptothorax Acervorum* uses a form of communication called "tandem calling": a successful ant contacts an unsuccessful ant and directly brings it to the found source. In SDS, the agents deployed to explore the search space use a form of direct communication similar to tandem calling. A second property of SDS which distinguishes it from any other existing heuristic is the principle of partial evaluation. As argued by De Meyer: "SDS adds an important concept to the 'library of ideas' in optimisation: the principle of partial evaluation." [De Meyer, 2004].

An analogy commonly put forward to clarify the two principles of direct communication and partial evaluation is the restaurant game. In this illustration, a group of people dines every day in a given number of available restaurants. The first day, each person chooses a restaurant and a meal available in that restaurant. The day after, the persons who enjoyed their meal decide to go back to the same restaurant while the persons who did not enjoy their meal contact another person: they follow the person they contacted if he liked his meal or otherwise they just choose a new restaurant. After several days of repeating this process, most of the people of the group are expected to go dining in the same restaurant, and this restaurant is supposed to be the best one. In this example the evaluation of a meal in a restaurant constitutes a partial evaluation of the restaurant, and direct communication happens when the disappointed persons contact another person.

3.2.2 The algorithm

The standard algorithm of SDS is presented in table 3.1. The different phases of the algorithm are described in the following.

Initialise The initialisation phase commonly consists in attributing a random hypothesis to each agent of the population. A deterministic uniform repartition of the agents

```
Initialise(Agents)
repeat
    Test(Agents)
    Diffuse(Agents)
until (Halting Criterion)
```

Table 3.1: Standard SDS operation

in the search space is also possible or some a priori knowledge can be used to bias the attribution of hypotheses.

Test For each agent, a partial evaluation of the hypothesis is performed. This usually requires that "the objective function is decomposable into components that can be evaluated independently" [De Meyer, 2004], typically of the form of a summation. If the partial evaluation is positive the agent becomes active, otherwise it stays inactive.

Diffuse Each inactive agent X contacts another agent Y at random. If Y is active X copies his hypothesis, otherwise X selects a new hypothesis at random (this classical type of recruitment is termed passive; next section introduces other types of recruitment).

Halt In practice, an halting criterion is necessary to determine when to stop the computation. A strong criterion requires that the number of agents pointing at the same hypothesis stabilises and stay bounded for a given number of iterations while a weak criterion requires the same thing only for the total number of active agents.

3.2.3 Variants

In the description of SDS as given in the previous section several aspects of the algorithm have been fixed in a rather arbitrary manner, and can be modified. This is particularly justified when having the NFL theorem in mind stating that an algorithm can never perform well on every problem. Different variants of standard SDS have been introduced, in particular in [Nasuto, 1999] and [De Meyer, 2004].

Local Search

Contrary to all the heuristics presented in section 3.1.2, standard SDS is not thought to perform local search. This is not a problem for the classical applications of SDS: for example in text search, when a string is partially matched at a certain place in the text, there is no specific reason to think that a better match will occur in its neighbourhood. However if local search appears necessary, it can easily be superimposed to the standard structure in a way equivalent to mutation in GA: a small random offset can be introduced during the copying of hypotheses so that slightly different solutions are explored.

Recruitment Strategy

A second way to alter standard SDS is to modify the recruitment strategy. Different mechanisms have been studied in [Myatt et al., 2006].

In the diffusion phase of the standard SDS algorithm, the recruitment strategy consists in the inactive agents contacting other agents at random (strategy termed passive). Of course this is not the only possibility: for example in the recruitment method used by *Leptothorax Acervorum* based on tandem calling, this is the successful ants that contact other ants. This remark suggests a second form of recruitment for SDS termed active. During the diffusion phase each active agent X contact another agent Y at random: if Y is inactive, X gives him his hypothesis.

A third recruitment strategy that has been studied is dual recruitment, which is realised by operating simultaneously passive and active recruitment strategies (either with passive or active priority). It is mentioned in [Bishop, 2007] that "this mix of recruitment mechanisms in a system is considered to be the most biologically plausible and is therefore of special interest."

Relate Phase

The introduction of a relate phase constitutes the last of the principal ways of altering standard SDS. In [Nasuto, 1999], the analysis in terms of exploration and exploitation has shown that SDS is biased toward exploitation. When a good solution is found in the search space most of the agents are rapidly allocated to its exploitation, and if a perfect match exists *all* the agents will eventually converge to it (since the agents allocated to it cannot become inactive any more). However if several solutions exist in the search space and need to be explored, or in the case of a dynamically evolving environment, it might be useful to allocate more agents to the exploration of the search space. This can be realised by the introduction of a relate phase after the diffusion phase in which each active agent contacts another agent at random and deactivate either if the other agent supports the same hypothesis (context-sensitive SDS) or simply if the other agent is also active (context-free SDS).

3.2.4 Analysis

When the use of heuristics first started to spread in the field of computer science in the 80s and 90s, not much importance was given to their mathematical analysis. After all, heuristics were supposed to be experimental methods and were used when exact mathematical techniques could not be applied. It seems that the success of a heuristic during this period was more due to its metaphorical elegance than to any established mathematical property. However when several heuristics started to be available, it became interesting to study their characteristics in more details. Some commonly used heuristics then showed unsuspected weaknesses: this is the case for example of the GA which tend

to converge to local minima [Barrios et al., 1998]. Due to its relative simplicity SDS was well-suited to mathematical analysis, such that today "it has become one of the best characterised metaheuristics" [Bishop, 2007].

Convergence in absence of noise

One of main results about SDS is its global convergence (in absence of noise), established in [Nasuto and Bishop, 1999] by using a Markov Chain model. Two situations are possible:

- "In the presence of the target in the search space all agents will eventually converge on its position."
- "In the situation when the target is not present in the search space (...) the Stochastic Diffusion Search converges in a weak sense, i.e.

$$(\exists a > 0)(\lim_{n\to\infty} Ez_n = a) \text{ "}$$

  where z_n is the maximal number of active agents in the n^{th} iteration pointing to the same position.

It is important to note that the weak convergence when the target is not present in the search space does not rule out the possibility to use the strong halting criteria in practice (section 3.2.2). Although "even after reaching a steady state all possible configurations of agents pointing to the best instantiation of the target as well as to disturbances occur infinitely often", in practice "the algorithm will stabilise for a long enough period thus enabling termination" [Nasuto and Bishop, 1999].

Characterisation of steady state behaviour

This first result has then been improved with the introduction of an Ehrenfest Urn model in [Nasuto, 1999]. The Ehrenfest model was originally conceived to explain the second law of thermodynamics in the terms of statistical physics [Ehrenfest and Ehrenfest, 1907]. In particular it was an attempt to solve the apparent paradox between the statistical possibility that a system in equilibrium moves away from it in an arbitrarily large manner, and the observation that it never happens in practice. This problem is analogous to the problem of weak convergence in SDS.

Convergence in noisy search spaces

A last important result has been the characterisation of the convergence behaviour of SDS under the presence of noise [Myatt et al., 2004]. Two parameters are introduced to define the level of noise:

- α is the Test Score of the single optimal solution. Hence, $(1 - \alpha)$ corresponds to the level of noise affecting the solution.

· β is the Test Score of every non-optimal hypothesis. It defines an "homogeneous background noise".

Intuitively it appears that there must be a function f such that if $\alpha \leq f(\beta)$ then the convergence cannot happen. For example, clearly if $\alpha \leq \beta$ then the solution is indistinguishable from the background noise and cannot possibly be found. Myatt, Bishop and Nasuto establish that the minimum value of α for which there is convergence is:

$$\alpha_{min} = \frac{1}{2 - \beta}$$

3.3 SDS as a model for neural activity

Due to its intrinsically parallel nature and to the plausibility of the one-to-one communication, the idea to implement SDS in a neural network architecture is rather natural. In fact, SDS developed from its inception in a connexionist context: it was originally proposed as an extension of Hinton Mappings that address the problem of stimulus equivalence in pattern recognition [Bishop, 1989a]. Yet this idea brings the thought-provoking and inspiring view that the brain might be a swarm of neurons communicating between each other, rather than a simple computing device.

3.3.1 Generalities

In parallel to the different results that have been presented in the previous section, the study of Stochastic Diffusion Search as a metaheuristic for search and optimisation has revealed several properties that fit well in the framework of the recent theories of cognition.

First, as a Swarm Intelligence process, SDS possesses two important properties:

1. the processing of the information is decentralized, and
2. a global self-organising behaviour—seemingly intelligent—emerges from the application of simple and local rules.

Second, similarly to ACO, the meaning is embedded in the entire population and not simply supported by individual agents. In the case of string matching for example, the position of the pattern after convergence is indicated by the formation of a cluster of agents, possibly dynamically fluctuating (in the case of a partial match, agents will keep exploring the search space while the cluster will globally stay on the best match).

Third, by the very essence of SDS the interaction between agents is more naturally described in terms of communication than computation, thus giving an illustration of a possible new metaphor for cognition.

3.3.2 Nester

The idea to use SDS as a model for a neural network led to the description of NESTER (standing for NEural STochastic nEtwoRk). In [Nasuto et al., 2009], Nester is argued to address important challenges for AI. In particular, it fulfils three of the suggestions given in [Selman et al., 1996]: it offers a revision "of the conventional McCulloch-Pitts neuron model (...) based on recent biological data", it constitutes a "fast general purpose search procedure" and "automatically allocates information processing resources to search tasks" [Nasuto et al., 2009].

Before arguing in favour of Nester, Nasuto et al. review several limits to classical neural networks. In particular, the assumption that information is encoded in the mean firing rate of neurons lead to a fundamental restriction: it only allows linear knowledge representation through a simple distribution of weights. Also, the neglect of the geometrical properties of the neurons combined to the application of feedforward learning lead to a total lack of dynamics.

