Contributions to Simulation-based High-dimensional Sequential Decision Making

Jean-Baptiste Hoock

To cite this version:

Jean-Baptiste Hoock. Contributions to Simulation-based High-dimensional Sequential Decision Making. Other [cs.OH]. Université Paris Sud - Paris XI, 2013. English. NNT : 2013PA112053 . tel-00912338

HAL Id: tel-00912338
https://theses.hal.science/tel-00912338v1

Submitted on 2 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thesis : Contributions to Simulation-based High-dimensional Sequential Decision Making

Jean-Baptiste Hoock

10/04/2013

Contents

1 Introduction **5**
1.1 Game, Planning, MDP . . . 6
1.2 Applications . . . 7
1.2.1 MoGo . . . 8
1.2.2 MASH . . . 8
Environment . . . 9
Methods . . . 10
Building an observation . . . 10
Decisions . . . 11
Rewards . . . 11
Goal . . . 11
A second application: touch 10 flags . . . 11
Brief summary . . . 12
1.3 Issues . . . 12
1.3.1 Differences between both problems . . . 12
1.3.2 Similarities between both problems . . . 13
1.3.3 Considering the number of states for the end game of Chess . . . 15
1.3.4 Main issues tackled in this thesis . . . 15
1.4 State of the Art . . . 17
1.4.1 Brute force . . . 18
1.4.2 Expert Knowledge . . . 18
1.4.3 Greedy algorithm . . . 20
1.4.4 Dynamic Programming . . . 21
1.4.5 QLearning . . . 23
1.4.6 Direct Policy Search . . . 24
1.4.7 MinMax . . . 24
1.4.8 Alpha-Beta . . . 26
1.4.9 A* . . . 27
Best-first search . . . 27

A* 29
1.4.10 Proof Number Search 29
1.4.11 Iterative deepening depth-first search 30
Depth-first search strategy 31
Breadth-first search strategy 31
Iterative deepening depth-first search strategy 31
1.4.12 Monte-Carlo Tree Search 32
1.4.13 Summary 33

I Online learning of Simulations : Monte-Carlo Tree Search 43

2 Simulating with the Exploration/Exploitation compromise (Monte-Carlo Tree Search) 45
2.1 Introduction 45
2.2 Monte-Carlo Tree Search 45
2.3 Scalability of MCTS 51
2.3.1 The limited scalability by numbers 51
2.3.2 Counter-examples to scalability 52
2.4 Message-passing parallelization 56
2.5 Conclusion 59

3 Reducing the complexity by local solving 61
3.1 Introduction 61
3.2 Glossary of Go terms 62
3.3 State of the art 64
3.3.1 Detection of the Semeai 64
3.3.2 Resolution of the Semeai 64
Without tree 64
With tree 64
3.4 GoldenEye algorithm: combining A* and MCTS 66
3.4.1 Statistical detection 66
The approach 68
Choosing parameters 69
Improvements 69
3.4.2 Resolution 72
Motivation 73
Describing the architecture 74
Search principle 75
Monte-Carlo Simulation 78

The update of the tree . . . 80
Choosing leaf . . . 81
Some improvements in the solving . . . 83
Summary . . . 85
3.4.3 Hierarchical solving . . . 86
3.5 Results . . . 88
3.5.1 Semeais from Yoji Ojima . . . 88
3.5.2 Comparison between improvements . . . 89
3.5.3 Study of the parameter *horizon* of the simulation . . . 96
3.5.4 Difficult semeais . . . 100
First Semeai: a win with ko . . . 100
Second Semeai: an open semeai . . . 102
Other semeais . . . 102
3.5.5 The accuracy of the score . . . 102
3.5.6 Failure cases . . . 104
3.5.7 Discussion . . . 105
3.6 Conclusion . . . 106

II Offline learning of Simulations : Direct Policy Search 111

4 Building a Policy by Genetic Programming (RBGP) 115
4.1 Introduction . . . 115
4.2 Framework and notations . . . 116
4.3 Experiments with RBGP algorithm . . . 120
4.3.1 Finding good shapes for simulations in 9x9 Go . . . 123
4.3.2 Finding bad shapes for simulations in 9x9 Go . . . 124
4.3.3 Improving 19x19 Go with database of patterns . . . 125
4.3.4 Improving 19x19 Go without database of patterns . . . 125
4.4 Progress rate of Racing-based GP . . . 126
4.4.1 Too mild rejection: $q >> \epsilon$ . . . 127
4.4.2 Well tuned parameters: $q = \epsilon$ . . . 128
4.4.3 No prior knowledge . . . 128
4.5 Experimental results . . . 129
4.6 Conclusion . . . 130

5 Learning prior knowledges 135
5.1 Notations . . . 136
5.2 Learning a Categorization of the Decisions . . . 137
5.2.1 Categorization . . . 137

5.2.2 Experiment on 10 flags problems 138
5.3 Learning a Clustering of the Features 141
5.3.1 Simulations for the generation of the collection 141
5.3.2 Clustering features 144
5.3.3 Metric 145
5.4 The policy 145
5.4.1 Memory for switching subgoals 145
5.4.2 Useful Macro-Decisions 147
5.4.3 Exploration for finding a subgoal 147
5.4.4 Vote for reaching a subgoal 148
5.5 Learning a Sequence of Subgoals 149
5.5.1 Monte-Carlo simulation 150
5.5.2 Policy of GMCTS 150
5.6 Experiments 152
5.6.1 The optimization of a draw of letters 152
5.6.2 Results 154
Results on the draw of letters 154
Results on the MASH application: "blue flag then red flag" 156
Discussions 156
5.7 Conclusion 158

III Conclusion **161**

Index **167**

Figures **170**

Tables **172**

Bibliography **173**

Acknowledgements

Je remercie chaleureusement mon Directeur de thèse Olivier Teytaud qui pendant 5 ans m'a laissé libre-cours à mon imagination; les algorithmes proposés dans cette thèse en sont le fruit. Cela a été un très grand plaisir de travailler avec toi.

Je remercie les rapporteurs de ma thèse Bruno Bouzy et Alain Dutech qui ont relu ma thèse avec une grande attention. J'ai beaucoup apprécié qu'on sente la présence du jeu d'échecs en arriere-plan dans le manuscrit et le regard que vous avez eu sur les algorithmes proposés dans cette thèse: par exemple, la façon dont se combinent les idées pour résoudre le problème MASH ou encore l'analogie faite entre l'algorithme de résolution de GoldenEye et la façon de penser d'un joueur d'échecs. Merci aussi pour toutes les critiques constructives qui me seront très utiles pour la rédaction de futurs documents scientifiques. Je remercie également Stéphane Doncieux, Tristan Cazenave et Jean-Claude Martin d'avoir accepté d'être membre de mon jury et ainsi de s'intéresser à mes travaux.

Je tiens à remercier toute l'équipe MASH, en particulier François Fleuret pour la gestion du projet MASH, Philip Abbet pour la gestion du simulateur 3D de MASH, Vladimir Smutny et Jan Sindler pour la gestion du bras-robot, André Beinrucker, Gilles Blanchard et Urun Dogan pour les discussions sur la sélection de features, Yves Grandvalet pour les discussions sur le clustering.

Je remercie Damien Ernst et Raphaël Fonteneau pour m'avoir accueilli un mois à l'institut Montefiore à Liège.

Je remercie aussi GRID5000 pour les nombreuses expériences qui ont pu être lancées. Sans la grille GRID5000, il aurait fallu plus d'1 siècle de calculs machines mono-coeur pour obtenir les résultats de RBGP.

Je remercie aussi toute l'équipe TAO avec laquelle j'ai travaillé pendant ces 5 années en tant qu'ingénieur pendant les 2 premières années et les 3 autres en tant que doctorant. Je remercie en particulier Nataliya Sokolovska, Hassen Doghmen, Adrien Couëtoux et Jérémie Decock des collègues de travail avec qui j'ai travaillé en étroite collaboration ainsi que Marc Schoenauer et Michèle Sebag les directeurs de notre équipe.

Je remercie tous mes amis en particulier Lucie Guesdon et Bernard Helmstetter qui ont eu une part très active dans la relecture du document mais aussi Fabien Teytaud pour les nombreuses idées dont nous avons discuté ensemble et pour avoir relu en partie ma thèse, mais également Marie-Christine et Jean-Claude Mignot, Marie-Nicole Rousseau, Soria Tebib, Marie et Brice Mayag, Celine Guesdon, Arpad Rimmel et Jean-Charles Seydoux qui m'ont soutenu et bien sûr Jacques Bibai qui a toujours été là et qui m'a beaucoup aidé lors de multiples discussions.

Merci aussi à mes plus grands fans; je pense à tous mes proches en particulier maman, mon père Jacques et mon frère Cédric qui m'ont toujours soutenu.

Pardon aux personnes que j'ai pu oublier de citer; il y aurait certainement moyen d'en faire une thèse. Encore une fois, merci à vous tous.

Abstract

My thesis is entitled "Contributions to Simulation-based High-dimensional Sequential Decision Making". The context of the thesis is about games, planning and Markov Decision Processes.

An agent interacts with its environment by successively making decisions. The agent starts from an initial state until a final state in which the agent can not make decision anymore. At each timestep, the agent receives an observation of the state of the environment. From this observation and its knowledge, the agent makes a decision which modifies the state of the environment. Then, the agent receives a reward and a new observation. The goal is to maximize the sum of rewards obtained during a simulation from an initial state to a final state. The policy of the agent is the function which, from the history of observations, returns a decision.

We work in a context where (i) the number of states is huge, (ii) reward carries little information, (iii) the probability to reach quickly a good final state is weak and (iv) prior knowledge is either nonexistent or hardly exploitable. Both applications described in this thesis present these constraints: the game of Go and a 3D simulator of the european project MASH (Massive Sets of Heuristics).

In order to take a satisfying decision in this context, several solutions are brought:

1. Simulating with the compromise exploration/exploitation (MCTS)

2. Reducing the complexity by local solving (GoldenEye)

3. Building a policy which improves itself (RBGP)

4. Learning prior knowledge (CluVo+GMCTS)

Monte-Carlo Tree Search (MCTS) is the state of the art for the game of Go. From a model of the environment, MCTS builds incrementally and asymetrically a tree of possible futures by performing Monte-Carlo simulations. The tree starts from the current observation of the agent. The agent switches

between the exploration of the model and the exploitation of decisions which statistically give a good cumulative reward. We discuss 2 ways for improving MCTS : the parallelization and the addition of prior knowledge.

The parallelization does not solve some weaknesses of MCTS; in particular some local problems remain challenges. We propose an algorithm (Golden-Eye) which is composed of 2 parts: detection of a local problem and then its resolution. The algorithm of resolution reuses some concepts of MCTS and it solves difficult problems of a classical database.

The addition of prior knowledge by hand is laborious and boring. We propose a method called Racing-based Genetic Programming (RBGP) in order to add automatically prior knowledge. The strong point is that RBGP rigorously validates the addition of a prior knowledge and RBGP can be used for building a policy (instead of only optimizing an algorithm).