On the contrary, Nester codes the information in the inter-spikes intervals thus allowing a more complex knowledge representation. And although it operates in discrete time, it has the same dynamical properties as SDS. The architecture of Nester is composed of three layers of neurons: a retina, a layer of matching neurons (the equivalents of the agents in SDS) and a memory layer. When Nester is in operation, the matching neurons formulate hypotheses about the position of the memory on the retina (coded in the inter-spike intervals) and they tend to synchronise their firing rate when the memory is located. This property constitutes another major support to the idea to establish SDS as a model for neural activity: it might explain phenomenons of attention and give a natural answer to the binding problem in neuroscience and philosophy [Nasuto and Bishop, 1998].

3.3.3 Coupled SDS

Coupled SDS (CSDS) was first introduced by Bishop [Bishop, 2003] and was then further developed by Williams [Williams, 2010]. It is an extension of a previous improved version of SDS: Data Driven SDS (DDSDS) [Myatt and Bishop, 2003] and [Williams, 2010].

DDSDS has been conceived to solve the problem of robust hyperplane estimation, which consist in finding the hyperplane best fitting a given data in the presence of unknown outliers: points that should not belong to the hyperplane. A first approach to this problem using SDS could consist in:

1. Formulating an hypothesis about the hyperplane for each agent (selection of n points in n dimensions).

2. Testing the hypothesis by confronting it to a point of the data (if the distance between the selected point and the hyperplane is under a given threshold the agent becomes active, otherwise it stays inactive).
3. Diffusing the hypotheses in the population of agents.

This approach would indeed lead to a rather good hyperplane estimation. However, the process can be improved by testing hypotheses against points belonging to other agents hypotheses rather than points from the entire space. Hence in DDSDS, an agent's hypothesis is composed of two parts: the manifold hypothesis (corresponding to the previous hypothesis: the n points) and a data hypothesis (one point against which other hypotheses are tested). DDSDS has been shown to lead to similar results than one of the best known method for hyperplane estimation: RANSAC.

CSDS is a further modification of DDSDS based on the observation than fundamentally, the two parts of the agents' hypotheses in DDSDS are independent. Hence, CSDS proposes to consider two populations of agents: one supporting the manifold hypothesis and one supporting the data hypothesis. In operation, agents of the two population communicate by pair to form full hypotheses. During the test phase the two agents forming an hypothesis are affected, while diffusions phases are undertaken normally and independently in each population.

There is something very elegant in the way CSDS works. In particular, the dynamical coupling that appears between the two populations of agents evoke the introduction of a meta-level in the communication metaphor at the basis of SDS: each population is driven at the same time by the internal communication between the agents constituting it and the external communication with the other population. CSDS also introduces the possibility to divide complex hypotheses into simpler elements. This is an interesting idea if SDS were to be taken seriously as a model for neural activity: it could give insights about how complex thoughts are decomposed and processed at a low level. Finally, the assumption of the existence of phenomenons similar to CSDS in the brain does not seem biologically unreasonable if one consider that the cortex is organised in relatively distinct structures at different levels: the neuron, the vertical columns and horizontal layers, the Brodmann areas and the 4 lobes.

These reflections are of course highly speculative, but they open a new way in the further study of SDS: the possibility to extend the concept of coupling between populations of agents. The next chapter introduces an example of how this might be done, and what it might be used for.

Chapter 4

Game-Tree exploration using Stochastic Diffusion Search

The following chapter constitutes the heart of the thesis. It presents the proposed answer to the research question, namely how the SDS metaheuristic (chapter 3) can be applied to the problem of computer game-playing (chapter 2).

4.1 Why the problem is non-trivial

It is explained in the introduction that the present thesis originates from two ideas: first using composite hypotheses in SDS, and second taking advantage of the inherent characteristic of SDS—its ability to balance resources allocation between exploration of the search-space and exploitation of the promising solutions—in real-time game-playing where important search-spaces have to be explored efficiently. However this section will argue that the problem is in fact non-trivial, because computer game-playing constitutes a very specific search problem as suggested by the term "adversarial search" used in [Russell and Norvig, 2010].

4.1.1 The chosen framework

The first question that raises once the idea to apply SDS to game-tree exploration has emerged is: "what is the good framework to do it?" It has been shown in chapter 2 that there are two main approaches in computer game-playing: the historical one based on minimax and alpha-beta pruning (which proved to be efficient in games with relatively small branching factors such as Chess or Checkers), and the more recent one based on Monte-Carlo Tree Search (that appeared to be more successful in games with large branching factors such as Go). After consideration of the two possibilities, Monte-Carlo Tree Search appears to constitute a better environment in which to deploy SDS for two main reasons:

· First, the concept of partial evaluation—important in SDS—fits naturally into the notion of random game simulation in Monte-Carlo based approaches. On the

contrary, evaluation functions for non-terminal states proper to Minimax-based approaches are not especially well-suited to be partially performed.

- Second and more importantly, what makes the strength of Monte-Carlo Tree Search techniques is precisely that they address the "exploration-exploitation dilemma" that appears in game-tree exploration (chapter 2)—the problem that constituted the main motivation to apply SDS to this field.

For these two reasons, the rest of the thesis will focus on Monte-Carlo Tree Search and leave aside the historical approach to computer game-playing, based on the combination of Minimax with the use of an evaluation function for non-terminal states.

4.1.2 Wrong Tracks

Once the Monte-Carlo framework has been chosen, it might seem straightforward to apply SDS. A population of agents could be used to hold hypotheses about the different moves available at the current stage of the game and a partial evaluation of each move would be constituted by one random game performed after it. In this way, the agents would tend to cluster around the moves that have the highest probabilities to lead to a win. This solution constitutes a trivial application of SDS to the problem of computer game-playing but it is not satisfying because it suffers from the same drawback as early Monte-Carlo methods: since it is not exploring beyond the first node of the game-tree it is unable to detect short term consequences of the moves and thus lead to a play that is poor tactically.

In order to explore deeper into the game-tree, the idea to use composite hypotheses can be used. For example the naive approach could consist in extending of one move the hypothesis of an agent every time it is evaluated positively, and shorten it from one move every time it is evaluated negatively. In this way the population of agents would be expected to cluster around the regions of interest of the search space. However, this reasoning is fallacious because the evaluation made for each move is not independent of the evaluations made for the previous moves. For example an agent that evaluated by chance a bad move for Min as a good one would tend to attract all the agents in its evaluation of the following move for Max because the resulting probability that the move is good for Max is higher.

The two arguments just exposed led to the conviction that basic SDS in itself is unable to perform non-trivial game-playing. In particular, it seems that the problem of the independence of the move evaluations necessitates that each node of the game tree is studied by a distinct population of agents.

4.1.3 A possible lead

In order to successfully apply SDS to game-tree exploration, a careful examination of the problem is necessary. In particular, the question as to where the exploration-exploitation

dilemma appears need to be answered precisely. In fact, the outline of a Monte-Carlo Tree Search as proposed in [Chaslot et al., 2008] (section 2.3.2) states that the dilemma appears at each node of the tree, during the *selection* phase: "while the state is found in the tree, the next action is chosen according to the statistics stored, in a way that balances between exploitation and exploration.(...) Several effective strategies can be found in (Chaslot et al., 2006) and (Koesis and Szepesvari, 2006)" The approach followed in this thesis was to keep the structure of a Monte-Carlo Tree Search, but to replace the existing strategies to select a move at a node by SDS.

4.2 First step: use of multiple populations

The first main consequence of applying SDS where the exploration-exploitation dilemma really appears (i.e. at each node of the game tree) is the necessity to resort to several populations of agents—one per node studied.

4.2.1 Resolution of a multi-armed bandit

Following [Kocsis and Szepesvári, 2006], the problem of move selection at a node in the exploration of a game-tree will be illustrated by the metaphor of the multi-armed bandit problem. The resolution of this problem by using SDS constitutes the starting point of the algorithm proposed in this thesis.

Formulation of the problem

"A bandit problem with K arms is defined by the sequence of random payoffs $X_{it}, i = 1, ..., K, t \geq 1$, where each i is the index of a gambling machine. Successive plays of machine i yield the payoffs $X_{i1}, X_{i2}, ...$ For simplicity, the X_{it} are assumed to lie in the interval $[0, 1]$" [Kocsis and Szepesvári, 2006]. Informally, solving the "exploration-exploitation dilemma" for a gambler consists in finding rapidly the machine with the best expected value (exploration) in order to concentrate on that machine (exploitation).

The algorithm

The application of standard SDS (as described in chapter 3) to the multi-armed bandit problem is very straightforward. The different phases of the algorithm are presented in the following:

Initialisation Agents' hypotheses are initialised by choosing an arm of the bandit randomly (value between 1 and K).

Test Agents' hypotheses are tested by pulling the corresponding arm (if it is a win, the agent becomes active, otherwise it stays inactive)

Diffusion Every inactive agent selects at random another agent to communicate with. If the agent selected is active, the first agent copies his hypothesis and becomes active, otherwise it selects a new hypothesis at random (passive recruitment adopted).

Halt In the context of game playing, the bandit represents a node in the game-tree and the simulation gives information to the player on which move to choose. Several strategies can be adopted for the halting criteria. The simpler one is to keep the simulation going on until the player runs out of time and have to choose a move. Another strategy could be to evaluate the degree of confidence for the different moves and to halt the simulation and play when a threshold is reached.

Results

The application of SDS as described above on a 5-armed bandit (figure 4.1), with 200 agents and for 50 iterations, gives the results shown in figure 4.2.