In some applications such as MASH, simulations are too expensive in time and there is no prior knowledge and no model of the environment; therefore Monte-Carlo Tree Search can not be used. So that MCTS becomes usable in this context, we propose a method for learning prior knowledge (CluVo). Then we use pieces of prior knowledge for improving the rapidity of learning of the agent and also for building a model. We use from this model an adapted version of Monte-Carlo Tree Search (GMCTS). This method solves difficult problems of MASH and gives good results in an application to a word game.

Chapter 1

Introduction

Solving complex problems is always an exciting challenge.

For example, implementing a MCTS algorithm for playing chess better than me (amateur) is a challenge.

Another interesting perspective is to solve a complex problem by starting with almost nothing (e.g. without prior knowledge, agnostic framework, unknown goal, no - or few - assumptions on the architecture[1] ...). In the space exploration, robots have to solve difficult problems in an unknown environment without help. In 2012, the robot Curiosity explores the planet Mars sometimes without human intervention because communications take a lot of time between Earth and Mars. The robot makes autonomous decisions in order to reach a destination in an environment where humans have never walked. Possibly, when communications take too much time, the robot will evolve autonomously in a completely unknown environment and will have to determine itself what is interesting for humans.

This thesis has allowed me to do both. The first has been the creation of a chess engine who plays chess better than me. In addition to the state of the art, I have reused in my chess program an algorithm called MCTS which has been very efficient in the field of Go. The second was the elaboration of an algorithm for solving complex problems in an unknown environment (a 3D simulator). In this introduction, a first part is dedicated to present definitions of problems, the second presents applications on which I have worked[2]. The third part elaborates the issues and the fourth presents state of the art for solving those problems.

[1]No assumption on architecture can be useful for the genericity - the interface or the solver should not be redefined at each time the robot solves a new task.

[2]My work on chess is not presented in this thesis.

1.1 Game, Planning, MDP

During this thesis, I have worked on sequential decisions-making problems in the context of games, planning and MDP.

An agent (e.g a Chess player or a robot) interacts with its environment by successively making decisions (e.g. playing moves for the chess player or moving for the robot). The agent starts from an initial state until a final state in which the agent can not make decision anymore (e.g. a mate for Chess or the robot is fallen into a hole). At each time step, the agent receives an observation (e.g. the chessboard or the view of robot cameras) of the state of the environment. From this observation and its knowledge, the agent makes a decision which modifies the state of the environment. Then, the agent receives a reward and a new observation. The goal is that the agent learns what it must do in the environment (e.g win a chess game or explore the environment) thanks to rewards given to the agent.

This section defines terms relating to these problems.

A Markov decision process (MDP) is a 4-tuple $(S, A, P(.,.), R(.,.))$ where

- S is a finite set of states,
- A is a finite set of actions.
- $P_a(s,s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability that action a on the state s leads to state s' at time $t+1$.
- $R_a(s,s')$ is the immediate reward received after transition to state s' from state s with transition probability $P_a(s,s')$.

There is a state space S; elements of S are termed states; a component of state is termed feature. When making a decision d (i.e. action) in state s, we switch to a state s'; we note $s' = s.d$. In terms of graph, we can imagine that s and s' are nodes, and there is an edge (directed link) from s to s', with label d. A decision d is legal in s if there is an out-edge from s with label d. Going from s to s' by action d is termed a transition. Some states are equipped with a reward. A state for which the situation can not change anymore, and this whatever the decisions is called a final state (or terminal state).

A simulation is a sequence of states. Each state is labelled with

- either the name of the player who makes a decision at this state;
- or a probability distribution on legal decisions at this state.

There might be one player, or many players. When there are more than one player, the MDP is termed a stochastic game (SG). A history is a sequence of states with actions between them. It is legal if it is a sequence $s_1, d_1, s_2, d_2, s_3, ...$ of states such that $s_{k+1} = s_k.d_k$ for all k. A strategy is a mapping from legal histories to actions. A state is distinguished as the initial state.

A policy denoted π is a function which returns from a state s a decision d and γ is the discount factor which satisfies $0 < \gamma \leq 1$. The goal is to find a strategy π such that starting at the initial state and making decisions with the strategy, the expected sum of rewards over a simulation $\sum_{t=0}^{+\infty} \gamma^t R_{\pi(s_t)}(s_t, s_{t+1})$ is as high as possible. The discount factor γ is typically closed to 1. When there is an opponent, we want the sum of rewards to be as high as possible for a given opponent, or for the worst case on possible opponents, depending on the context.

Some simple elements have a huge impact on the difficulty of the problem:

- sometimes, we know all probability distributions involved, and sometimes we can just simulate them;
- sometimes, we can simulate, for any s and d, the transition $s' = s.d$;
- sometimes, typically in a physical system, if we want to simulate $s' = s.d$, then we must first find s (by successive transitions);
- sometimes, the environment is partially observable. A partially observable MDP is denoted POMDP.

A classical SG where we can easily simulate is 2 player zero-sum board game which is deterministic, sequential and with perfect information such as Chess and the game of Go. A 2 player zero-sum game is a game where a player's gain (or loss) is exactly balanced by the loss (or gain) of its opponent.

MDP solving consists in, given a MDP, finding an approximately optimal policy. For solving MDP, we distinguish 2 kinds of learning. In online learning, the learning is performed in real time during the simulations, whereas in offline mode the learning is performed once and for all in a separate training phase.

1.2 Applications

The applications on which I have worked belong to the domain of games, planning and POMDP. To support my thesis, I worked on 2 projects called MoGo (game) and MASH (planning and MDP). First I present the project

MoGo, a software of Go and then I describe 2 applications of the project MASH.

1.2.1 MoGo

MoGo is a software of Go, designed in 2006 by Sylvain Gelly and Yizao Wang. The game of Go (Fig. 1.1) is a 2 player game originated in China more than 2,500 years ago. Several rules (such as counting points) are available following the country. MoGo plays with Chinese Rules. The game of Go is a board game. The board is called a goban. The classical size of the goban is 19x19. Each player has stones (= pawns). The player, who starts the game, is Black whereas the second player is White. The Black player puts one of its black stones on an empty intersection of the board, then the turn passes to the White player who puts one of its white stone and alternatively, both players put a new stone on an empty intersection of the goban or not (= pass) until both players agree on the end of the game. Stones can be captured but are never moved from an intersection to another. A player passes to mean the end of the game. The game finishes when both players pass. The goal is to control a biggest territory than his opponent (*Right* of Fig. 1.1). A chain is a 4-connected group of stones of same color (Fig. 1.1). For more details about rules, http://gobase.org/studying/rules.

Even if the game of Go is an adversarial game (2 players), fully observable and easy to simulate whose rules are very simple, playing well is very difficult. So, this game is a real nice challenge for artificial intelligence. After the famous win of Deep Blue against the world champion of chess Garry Kasparov in 1997, the game of Go is become the new challenge of the Artificial Intelligence. MoGo was the first program who beats with reasonable number of handicap stones (i.e. MoGo has an advantage of 9 stones), the 7 August 2008 against Kim MyungWan 8p (a professional player with a very high-level grade) on KGS (Fig. 1.1). The precedent main result about a match between a program against a strong player was in 1995, the professional player Janice Kim defeated the top program of the day despite of a whopping 25 stone handicap.

Some other results are given in the following links:
http://senseis.xmp.net/?MoGo and newer, http://www.lri.fr/ teytaud/mogo.html

1.2.2 MASH

Many specialized methods (e.g. in Data Mining [Savaresi et al., 2002] [Defays, 1977]) are proved efficient in a lot of specific domains. Ensemble

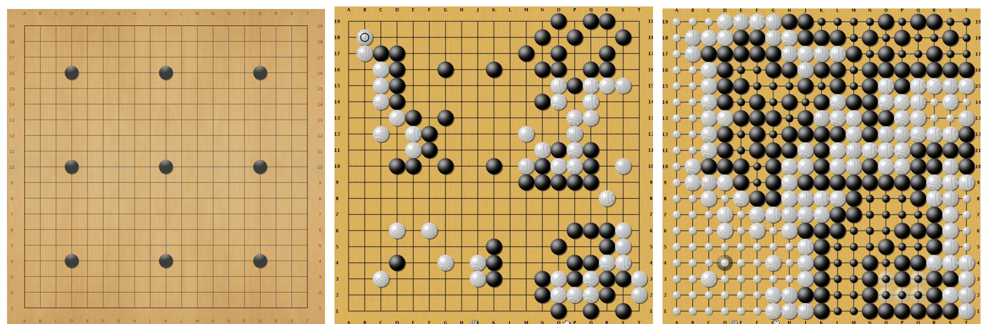


Figure 1.1: The famous game between MoGo and Kim MyungWan (8p). **Left:** White to play. MoGo starts with an advantage of 9 stones. **Center:** Black to play. Middle Game. Example of white (and black) chain is respectively O3 O2 P2 Q2 Q3 (and Q16 R16 R17). **Right:** Counting points. The game is finished and MoGo won by 1.5 points (black territories-white territories+0.5 of komi for avoiding draw games).

methods such as Adaboost [Freund and Schapire, 1995] try to take advantage of each specific method by building more collaborative approaches [Dubout and Fleuret, 2011]. In this context, the MASH project [3] was created. The aim of this project is to create new tools for the collaborative development of large families of feature extractors in order to start a new generation of learning software with great prior model complexity. Targeted applications of the project are (i) classical vision problems and (ii) goal-planning in a 3D video game and with a real robotic arm.

Below, we describe a goal-planning problem in the 3D video game. This problem is named "blue flag then red flag". The environment is first described, then methods, observations, decisions, and rewards. Finally, the best solution of this problem is given. In the last subsection, a second problem in the 3D video game is briefly discussed. This second problem is named "10 flags".

Environment

The environment is a square various size room with globally grey textures. The square room enclosed by 4 walls, contains an avatar and 2 flags. One of the two flags is red and the other flag is blue (See Fig. 1.2).

[3] http://mash-project.eu/wiki/index.php/About_the_MASH_project. MASH means MAssive Sets of Heuristics.