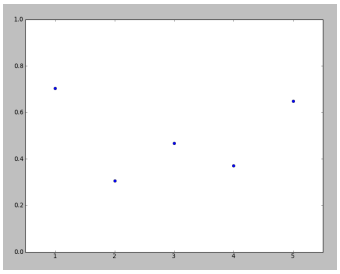


Figure 4.1: Expected value of each arm of the bandit

Comments

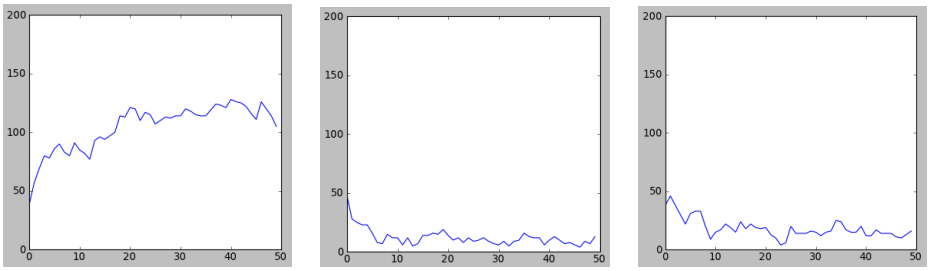
Apart from any consideration on the optimality of the algorithm for the moment (concerning the number of agents used for example), SDS appear to "solve" (in the informal way described earlier) the exploration-exploitation dilemma. In about 10 iterations, the distribution of the agents on the different arms roughly corresponds to their expected values.

4.2.2 Resolution of a simple game-tree

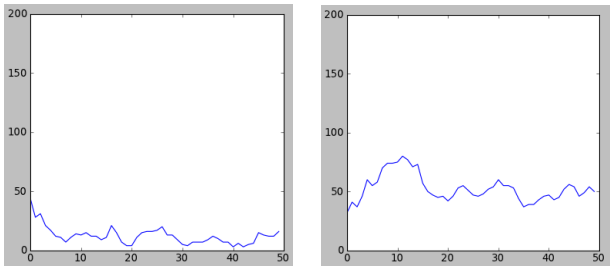
Now that it has been shown that SDS is able to solve the multi-armed bandit problem, the idea can be pushed further. The founding idea of Monte-Carlo tree search is that every node of the game tree can be treated as a multi-armed bandit.

Formulation of the problem

The simple game-tree presented in figure 4.3 constitutes an interesting study case. In particular, it shows why Monte-Carlo methods need to explore game-trees further than



(a) Arm 1 (b) Arm 2 (c) Arm 3



(d) Arm 4 (e) Arm 5

Figure 4.2: Evolution of the distribution of the agents among the different arms of the Bandit

one move. In the situation of Max, a Monte-Carlo method restricted to the next coming move would tell him to play move **b**. Indeed, he has 67% chances to win if he plays the move **b** against 33% chances if he plays move **a**, *if the rest of the game is played randomly*. However, *in the case of an optimally playing opponent*, move **b** would lead to a defeat of Max (Min playing his move **c**), whereas the victory is assured after move **a**. A good algorithm for game-tree exploration is expected to detect this kind of situation and play the correct move.

In the case of Monte-Carlo Tree Search methods, the ability to solve such situations comes from the combination of the *Selection* phase and the *Backpropagation* phase. In the Backpropagation phase, the outcome of the last simulated game is used to update the statistics of the crossed nodes. But since at each node, the selection is biased toward the current best move for the corresponding player, even the statistics of the first node tend to reflect the best move to play in a Minimax sense (for example in the case of the studied game-tree, the evaluations would select Max's move b at first, but then the selection of Min's move c as a second move would decrease the number of good evaluations for b as a first move by backpropagation).

The algorithm

A general idea to perform in a similar way with SDS is to create hypotheses by communication between agents from populations of different layers in the tree. The behaviour of the agents in the populations of upper layers is expected to influence the behaviour of

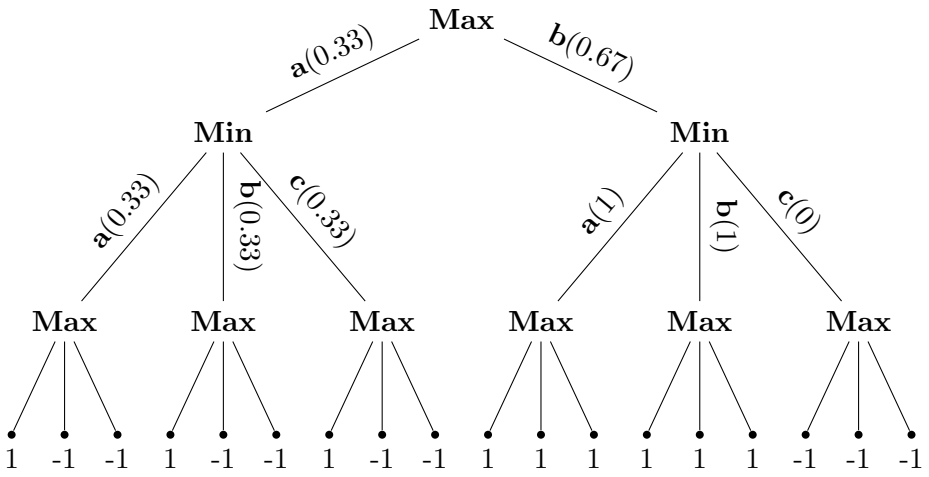


Figure 4.3: Studied Game-Tree.
The weights on the branches indicates the probability to win for Max, if he plays the corresponding move and if the rest of the game is played randomly.

the agents in the populations in lower layers in a way comparable to coupled SDS. The proposed algorithm keep the standard structure, but the different phases are modified.

Initialisation During the initialisation phase, a population of agents is generated for each node of the game-tree up to a fixed depth D. For each population, agents' hypotheses are initialised to a possible move of the corresponding node.

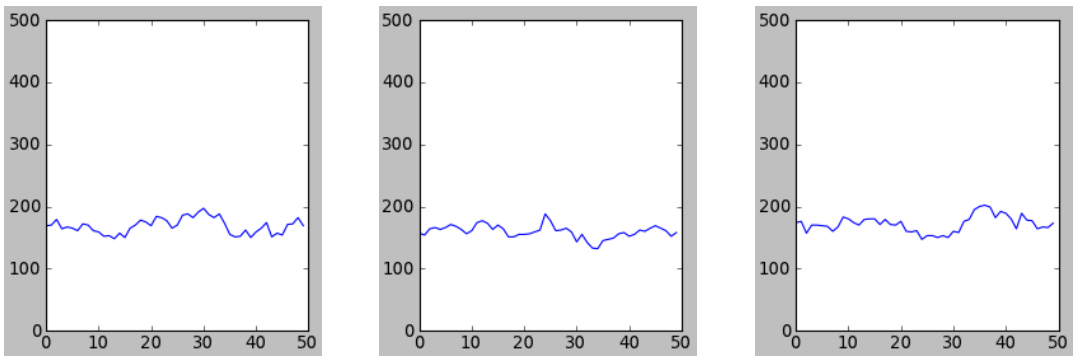
Test As in coupled SDS, during the test phase a complete hypothesis need to be formed. This is done in a rollout-based manner by selecting an agent from the population corresponding to the root of the game-tree, then an agent in the population pointed by the previous agent, etc, until depth D is reached. Then a random game is simulated (starting after the succession of moves corresponding to the hypothesis formed). If it is a win for Max, the agents in populations corresponding to Max's nodes become active and the agents in population corresponding to Min's nodes become inactive (if it is a loss, it is the contrary).

Diffusion Again as in coupled SDS, during the diffusion phase each population of agents acts independently, i.e. each population undertake a diffusion phase in the sense of Standard SDS without communicating between each other. The different existing diffusion methods can be used.

Halt As mentioned in the section on the multi-armed bandit, the halting criteria can either be the time limit accorded to the player, or depend on a confidence threshold.

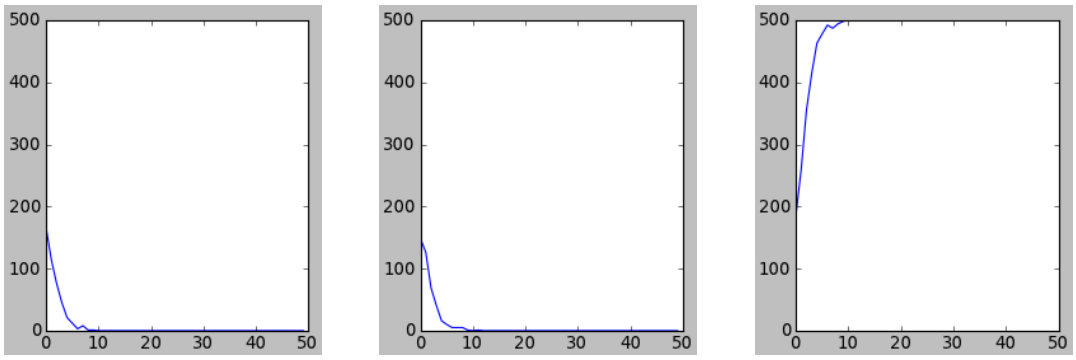
Results

This algorithm has been applied on the game-tree shown in figure 4.3 to a depth of 2 (3 populations of agents were generated: one for the root node (Max) and one for each of the nodes corresponding to Min's turn). The results are shown in figures 4.4, 4.5 and 4.6 (for 50 iterations, with 500 agents defined per population).



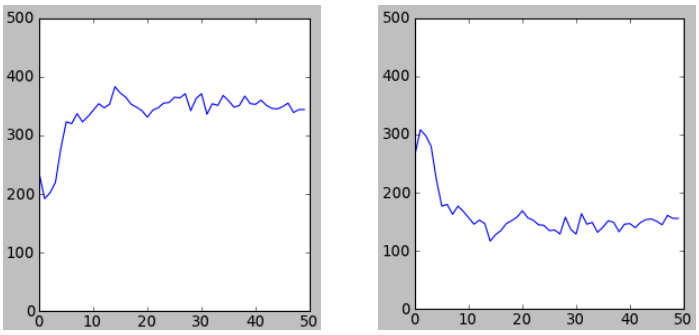
(a) move a (b) move b (c) move c

Figure 4.4: Evolution of the distribution of the agents in Min's node population corresponding to Max's move a



(a) move a (b) move b (c) move c

Figure 4.5: Evolution of the distribution of the agents in Min's node population corresponding to Max's move b



(a) move a (b) move b

Figure 4.6: Evolution of the distribution of the agents in the root node population

Comments

Not surprisingly, the agents in Min's nodes populations show the expected behaviour. After about 10 iterations, the distribution of the agents in the population corresponding to Max's move a is still roughly equilibrated, while all the agents support Min's move c in the population corresponding to Max's move b. This is not surprising because agents in these populations are simply dealing with the problem of move selection as a multi-armed bandit problem as studied in the previous section. More interestingly, the population corresponding to the root node also shows the expected behaviour. While agents tend to support Max's move b in the first few iterations, the trend reverses quickly and again, after about 10 iterations, the majority of agents stabilises to support his move a. This behaviour corresponds to what was expected.

4.3 Second step: Reallocation Policies

One drawback of the method just presented is that the number of agents in each population is fixed, and many agents are useless because they are not contacted to form an hypothesis. In the coming two sections, possible reallocation policies are presented in order to optimise the use of the agents.

4.3.1 Reallocation within layers

The first possible reallocation policy relies on the fact that not more hypotheses than the number of agents in the population at the root node of the tree can be formulated at each iteration of the algorithm. Indeed, each agent of the root node is used to form one and only one hypothesis. Hence, in order to use all agents, the total number of agents in each layer of the tree (all the nodes that have the same depth) should be equal to the number of agents in the root node population. Moreover, the allocation of agents in populations of a given layer should correspond to the hypotheses of the agents in the layer underneath. This can be done by slightly modifying the different phases of the algorithm, and by adding a "contacted" variable to the agents.

Initialisation During the initialisation phase, a given number of agents is distributing among the different nodes of each layer in a random way.

Test The test phase is performed in the same way as previously, but the "contacted" variable of the agents forming an hypothesis is set to true while it stays equal to false for the other agents.

Diffusion The diffusion phase is now divided in two sub-phases:

1. The agents that were not contacted to form an hypothesis are reallocated to a new population on the same layer of the game-tree. This could be done in different ways, but the method consisting in reallocating them in a rollout-based manner similar to the formation of an hypothesis lead to the expected behaviour (uncontacted agents pick an agent at random in the root node, then in the node pointed by the first agent, etc, until they reach an agent from their layer—once it is done, they join the population of this agent).
2. The usual diffusion phase is performed in every population by applying the passive recruitment procedure.

This reallocation policy has been tested on the simple game tree shown in figure 4.3. Figure 4.4 shows the evolution of the number of agents in Min's node population corresponding to Max's move a. It roughly follows the number of agents supporting move a in the root node population and also shows the switch in the first few iterations between supporting Max's moves b and a.

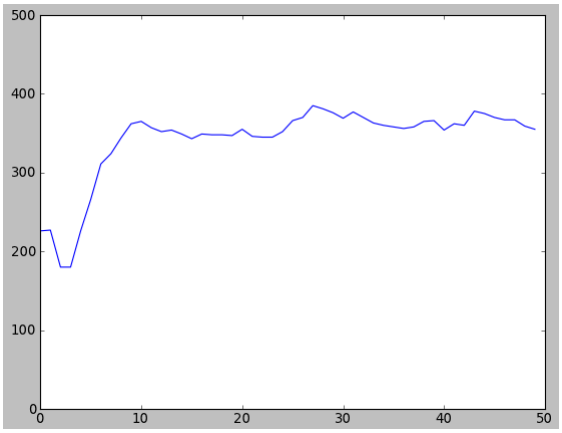


Figure 4.7: Evolution of the number of agents in Mins node population corresponding to Maxs move a when reallocation of the agents within the layers is used

4.3.2 Reallocation within the entire tree

The second possible reallocation policy relies on the observation that as long as communication of hypotheses is limited to agents of the same population, agents can in fact be reallocated anywhere in the tree. Hence, contrary to the first reallocation policy that fix the number of agents on each layer and that fix the number of layers studied, a more complex reallocation policy might lead to a more elegant and natural balance of the resources within the entire game-tree. This can be done by modifying the phases of the algorithm as follow:

Initialisation During the initialisation phase, all the agents are allocated to the root node population.

Test The test phase is performed in the same way as previously, and again a "contacted" variable indicate if an agent has been contacted to form an hypothesis.

Diffusion The diffusion phase is now divided in three sub-phases:

1. The agents that were not contacted to form an hypothesis are "backscattered". This can be realised either by putting them in the parent node population, or directly in the root node population.
2. Some inactive agents are "scattered" toward the children nodes by using an active recruitment procedure. Every active agent selects another agent at random; if the selected agent is inactive, it is sent toward the child node pointed by the active agent's hypothesis.
3. Finally the usual diffusion phase is performed in every population by applying the passive recruitment procedure.

This algorithm has been tested in a Hex playing program (chapter 5) and lead to a spontaneous distribution of the agents on the game-tree. However, the distribution is strongly dependent on the procedure chosen. In particular, the active recruitment phase scattering the agents seems to be too strong in comparison to the backscattering phase and do not allow enough agents to stay at the root node. In the current state of the research, this effect has simply been nuanced by introducing a random condition on the displacement of an agent by another active agent. However, this solution is not completely satisfying because it introduces an arbitrary factor (the random condition) which is in contradiction with the wish to build a genuinely self-organising system.

4.4 Pros and Cons

In this chapter, it is maintained that standard SDS in itself is unable to perform satisfying game-tree exploration. A method sharing some common points with Coupled SDS and inspired from the existing Monte-Carlo Tree Search techniques is proposed. Its main

characteristics are the use of a population of agents per node in the studied part of the game-tree, the formation of hypotheses in a rollout-based manner by the collaboration of an agent per population crossed and the dynamical reallocation of the agent within the different populations. The advantages and disadvantages of this algorithm are varied.

Pros First, the method presented offers a natural extension of the idea of Coupled SDS. Thus it is concerned with the comments made in section 3.3.3. In particular, it is an elegant illustration of dynamical coupling between several populations of agents, and it might be seen as constituting a broadening of the communication metaphor at the basis of SDS. Also, it shows how SDS's problem-solving abilities can be improved by the collaboration of several populations of agents, and how complex hypotheses can be formed from several simpler ones. Second, the method is isomorph to Monte-Carlo Tree Search—the most efficient method in game AI for games with large branching factors. Finally, due to the parallel nature of SDS, the method might be implementable in a connectionist architecture.

Cons The application of SDS made in the method presented here appears to be more horizontal (within the nodes of the tree) than vertical (within the tree itself) as it was envisaged in the first intuitions and no used has been made of the idea to form composite hypotheses. This results in the use of many populations (one per node), and in the fact that SDS is used to balance exploration and exploitation in rather small spaces (the available moves at each node) instead of the entire search-space. Also, the use of a population of agents to handle the problem of move selection at a node is much more expensive in resources than other available techniques simply storing the win/loose ratios. Concerning the possible implementation on a parallel device, the multiplication of the number of populations makes it more complicated than it is for standard SDS.

Having these different remarks in mind, the next chapter explains how the method has been applied to practical game-playing on Hex.

Chapter 5

Example Application to the Game of Hex

In this chapter, the method proposed in chapter 4 to solve the theoretical problem of adversarial search in game-tree exploration is applied to practical computer game-playing. The game of Hex has been chosen for three main reasons:

- First, it is relatively simple and well-suited to perform rapid random game evaluations: the only action available to each player at each turn is to put one stone of his colour on the board.
- Second, the size of the board is changeable without modifying the nature of the game. This allows the algorithm to be tested on different sizes of game-trees.
- Third, it is mathematically elegant and has interesting properties in game theory.

5.1 Introduction to Hex

Hex is a combinatorial game (see chapter 2) that belongs to the family of connection games. It is an example of a game that has been solved in a non-constructive way: it is proved that the first player has a winning strategy, but the strategy in question is not known. The proof of this result is given in the following sections, after the rules of the games have been presented.

5.1.1 Rules and History

The game of Hex was first invented in 1942 by the Danish scientist, artist and poet Piet Hein. It was then independently reinvented in 1947 by the American mathematician John Nash, while still a student at Princeton University. It was first known under the name *Polygon* in Denmark and was called after its creator *John* or *Nash* at Princeton University, before Parker Brothers marketed a version under the name *Hex* in 1952.

The game is played on a rhombic board covered with hexagonal cells. Each of the two players is associated with a colour (blue and red in the following) and two opposite sides of the board (for example top-right and bottom-left for blue and top-left and bottom-right for red). The rules are extremely simple: the two players alternatively place a stone of their colour on a single cell within the entire board and try to form a connected path between their two sides. The game ends when one of the two players managed to build such a connection. The usual size of the board is 11×11, but due to the relationship that Hex maintains with Go, the sizes 13×13 and 19×19 are also common. According to Sylvia Nasar's biography of John Nash *A Beautiful Mind*, this one recommended 14×14 as the optimal size. Figure 5.1 shows a typical finished game: Red wins because he managed to connect his two sides of the board.