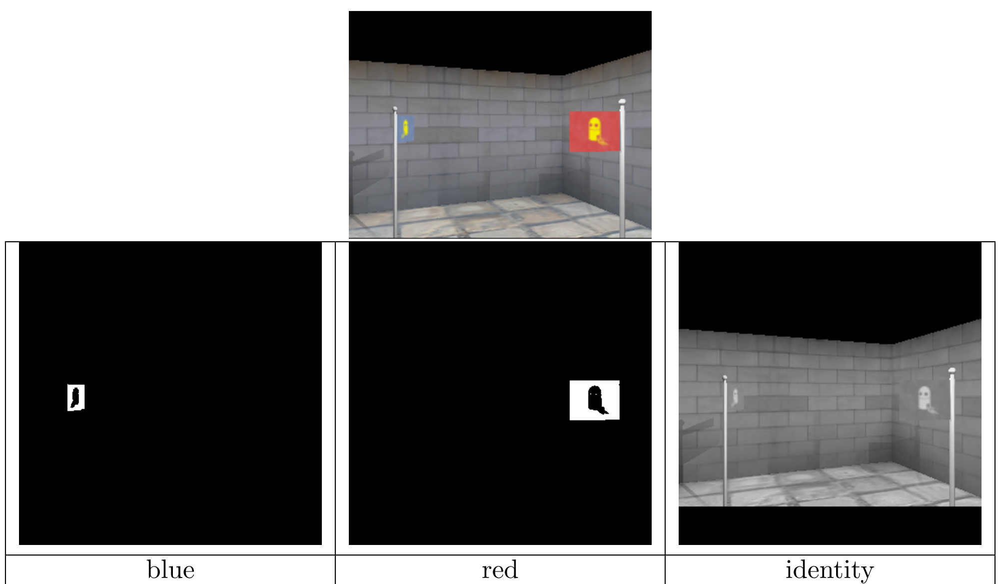


Figure 1.2: "blue flag then red flag" application. **Top:** a view of the 3D avatar. **Bottom:** Images resulting of the application of the 3 heuristics on the top view.

Methods

Methods are heuristics. More specifically, a heuristic is an image processing operator. The heuristic is applied on the view of the 3D avatar (Fig. 1.2). 3 heuristics are used:

- *red* which indicates if a pixel (an element of the view of 3D avatar) is red or not
- *blue* which indicates if a pixel is blue or not
- *identity* which transforms the value of the pixel to a gray-scale value.

Each heuristic gives around 100,000 features. In the observation, the solver ignores what heuristic provides the feature and has no notion of heuristics.

Building an observation

The cumulative number of features given by the 3 heuristics *red*, *blue* and *identity* are $3 \times 100,000$ features but it's too huge. In order to reduce the size of observation space, 10,000 features are randomly selected over the 300,000 features. The observation given to the solver will be always made with these 10,000 features.

Decisions

There are 4 decisions:

- 0 (which means "go forward")
- 1 (which means "go backward")
- 2 (which means "turn right")
- 3 (which means "turn left")

Note that the solver does not know the meaning of decisions.

Rewards

- Hit a wall: -1
- Touch the first time the blue flag: +5
- Touch (or hit) another time the blue flag: 0
- Hit the red flag without touched the blue flag: -5
- Hit the red flag after having touched the blue flag: +10
- Else: 0.

This definition of rewards is not informative, because most often, the avatar moves without touching anything and receives a reward of 0.

Goal

A final state is reached if and only if once having touched the blue flag, the red flag is touched. The best cumulative reward is 15, by first touching the blue flag and then touching the red flag without hitting wall. We can distinguish 2 subgoals (i) touch the blue flag and then (ii) touch the red flag.

A second application: touch 10 flags

The environment is the same except 3 red flags and no blue flag (Fig. 1.3). Decisions don't change. Only 2 heuristics (in state of 3) are used: *red* and *identity*. The task is failed (bad final state) if after the application of 150 decisions, no red flag is touched, then a reward of -1000 is generated. Touching a red flag gives a reward of 10 points. Finally, when a red flag is touched, it disappears and reappears in another place. The final good

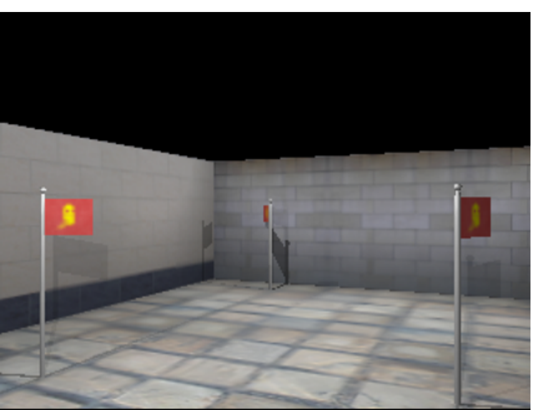


Figure 1.3: “10 flags“ application.

state is reached when 10 flags touched. The optimal cumulative reward is so $100 = 10 \times 10$ and is obtained if the wall has not been hit (else reward = -1 as in the first application described before).

Brief summary

MASH is partially observable, heavy to simulate, we have no access to the transition probabilities, and we can not undo, we are totally agnostic on the problem (We ignore the semantic of actions/observations).

1.3 Issues

In order to understand issues tackled in this thesis, we will compare the game of Go and the MASH applications.

In a first section, we will discuss differences between the two applications. In a second section, we will discuss similarities; in a third section, we will give more details on one famous specific case, namely Nalimov tables; and in a fourth section, we will explain the issues tackled in this thesis.

1.3.1 Differences between both problems

Reminder, the game of Go is an adversarial game (2 players), fully observable and easy to simulate whereas MASH is partially observable, heavy to simulate, we have no access to the transition probabilities, we can not undo and we are totally agnostic on the problem.

Another difference can be remarked. Although the decision space is discrete and not too huge in both applications, the decision space is very small for the MASH applications (only 4 decisions), whereas for the game of Go, the number of decisions per state is greater than 200 in average.

1.3.2 Similarities between both problems

But in both cases, we can notice 4 constraints:

1. the number of states is huge
2. the reward carries little information
3. small probability to reach **quickly** a **good** final state
4. no knowledge or inexploitable knowledge

Illustration(s) of these constraints:
1 - A lot of states can be due to:

- Case 1 - A : Stochasticity (e.g. a real case such as a robot-arm in a noisy environment. In the MASH application, the simulator is stochastic)
- Case 1 - B : Dimensionality of the state space (e.g. in the MASH problem, $dim = 10,000$ because of the random sampling of 10,000 features and in MoGo, $dim = 361$ - if each component of the state is one intersection of the board of size 19x19)

It leads to the Exponential states. In the MASH problem, $\#S > 2^{10000}$; we assume that a feature can have at least 2 values. In MoGo, $\#S \simeq 10^{171}$; this information can be seen and compared with other games at en.wikipedia.org/wiki/Game_complexity.

2 - Reward which carries little information is: a reward giving little information such as a direction for reaching a goal. Most of cases, you obtain almost always the same reward during a simulation. For the MASH application, the avatar receives very often the reward 0 because of moving without hitting a wall or a flag. The reward does not give a good information such as a distance to a flag; there is no idea about the direction to accomplish the task.

For the MoGo application, the reward information is delayed. The reward is given at the end of the game (i.e. at the end of a simulation of 150 moves in average) and represents a win or a loss. When no reward, we can consider the reward is 0 like in MASH.

3 - Small probability to reach ***quickly*** *a* ***good*** *final state:* The number of final states is weak. For the MASH application, the only case to reach a final state is to touch the blue flag and the red flag. There are more situations in which the avatar has not touched the red flag and still more situations in which the avatar has not touched the blue flag (with or without touching the red flag).

For MoGo application, a final state can not be reached without defining clearly territories (which assumes to play around 150 moves).

- Case 3 - A : The length of an optimal (or good) strategy is big. The more the strategy is long, the more the "quickly" is difficult to realize. For the MASH project, a complete turn requires already 60 decisions and sometimes, through a room requires more 60 decisions. So, for touching optimally a flag, 60 decisions can be required times by 2 for touching optimally the second flag.
  For the MoGo project, the length of a game is closed to 150 moves (i.e. 75 decisions per player). Against a strong player, the win is a very difficult task to realize. Either the loss is quick, or the winner is not clearly defined before 150 moves.

- Case 3 - B : Reach a bad final state is easy. The more there are bad final states, the more the "good" is difficult to realize. For the second application of MASH, touching 10 flags, after 150 decisions, if a flag is not touched, the task is failed. In MoGo, a decision which leads to a bad state (a loss) is very easy against a good player because only one decision can destroy all your good decisions you have already played and lead to a defeat even if you played then optimally.

4 - No prior knowledge or inexploitable prior knowledge: For the MASH project, there is no prior knowledge. We are no hypothesis on the structure of the state, no idea of the meaning of values of components of the state. The number of decisions and the semantic of decisions are unknown and the reward carries little information.

Inexploitable knowledge can mean:

- Case 4 - A : wrong knowledge (e.g. false human idea).

- Case 4 - B : knowledge that we don't know how to use. Before 2006, classical algorithms did not exploit very well with basic prior knowledge such as patterns in the game of Go.

- Case 4 - C : too computationally expensive knowledge. In the game of Go, it's possible to have an efficient evaluation function, but it requires

Table 1.1: The first column Application is the name of the application. The second column State is an evaluation of the dimensionality of the state space (followed sometimes by the number of states). The third column is the number of possible decisions. The fourth column is the knowledge of the transition function. The fifth column is: the environment is partially observable (yes) or not (no). The sixth column is an idea of the number of decisions before reaching a good final state. smtm, avg and decs are respectively the abbreviation of sometimes, average and decisions. For all applications, the reward (different of evaluation function) carries little information.

Application	State	Dec	Transition	Prior	PO	Solution
Chess Endgame	$< 10^{47}$	15 in avg	Yes	Not a lot	No	smtm, more 100 moves
Chess	10^{47}	30 in avg	Yes	A lot	No	50 moves
Go 9x9	10^{38}	60 in avg	Yes	Inexploitable	No	45 moves in avg
Go 19x19	10^{170}	250 in avg	Yes	Inexploitable	No	150 moves in avg
MASH	$2^{10,000}$	4	No	No	Yes	smtm, more 100 decs

a lot of analysis such as predicting territory [Bouzy, 2003], determining the status of a chain...

1.3.3 Considering the number of states for the end game of Chess

Nalimov tablebases[4] are database for end games of Chess. The algorithm generating the database is inapplicable for the Game of Go because along the game, stones are added on the board whereas in Chess pieces are removed by capture. However, 3 over 4 constraints are checked (See Fig. 1.4). What is about the last constraint which concerns the number of states? In 2012, we can consider that the number of states is not too large with 6 pieces because 6-pieces tablebases have been generated. On the contrary, 7-pieces is too huge.

1.3.4 Main issues tackled in this thesis

A brief summary of properties of applications is presented in Tab. 1.1.

This study of properties leads to the following question: **How to solve complex tasks under these 4 constraints? Reminder, the 4 constraints are:**

1. **the number of states is huge**

[4] http://www.k4it.de.