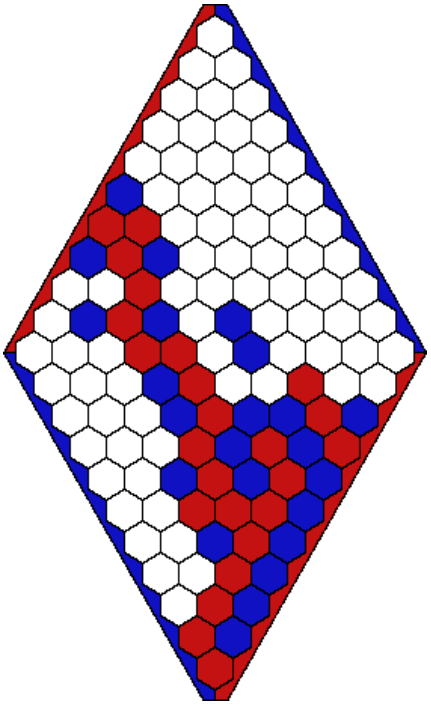


Figure 5.1: A typical finished game at Hex (*From Wikimedia Commons*)

5.1.2 Game Theory

The proof of the existence of a winning strategy for the first player relies on two central points: the fact that there can be no draw in Hex, and the strategy stealing argument. A relatively simple proof of the first point is given in a paper from David Gale: "The game of Hex and the Brouwer fixed-point theorem" and is outlined in the next section. Interestingly, the same paper establishes an equivalence between this proof and the Brouwer fixed-point theorem, an important theorem in topology that states that for any continuous function f with certain properties there is a point x_0 such that $f(x_0) = x_0$.

One and only one winner

If the blue player is called x and the red player is called o, and if the two sides of the board corresponding to the blue player are called X and X' and the two sides corresponding to the red player are called O and O', Gale states what he calls the Hex theorem as follow:

Hex theorem: If every tile of the Hex board is marked either x or o, then there is either an x-path connecting regions X and X' or an o-path connecting regions 0 and 0', but not both. [Gale, 1979]

In the original paper, Gale gives a very intuitive illustration of the theorem: " Imagine, for example, that the X-regions are portions of opposite banks of the river "0" (...) and that the x-player is trying to build a dam by putting down stones. It is quite clear that he will have succeeded in damming the river only if he has placed his stones in a way which enables him to walk on them from one bank to the other." In other words, the only way one has to prevent his opponent from winning is by winning himself and thus there is always a winner. Then Gale continues his analogy: "if the x-player succeeds in constructing a causeway from X to X', he will in the process have dammed the river and prevented any flow from 0 to 0'." In other words, if one of the players wins he prevents his opponent from winning at the same time and there can be only one winner. Although the theorem is intuitive, the proofs of the two results (the existence and the uniqueness of a winner) are rather delicate. The uniqueness directly follows from a fundamental result of topology called the Jordan Curve Theorem. This theorem asserts that every non-self-intersecting continuous loop divides the plane in an "interior" region and an "exterior" region so that any continuous path connecting a point of one region to a point of the other intersects with that loop somewhere. Although this theorem is also very intuitive, it is difficult to establish formally.

A constructive proof of the existence of a winner is given in [Gale, 1979]. It is outlined in the first part of the paper as follow (figure 5.2 is the corresponding figure on which colours have been added for clarity):

> "We consider the edge graph Γ of the Hex board to which additional edges ending in vertices u, u', v, v' have been added to separate the four boundary regions, as shown in the figure. We now present an algorithm for finding a winning set on the completely marked board. We shall make a tour along Γ, starting from the vertex u and following the simple rule of always proceeding along an edge which is the common boundary of an X-face and an 0-face. Note that the edge from u has this property since it separates regions X and 0. The key observation is that this touring rule determines a unique path; for suppose one has proceeded along some edge e and arrives at a vertex w. Two of the three faces incident to w are those of which e is the common boundary, hence one is an X-face, the other an 0-face. The third face incident to w may be either an X-face or an 0-face, but in either case there is exactly one edge e' which satisfies the touring rule."

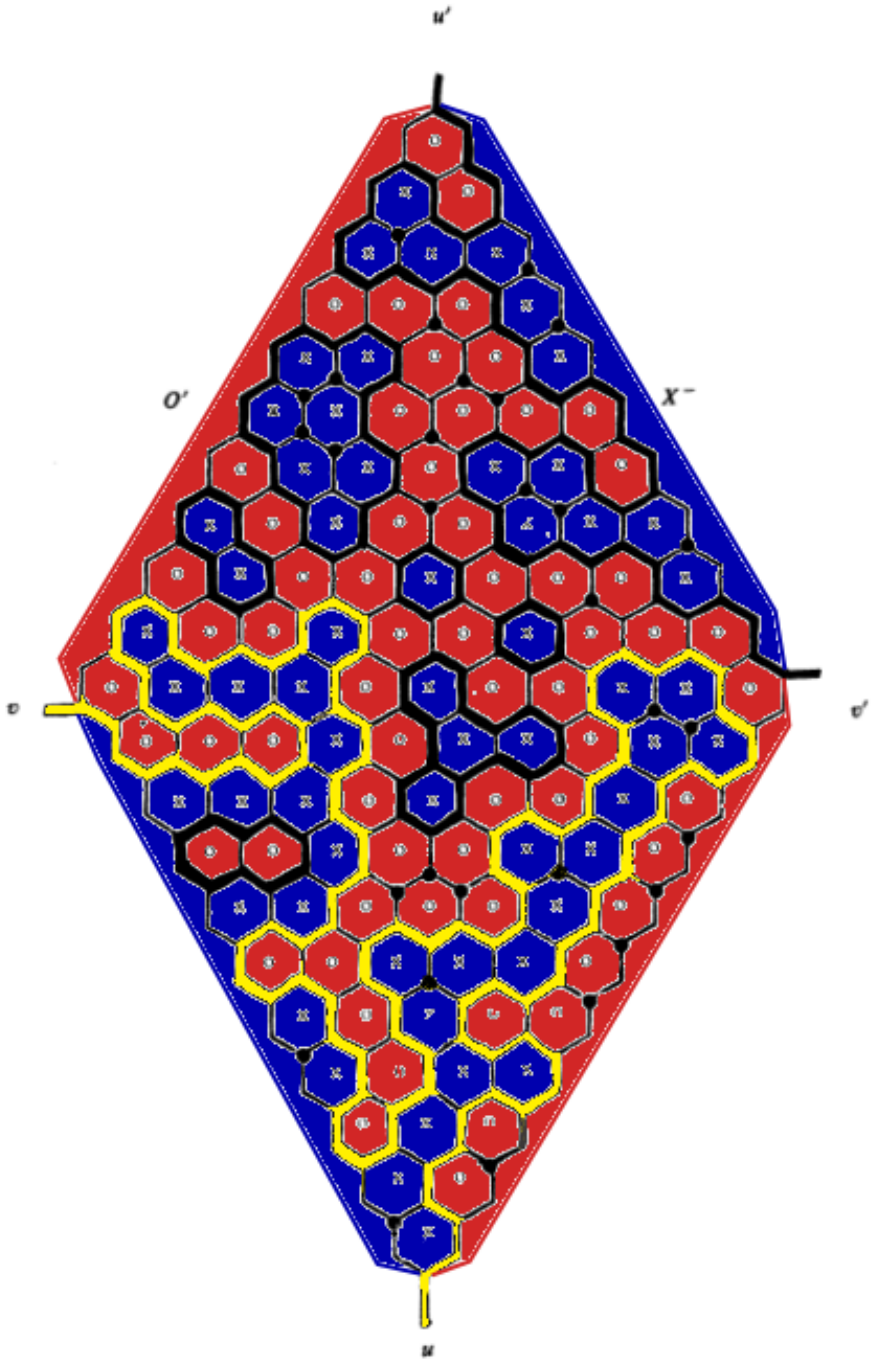


Figure 5.2: Illustration of the tour followed by Gale's algorithm on a full Hex board (starting at u and finishing at v in yellow). The red cells on its right form a winning path for O. (*From [Gale, 1979], colours added.*)

The tour constituted by this algorithm is highlighted in yellow on figure 5.2 (it starts at vertex u on the bottom and ends at vertex v on the left). Two characteristics of this tour show that there necessarily is a winner:

1. It will never revisit any vertex. The reason of this is that by construction, the degree of every vertices of the tour is at most two. But since the degree of the first vertex of the tour is one (vertex u), the tour must constitute a simple path and end on a vertex of degree one. The only possibilities are u', v or v'.

2. It cannot end on vertex u' (top one). The reason of this is that the tour starting at u always keep blue cells on the left and red cells on the right while the edge ending at u has a red cell on its left and a blue cell on its right[1]. Hence the tour can only end on v or v'.

To conclude, one only need to notice that if the tour ends on v the red cells on its right form a winning path for O and if it ends on v' the blue cells on its left form a winning path for X. Hence, there is always a winner at Hex.

The Strategy Stealing Argument

The absence of draw at Hex has an interesting consequence if one reminds Zermelo's theorem: according to [Hart, 1992], Zermelo's theorem states that "in Chess, either White can force a win, or Black can force a win, or both sides can force a draw." Now it can be maintained that in Hex, either Blue can force a win or Red can force a win. In fact, an ingenious reductio ad absurdum from John Nash called the strategy stealing argument proves that this is the first player who has a winning strategy. The argument goes as follow: let one suppose that there exists a winning strategy for the second player. In this case, the first player can steal this strategy to build his own winning strategy in the following way. First he places one of his stones anywhere on the board and let the second player play as if he was the first one. Then he follows the second player's winning strategy until either the game finishes, or the winning strategy tells him to play the move he played first. In the second case, he just plays anywhere and starts following the winning strategy again the next turn. In this situation both players have a winning strategy which is contradictory and the initial hypothesis that the second player has a winning strategy is false. It is important to note here that the stealing strategy argument only holds because it is never a disadvantage to play a move at Hex. However this is not always the case: in Chess there are situations called *Zugzwang* in which every move leads to a worse and often lost position (they happen most of the time in late endgames).