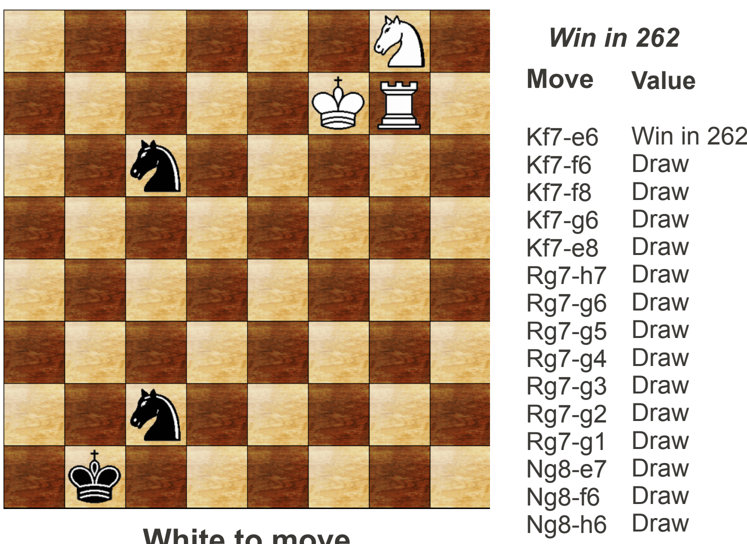


Figure 1.4: From http://www.k4it.de. Nalimov tablebases: an example which gives an idea of "the state space is huge or not", today (in 2012). In the end of the game of Chess, 3 constraints over 4 are checked. (i) The reward carries little information (win or loss), (ii) the probability to reach quickly a good final state is small. In the example, more 200 moves are necessary for winning and only one move (i.e. Ke6) wins. All other moves lead to a draw. (iii) There is less available prior knowledge as in the middle game. Understanding moves are very difficult even for a grandmaster, a very strong player in chess (Why Ke6 is the only winning move?). (iv ?) What is about the number of states? Because 6-pieces are solved but not 7-pieces, we can consider the number of states is not too huge with 6 pieces on the board, but becomes huge when added a seventh piece.

2. **the reward carries little information**

3. **small probability to reach quickly a good final state**

4. **no knowledge or inexploitable knowledge**

More specifically, for the game of Go, how to improve a player in order to finally beat a professional player? For the MASH application, how to find the task and accomplish it?

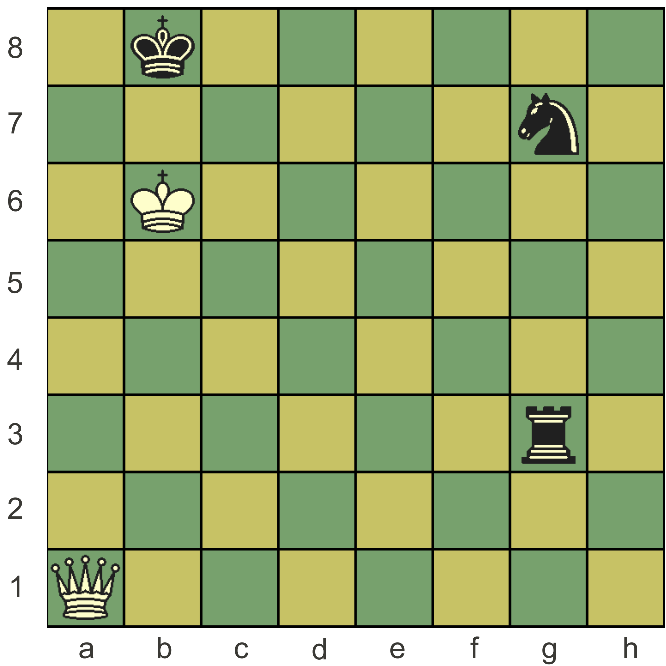


Figure 1.5: Illustration of one problem. White to play. Mate in 2 moves. The goal is to find the optimal move $Qa7+$ (the White **Q**ueen located at *a1* goes to $a7$ and gives chess denoted by + to the Black King located at *b8*).

1.4 State of the Art

For solving these kinds of problems, some algorithms exist.
We will illustrate some of them on Chess (Fig. 1.5). We use 325, 500, 950, $+\infty$ as heuristic values for respectively Knight, a Rook, a Queen, a King. In Fig. 1.5, there are one black Knight at g7, one black Rook at g3, one white Queen at a1, one white King at b6 and one black King at b8. A state is a position of chess plus the color of the player to move. The Queen can move along columns, rows and diagonals, the Rook can move along columns and rows and King one of 8 neighbours squares which is not controlled by the opponent [5]. If the King of the player is in check (i.e. captured by the opponent by changing the player to play), the player must remove chess by moving the King for example. If this situation is impossible, the game is lost for the player in check. See http://en.wikipedia.org/wiki/Rules_of_chess for a complete rule.

[5]Knowing the movement of the Knight is not necessary for understanding.

Algorithms presented in this introduction are the brute force, expert knowledge, greedy algorithms, dynamic programming, QLearning, Direct Policy Search and some tree searches such as MinMax, Alpha-Beta, A*, Proof-Number Search and Monte-Carlo Tree Search. The Nested Monte-Carlo Search algorithm will be briefly described.

1.4.1 Brute force

By a full search, all policies within a well-defined subspace of policy are tried and the brute force chooses the best one. Main problems are:

- the number of policies which can be extremely large, or even infinite
- the variance of the return may be large, in which case, a large number of simulations is required to accurately estimate the return of each policy.

For these reasons, the brute force can not work on both applications MASH and the Game of Go.

1.4.2 Expert Knowledge

Expert knowledge uses prior knowledge in order to infer rules which can be useful to find/solve more efficiently a task. Expert knowledge can have different forms such as mathematical rules, patterns.

For example, the Expert knowledge given in Fig. 1.6 is a pattern and can be applied on Fig. 1.5. For White, a7 (A2 on Fig. 1.6) is controlled by the White King b6 and the White Queen a1; for Black, only the black king controls a7 and c7 (C2 on Fig. 1.6). c7 is controlled by the white King for White. The pattern is checked. The White King is not in check and the Queen controls directly A2. Another example where the pattern is matched is Fig. 1.7.