Of course the strategy stealing argument is non-constructive and the winning strategy for the first player is not known for boards bigger than 9×9 cells. Yet in practical play it appears that playing first do constitutes a great advantage. To compensate this bias, the

[1]The rigorous proof is not based on this left-right consideration: "this would involve getting into the quite complex notion of orientation, which is not needed for our proof." [Gale, 1979]

swap rule allows the second player to choose between either playing normally, or taking the first player's position after his first move. This re-equilibrates the game because in this case the first player should play neither the strongest moves (such as the centre of the board) because the second player would switch its position with him, nor the worst moves (such as the two cells in the acute angles of the rhombus) because the second player would leave them to him. In the presence of the swap rule this is the second player who has a winning strategy since he can choose between taking the first player's move if it is a winning one or leaving it if it is a loosing one (although this information is not known in practical play).

5.2 Application of Stochastic Diffusion Search

The application of the method presented in chapter 4 to Hex was the opportunity to develop an environment to support the game. This environment, as well as the different AI produced have been written in C++.

5.2.1 Implementation of a Hex playing environment

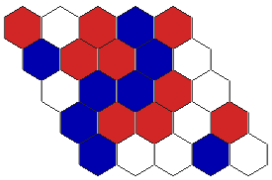
The first step in the implementation of a Hex playing environment was to choose a representation of the game well-suited to store the data. It was decided that a $n \times n$ Hex board would be implemented as a $n \times n$ matrix with the following convention:

- An empty cell is represented by 0 in the matrix.
- A cell occupied by the vertical player (Blue) is represented by 1.
- A cell occupied by horizontal player (Red) is represented by -1.
- The upper left corner is an acute angle.

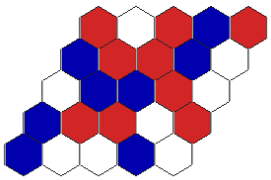
The necessity to fix the nature of the upper left corner is illustrated in figure 5.3 where the same matrix representation leads to two different outcomes of the game according to the convention adopted.

$$\begin{pmatrix} -1 & 0 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 & 0 \\ 0 & 1 & 1 & -1 & 0 \\ 1 & -1 & -1 & 0 & -1 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(a) Matrix representation of a Hex board



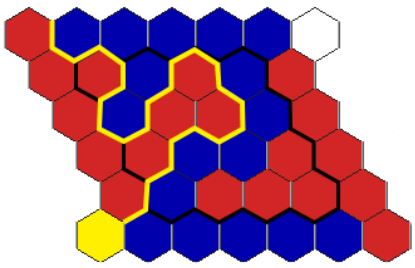
(b) Corresponding board with acute upper left angle: Blue wins



(c) Corresponding board with obtuse upper left angle: Red wins

Figure 5.3: Importance of the convention fixing the nature of the upper left corner in the matrix representation.

Once the internal representation of the board was set, an important step in the development of a Hex playing environment was the implementation of a function to determine the winner from a full board. This was realised by applying Gale's algorithm presented in the previous section: a tour is initialised at the left upper corner of the matrix and is followed until it comes out of it. If it comes out from the bottom-left corner, blue is the winner; if it comes out from the top-right corner, red is the winner. The details of the implementation are given in table 5.1 and illustrated in figure 5.4.



(a) Tour followed by Gale's algorithm

$$\begin{pmatrix} -1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & -1 & -1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 & 1 & -1 \\ 1 & 1 & 1 & 1 & 1 & -1 \end{pmatrix}$$

(b) Its equivalent in the matrix representation

Figure 5.4: Blue is the winner because the tour followed by Gale's algorithm comes out of the matrix from the bottom-left corner (it comes out from the upper-right corner for Red).

In order to detect a winning situation in the course of the game and not only on a full board the following trick has been adopted: between each turn, two copies of the board are created and filled with red stones for the first one and blue stones for the second one. If blue is the winner in the copy filled with red stones then he has already won in the non-full board, and if red is the winner in the copy filled with blue stones then he has already won in the non-full board.

Initialise
// Extend the board to include the boundary regions
newBoard(1:n+1,1:n+1) = board
newBoard(0,1:n+2) = ones(1,n+1) // Upper blue boundary
newBoard(n+1,0:n+1) = ones(1,n+1) // Lower blue boundary
newBoard(0:n+1,0) = -ones(n+1,1) // Left red boundary
newBoard(1:n+2,n+1) = -ones(n+1,1) // Right red boundary
// Initialise the tour
Red = (0,0) ; Blue = (0,1) ; Next = (1,1)

Repeat
// Determine which square of the matrix is the next one to be studied
if(Red[0] == Blue[0] & Red[1] == Blue[1]-1)
Next = (Red[0]+1,Red[1])
else if(Red[0] == Blue[0]-1 & Red[1] == Blue[1])
Next = (Red[0]+1,Red[1]-1)
else if(Red[0] == Blue[0]-1 & Red[1] == Blue[1]+1)
Next = (Red[0],Red[1]-1)
else if(Red[0] == Blue[0] & Red[1] == Blue[1]+1)
Next = (Red[0]-1,Red[1])
else if(Red[0] == Blue[0]+1 & Red[1] == Blue[1])
Next = (Red[0]-1,Red[1]+1)
else if(Red[0] == Blue[0]+1 & Red[1] == Blue[1]-1)
Next = (Red[0],Red[1]+1)
// If the next square to be studied is (0,n+2) then red wins
if (Next[0] == 0 & Next[1] == n+2)
Red Wins!
// If the next square to be studied is (n+1,-1) then blue wins
if (Next[0] == n+1 & Next[1] == -1)
Blue Wins!
// According to the colour of the next square, either red or blue takes its coordinates
if (newBoard[Next[0]][Next[1]] == -1) // colour(Next) = red
Red = Next
else
Blue = Next
until (Red wins or Blue wins)

Table 5.1: Gale's algorithm to determine the winner of a full Hex board (of size n) applied to the matrix representation.

Finally the last touch necessary to allow the developed environment to support game play between two humans was to create a user interface. A very rudimentary one has been implemented in the C++ console application. It is presented in figure 5.5 (empty cells are represented as dots ".", cells corresponding to the vertical player are represented as "X" and cells corresponding to the horizontal player are represented as "O").

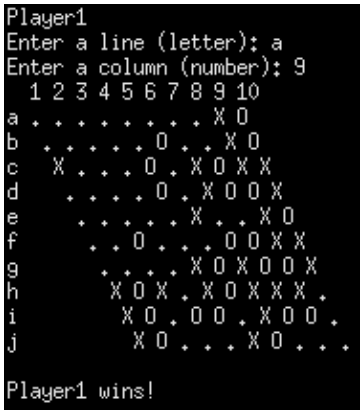


Figure 5.5: User Interface showing the end of a game between two human players

5.2.2 Implementation of 3 levels of AI

Once the environment implemented, it has become possible to start developing some rudiments of AI. The starting point was the idea at the basis of Monte-Carlo methods: the simulation of random games a great number of times.

Vanilla Monte-Carlo

The three algorithms presented in the following evaluate moves in the same way, i.e. according to the expected outcome they offer. In this category, the Vanilla Monte-Carlo algorithm is elementary: at each turn it performs a fixed number of random game simulations for every available move and play the one that gets the best results. Since the function to determine the winner from a full board has already been implemented, the only missing element to implement this algorithm is a function to simulate a random game.

The suitability of Hex to perform random game simulations was one of the main reasons to select this game as a study case. Indeed a random game just consists in alternatively placing red and blue stones on the board until it is full. Although this is relatively simple, some care has to be given to fill the board with a uniform distribution or the evaluation of the moves could be biased. The solution adopted that fulfils this requirement was to select a random number between one and the number of remaining cells and to play the cell whose index is this number when counting the empty cells in the matrix (counting in a left-to-right and top-to-bottom progression).

SDS applied in the root node

The first improvement that can be given to the vanilla Monte-Carlo algorithm is to sample the evaluation of the cells by applying the multi-armed bandit analogy: the moves that tend to give good results in at the beginning of the evaluation should receive more attention that the moves that appear to be bad. However, it is important to notice here that the improvement only concerns the calculating time; the value attributed to each move is theoretically the same as in the vanilla version. Indeed the value of a move is still assimilated to the probability that it leads to a win given random play, and this probability is not changed by the way evaluation is balanced between the different moves.

This has been done by applying SDS in the root node of the game tree as it was done in section 4.2.1. First a population of agents with hypotheses about the best move to play is initialised. Second the hypotheses are tested by performing a random game simulation: if the outcome is a win the agent becomes active, otherwise stay inactive. Third every inactive agent selects at random another agent for communication. If the selected agent is active the first agent copy its hypothesis, if the selected agent is inactive the first agent chooses a new hypothesis at random (passive recruitment strategy).

SDS applied in the entire tree

Then the crucial step was the implementation of the method presented in chapter 4. The main improvement that this method provides in comparison to the two previously discussed versions is a deeper exploration of the game tree, which is expected to result in better tactical play.

In addition to testing the method on Hex, it has been decided to experiment the reallocation policy within the entire tree as described in section 4.3.2. In order to do so, a powerful data structure was necessary to allow efficient manipulation of the agents and the possibility to dynamically displace them among different populations. In fact, this requirement was one of the reason to choose the programming language C++. The solution adopted was to create two classes: a simple class Agent and a class Node inspired from the linked list data structure. The members of the class Agent are simply a variable indicating the agent's activity, a variable indicating if the agent has been contacted, and a variable indicating the agent's hypothesis. The members of the class Node are a vector of Agents, a pointer toward the parent Node, and a vector of pointers toward the children Nodes: this is the parent and children members that make the implementation of a dynamical tree structure possible. Two more members that appeared to be useful are a variable indicating which player corresponds to the Node, and a vector indicating the moves that have been performed between the current board position and the position corresponding to the Node. Once these Classes were created, the application of the method developed in Chapter 4 to Hex—including the reallocation policy within the entire tree—was rather straightforward.

5.2.3 Analysis of the play

This section discusses the level of play that the programs presented in the previous section show. In particular, a comparison is made between the vanilla Monte-Carlo program and the program resulting from the application of the method introduced in chapter 4. After trying different board sizes, 7×7 appeared to be a good compromise between small boards on which the advantage of the first player is too strong, and big boards that require important computational resources to be played.

Vanilla Monte-Carlo

Not surprisingly, the vanilla Monte-Carlo program shows the characteristics described in section 2.3.1 on the Expected Outcome model. It has a relatively good strategical sense and always performs moves that increases its overall chance to win, but it is poor tactically. A good illustration of this remark are the situations called bridges in Hex where a player cannot stop the other player from connecting two groups of stones in one move because there are two ways to do it (see figure 5.6) . When a bridge is formed for one player, the best tactic for the other player is to play something else than trying to close the bridge. However the vanilla Monte-Carlo program is unable to do so because this requires to anticipate the next move of the opponent (see figure 5.7).

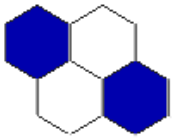
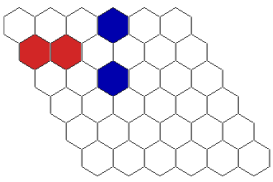


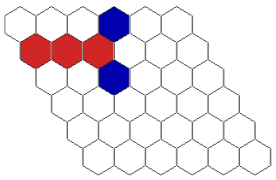
Figure 5.6: A bridge for Blue: Red cannot stop Blue from connecting the two blue cells on the next move.



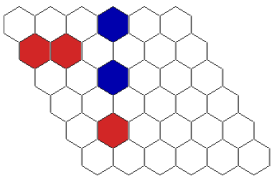
(a) An hypothetical game situation. It is Red to move.

SDS applied in the root node

As it is explained in the previous section, the program using SDS in the root node has the same level of play as the program using vanilla Monte-Carlo. It just uses SDS as a



(b) The move that the vanilla Monte-Carlo program would perform: it increases its chances to win if the rest of the game is played randomly.



(c) Example of a better tactical move.

Figure 5.7: Illustration of the weakness in tactic peculiar to the vanilla Monte-Carlo program.

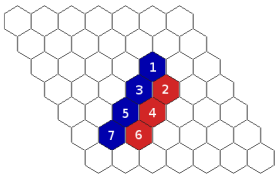
more efficient method to allocate the computing resources. After several trials, it was considered that 50 000 agents and 50 iterations of the algorithm leaded to satisfying results on the 7×7 board. However this is just an empirical result and a theoretical analysis of the problem could lead to more optimized values.

SDS applied in the entire tree

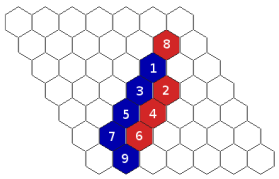
The first results of the program based on SDS applied in the entire tree were very disappointing. With 50 000 agents and for 50 iterations of the algorithm, the level of play observed was worse than the level of play of the two previous programs and a good proportion of the played moves seemed to be just random. After displaying the distribution of the agents among the different possible moves in the root node population it appeared that only a small part of the total number of agents were staying on this node. In fact, this problem revealed that the distribution of the agents in the entire tree is extremely sensitive to the reallocation policy adopted, and that the "scattering" force in the policy chosen was too strong in comparison to the "backscattering" force. Instead of modifying the entire policy, it has been decided to introduce a random factor to reduce the strength of the scattering force. In the active recruitment sub-phase of the diffusion phase, an active agent now send a passive agent in the population pointed by its hypothesis only with a chance of 1/5, instead of sending it systematically. This slight modification re-equilibrated the distribution of the agents in the game-tree and restored the level of play of the program. However the superiority of the new program over the vanilla Monte-Carlo version was not obvious.

In order to determine the difference of level between the program applying SDS only in the root node and the program applying SDS in the entire game-tree, 200 games have been performed between the two versions (the first player is alternated between each game). It was measured that the second version wins about 63% of the time (127 victories). After analysis of the games, it appeared that the same sketch is repeated most of the time. In the fist period of the game, the two programs usually put alternatively their

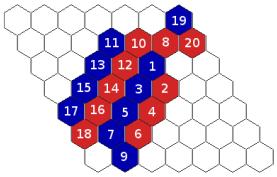
stones in a way that consists in trying to bypass each other as illustrated in figure 5.8a. Due to a one stone advantage, this scenario usually lead to a win for the first player. However there are some cases in which the program applying SDS in the entire tree manage to reverse the situation. In the figure 5.8a again, a vanilla Monte-Carlo evaluation of the board would urge Red to play one of the two cells in the lower left corner to try to connect the left boundary. Indeed this connection would be a great advantage since the connection with the right boundary is almost done. However if the analysis is pushed only one move further, it is clear that Blue will prevent this connection by playing the other cell (blue formed a bridge with the lower boundary). The program applying SDS in the entire tree is able to evaluate correctly this situation and plays move 8 (Red) as shown in figure 5.8b. However the program applying SDS only in the root node considers the situation as a threat and plays move 9 (Blue). In the end, move 8 gave a one stone advantage to the program applying SDS in the entire tree and lead to a win for him (figure 5.8c).



(a) First step of a typical game: the two programs try to bypass each other (Blue was the first player).



(b) Turning point: Red plays a tactical move and Blue makes a tactical mistake.

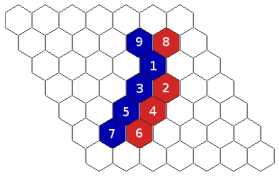


(c) Finished game: Red wins.

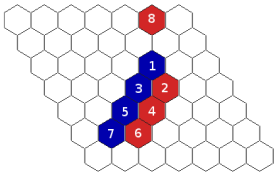
Figure 5.8: Typical game in which the superiority of the program applying SDS in the entire tree is expressed.

In the end, if the the application of SDS in the entire game-tree lead to a slight improvement of the level of play, it is still at a beginner level. In particular it is able to perform tactical moves of first order (as explained in figure 5.8), but no tactical moves of higher order have been observed with the current version of the program. For example in the situation described in figure 5.8b, Red manage to take the advantage because Blue plays in the lower left corner. However Red's move could be easily countered by playing move 9 as shown in figure 5.9a. A tactical move of second order for Red would consists in playing move 8 of figure 5.9b (Blue's counter-attack is countered). More work still need to be performed in order to determine if high order tactic is accessible to the method

developed in this thesis. There is no a priori reason to think that it is not, and a major improvement might come from the replacement of purely random game simulations by pseudo-random game simulations as it is suggested in [Chaslot et al., 2008]: "If all legal actions are selected with equal probability, then the strategy played is often weak, and the level of the Monte-Carlo program is suboptimal. We can use heuristic knowledge to give larger weights to actions that look more promising".



(a) Move 8 is a tactical move of first order. It can be easily countered with move 9.



(b) Move 8 is a tactical move of second order.

Figure 5.9: Illustration of the inaptitude of the program to play tactical moves of second order.

Chapter 6

Conclusion

This thesis proposed to address the question as to whether the Stochastic Diffusion Search metaheuristic is able perform non-trivial game-playing. The fact that it can perform trivial game-playing was admitted and shown on Hex: it is possible to use standard SDS to select the next move with the best expected-outcome in a Monte-Carlo sense (in a more efficient way than by evaluating each move the same number of times). The question was rather concerning its ability to explore the game-tree, in order to produce good tactical play.

6.1 Summary

Chapter 2 presented the problem addressed: computer game-playing. It was first formulated an important result about the games concerned (finite two-person zero-sum games with perfect information)—their determinateness. Historically, Zermelo was the first to see this result in 1913 before von Neumann and Morgenstern re-established it in 1953 in a stronger sense with the introduction of the concept of backward induction. Von Neumann and Morgenstern's proof led to the development of the first main framework for computer game-playing that has been particularly successful in Chess; its core elements are the use of the minimax algorithm combined to an evaluation function for non-terminal states and the pruning of insignificant parts of the game-tree by using Alpha-Beta search and other heuristics. The second main framework for computer game-playing appeared recently—in the last decade—as a partial solution to two limits of the first approach: it is applicable on games with large branching factors, and it requires little domain-dependent knowledge. This new framework is called Monte-Carlo Tree Search and has been particularly successful in Go; it relies on the random simulation of a great number of games.

Chapter 3 presented the metaheuristic employed: Stochastic Diffusion Search. It was stated that there are two main interesting aspects in SDS: it constitutes at the same time a metaheuristic for search and optimisation and a model for neural activity. Concerning the metaheuristic, the general algorithm was presented and some variants were discussed.

The main results concerning the convergence properties were also given. Concerning the model for neural activity it was argued that, generally speaking, SDS fits well in the framework of modern theories of cognition. Then the neural network instantiation of SDS (Nester) was presented. Finally, Coupled SDS was argued to constitute a broadening of the communication metaphor at the basis of SDS, introducing a meta-level between the populations. This idea has been reused in the method proposed in chapter 4.

Chapter 4 constitutes the heart of the thesis and presents the proposed answer to the research question. It was argued that the initial idea to use composite-hypotheses to allow one population of agents to explore the entire game-tree was fallacious due to the lack of independence in the evaluation of the moves at difference stages of the game-tree. Instead, a method inspired both from Monte-Carlo Tree Search and Coupled SDS was presented and shown to solve the problem of game-tree exploration. It is based on two main ideas: first the use of one population per node studied in the game-tree, and second the use of a reallocation policy to optimise the use of the agents.

Chapter 5 is an application of the method presented in chapter 4 to the game of Hex. First the game was introduced and several notable mathematical properties were demonstrated. Then the implementation of a Hex playing environment including three levels of AI was described. The first level of AI is simply a vanilla application of Monte-Carlo ideas. The second level is the trivial application of SDS into the root node of the game-tree. The third level was the application of the method presented in chapter 4 (SDS in the entire tree). At the end, it was established that the method proposed in the thesis lead to a slight tactical advantage over the other methods. However few experimentations have been performed yet, and due to the lack of sophistication of the implementation (no domain knowledge has been integrated at all, the game simulations are purely random) the program is still at a beginner level.

6.2 Further Work

The method proposed in this thesis to solve the problem of game-tree exploration using SDS has been presented in general terms, but has not been studied in depth. Clearly, more work would be needed to determine its real value. There are also two extensions that might be interesting to consider: the implementation in a neural network architecture and the application to Reinforcement Learning.

The implementation in a neural network architecture would constitute an interesting theoretical result in itself as no purely connexionist model currently exist to perform game-playing. Due the inherently parallel nature of SDS it should be rather direct to do in practice, although the multiplication of the number of populations as well as the complexity of the communication procedure to form complete hypothesis make it more complicated than for standard SDS.

The application to reinforcement learning would also constitute a rather interesting extension as it concerns the ability of a system to learn by itself, a fundamental notion in the conception of autonomous agents. The belief that the method presented in this thesis is applicable to reinforcement learning comes from the observation that game-tree exploration and reinforcement learning are both essentially problems of backward induction. Moreover, the paper originally introducing UCT—the method at the origin of Monte-Carlo Tree Search—was not only targeting game-playing but the general "problem of finding a near optimal action in large state-space Markovian Decision Problems (MDPs) under the assumption a generative model of the MDP is available." [Kocsis and Szepesvári, 2006].

Bibliography

B. Abramson. Expected-outcome: A general model of static evaluation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 12(2):182–193, 1990.

V.V. Anshelevich. The Game of Hex: an automatic theorem proving approach to game programming. In *Proceedings of the National Conference on Aritificial Intelligence*, pages 189–194. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2000.

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

D. Barrios, L. Malumbres, and J. Rios. Convergence conditions of genetic algorithms. *International journal of computer mathematics*, 68(3-4):231–241, 1998.

P.D. Beattie and J.M. Bishop. Self-localisation in the senario autonomous wheelchair. *Journal of intelligent & robotic systems*, 22(3):255–267, 1998.

K. Binmore. *Fun and games, a text on game theory.* DC Heath and Company, 1992.

J. Bishop. Coupled stochastic diffusion processes. *Proc. SCARP, Reading, UK*, pages 185–187, 2003.

J.M. Bishop. Stochastic searching networks. In *Artificial Neural Networks, 1989., First IEE International Conference on (Conf. Publ. No. 313)*, pages 329–331. IET, 1989a.

J.M. Bishop. *Anarchic techniques for pattern classification.* PhD thesis, University of Reading, 1989b.

J.M. Bishop. Dancing with pixies: strong artificial intelligence and panpsychism. *Views into the Chinese room: New essays on Searle and artificial intelligence.*, pages 360–378, 2002.

J.M. Bishop. Stochastic diffusion search. *Scholarpedia*, 2(8):3101, 2007.

J.M. Bishop. A cognitive computation fallacy? cognition, computations and panpsychism. *Cognitive Computation*, 1(3):221–233, 2009.

J.M. Bishop and J.S. Nasuto. Second-order cybernetics and enactive perception. *Kybernetes*, 34(9/10):1309–1320, 2005.

J.M. Bishop and P. Torr. The stochastic search network. *Neural networks for vision, speech, and natural language*, 1:370–397, 1992.

B. Bouzy and B. Helmstetter. Monte Carlo go developments. *Advances in computer games*, 10:159–174, 2004.

J.L. Casti. Five golden rules: great theories of 20th-century mathematics and why they matter. *New York: Wiley-Interscience*, 1, 1996.

D.J. Chalmers. Minds, machines, and mathematics. *Psyche*, 2(9):117–18, 1995.

D.J. Chalmers. Does a rock implement every finite-state automaton? *Synthese*, 108(3): 309–333, 1996.

G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, pages 216–217, 2008.

K. De Meyer. *Foundations of stochastic diffusion search.* PhD thesis, University of Reading, 2004.

M. Dorigo. *Optimization, learning and natural algorithms.* PhD thesis, Politecnico di Milano, Italy, 1992.

H.L. Dreyfus. *What computers still can't do: a critique of artificial reason.* The MIT Press, 1992.

P. Ehrenfest and T. Ehrenfest. Über zwei bekannte Einwände gegen das Boltzmannsche H-Theorem. *Physikalische Zeitschrift*, 8(9):311–314, 1907.

D.B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence.* IEEE Press, third edition, 2006.

D. Gale. The game of hex and the brouwer fixed-point theorem. *The American Mathematical Monthly*, 86(10):818–827, 1979.

H.J. Grech-Cini and G.T. McKee. Locating the mouth region in images of human faces. In *Proceedings of SPIE*, volume 2059, page 458, 1993.

S. Hart. Games in extensive and strategic forms. *Handbook of Game Theory with Economic Applications*, 1:19–40, 1992.

J. Haugeland. Syntax, semantics, physics. *Views into the Chinese room: New essays on Searle and artificial intelligence.*, pages 379–392, 2002.

D.W. Hubbard. *How to measure anything: finding the value of intangibles in business.* Wiley, 2010.

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

S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo Planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

J.R. Lucas. Minds, machines and Gödel. *Philosophy*, 36(137):112–127, 1961.

D. L. Mcclain. Once Again, Machine Beats Human Champion at Chess - New York Times. Nytimes.com, 5 December 2006.

D. Myatt and J. Bishop. Data driven stochastic diffusion networks for robust high-dimensionality manifold estimation. *Proc. SCARP, Reading, UK*, 2003.

D.R. Myatt, J.M. Bishop, and S.J. Nasuto. Minimum stable convergence criteria for stochastic diffusion search. *Electronics Letters*, 40(2):112–113, 2004.

D.R. Myatt, S.J. Nasuto, and J.M. Bishop. Alternative recruitment strategies for Stochastic Diffusion Search. *Artificial Life X, Bloomington USA*, 2006.

S. Nasuto and J.M. Bishop. Convergence analysis of stochastic diffusion search. *Parallel Algorithms and Application*, 14(2):89–107, 1999.

S.J. Nasuto. *Resource Allocation Analysis of the Stochastic Diffusion Search*. PhD thesis, University of Reading, 1999.

SJ Nasuto and JM Bishop. Neural stochastic diffusion search network-a theoretical solution to the binding problem. In *Proc. ASSC2, Bremen*, volume 19, 1998.

SJ Nasuto, JM Bishop, and K. De Meyer. Communicating neurons: A connectionist spiking neuron implementation of stochastic diffusion search. *Neurocomputing*, 72(4): 704–712, 2009.

A. Newell and H.A. Simon. Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126, 1976.

J.K. O'Regan and A. Noë. A sensorimotor account of vision and visual consciousness. *Behavioral and brain sciences*, 24(5):939–972, 2001.

R. Penrose. *Shadows of the Mind*, volume 52. Oxford University Press Oxford, 1994.

H. Putnam. *Representation and reality*. Cambridge Univ Press, 1988.

E. Rasmusen. *Games and information: An introduction to game theory*. Wiley-blackwell, 2007.

C.W. Reynolds. Flocks, herds and schools: A distributed behavioral model. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 25–34. ACM, 1987.

S.J. Russell and P. Norvig. Chapter 5. Adversarial Search. In *Artificial Intelligence: a Modern Approach*, pages 161–201. Prentice hall, 2010.

J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(11):1203–1212, 1989.

R.C. Schank and R.P. Abelson. Scripts, plans, goals and understanding: An inquiry into human knowledge structures. 1977.

U. Schwalbe and P. Walker. Zermelo and the early history of game theory. *Games and economic behavior*, 34(1):123–137, 2001.

J.R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 1980.

B. Selman, R.A. Brooks, T. Dean, E. Horvitz, T.M. Mitchell, and N.J. Nilsson. Challenge problems for artificial intelligence. In *Thirteenth National Conference on Artificial Intelligence-AAAI*, page 4, 1996.

C.E. Shannon. Programming a computer for playing chess. *Philosophical magazine*, 41 (314):256–275, 1950.

C.E. Shannon. Computers and Automata. *Proceedings of the IRE*, 41(10):1234–1241, 1953.

A. Turing et al. Faster than Thought, chapter Digital Computers Applied to Games. *Sir Isaac Pitman, London*, 1953.

J. v. Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100(1): 295–320, 1928.

F.J. Varela, E. Thompson, and E. Rosch. *The embodied mind: Cognitive science and human experience*. The MIT Press, 1992.

J. von Neumann. First draft of a report on the edvac. 1945.

J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior*. Princeton University Press, 1953.

R.M. Whitaker and S. Hurley. An agent based approach to site selection for wireless networks. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 574–577. ACM, 2002.

N. Wiener. *Cybernetics*. Technology Pr., 1948.

H. Williams. Stochastic Diffusion Search Processes. Master's thesis, Goldsmiths, University of London, 2010.

D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical report, Citeseer, 1995.

D.H. Wolpert and W.G. Macready. No free lunch theorems for optimization. *Evolutionary Computation, IEEE Transactions on*, 1(1):67–82, 1997.

D.H. Wolpert and W.G. Macready. Coevolutionary free lunches. *Evolutionary Computation, IEEE Transactions on*, 9(6):721–735, 2005.

E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. In *Proceedings of the Fifth International Congress of Mathematicians*, volume 2, pages 501–504, 1913.