The policy returns the decision which corresponds to play the Queen in the location A2 in Fig. 1.6. With this expert knowledge, the optimal move will be played (i.e. *Qa7+* in Fig. 1.5 and *Qc7+* in Fig. 1.7).

Defining an expert rule is not easy. In some cases, the rule is too fuzzy or useless and can not be exploited. In some other cases, a rule falsely restricts the search space because the knowledge can be wrong, false or incomplete. In our example (Fig. 1.6), the pattern is incomplete. 3 other rules are necessary: In Fig. 1.6, first, no white piece at the location C1, second, the White Queen can move without making chess its own King and third, no black piece controls indirectly the square C1 or C3 by transparency through the White

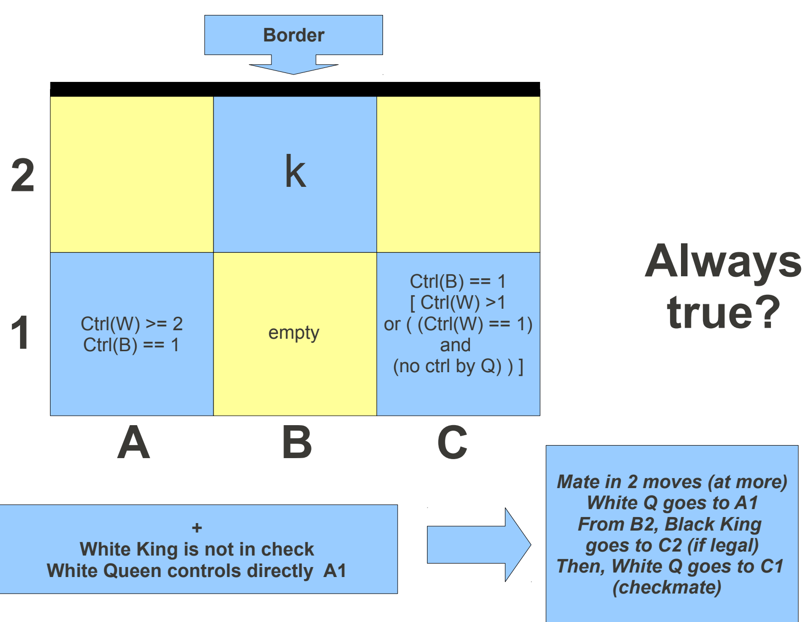


Figure 1.6: Expert knowledge (Pattern). White to Play. The 6-squares represent a pattern which should be checked on the board. The letter k is the location of the Black King. $Ctrl(color)$ is the number of pieces of the player *color* which controls the location. (W for White and B for Black). Q means Queen. In the square C1, the condition "no ctrl by Q" means that the White Queen does not control the square C1. **Bottom Left** is 2 other rules for a complete checking of the pattern. If the pattern is checked, then we can apply the conclusion shown by the arrow. But if the pattern is checked on the board, is the conclusion always true?

Queen. Fig. 1.8 gives 3 counter-examples. In Fig.1.7, add a white rook at c7, the Queen can not move to C7 because the piece can not capture a piece of own side; add a black Rook at h3, the Queen can not move to c7 without making its own King in check. For a complete correctness of this rule, many positions have been generated and a checking of the availability of the mate has been done.

In conclusion, it's hard to have a robust rule. Moreover, it lacks genericity; this illustrative rule is too specific for being used in another application.

In the game of Go, there are too many expert rules to add in order to have a perfect player. In the MASH application, there is no prior knowledge.

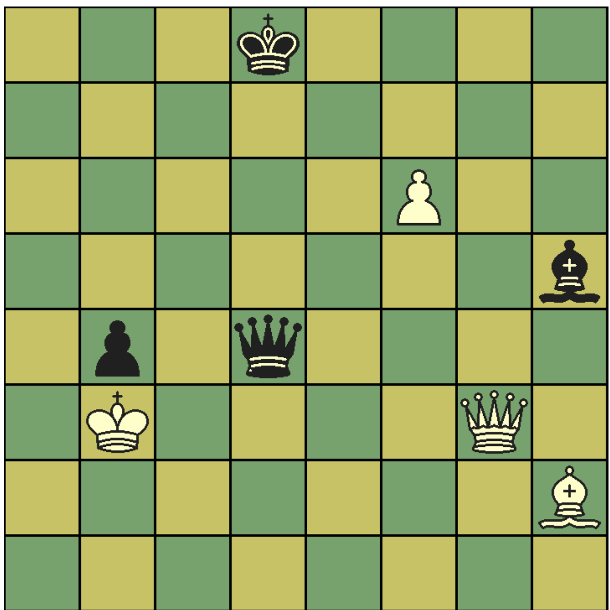


Figure 1.7: White to Play. Mate in 2 moves. The square c7 (A2 on Fig. 1.6) is directly controlled by the Queen. The bishop h2 and the queen g3 control this square ($Ctrl_{c7}(W) = 2$). Only the black King controls c7 ($Ctrl_{c7}(B) = 1$) and e7 (C2 on Fig. 1.6). The pawn controls e7. The white King is not in check. So, all conditions are filled to checkmate.

1.4.3 Greedy algorithm

A greedy algorithm such as the algorithm of Prim [Prim, 1957] chooses step by step an optimal local choice in order to find the optimal solution or a solution close to the optimal. For evaluating a state s, an evaluation function is required.
For the game of Chess, the evaluation function can be the gain of material or minus the number of answers of the opponent. In our example (Fig. 1.5), with the gain of material as the evaluation function, greedy algorithm chooses *Qxg7*, a very suboptimal move (From a winning position, the move can lead to a losing position). *Qxg7* is the only move winning material. If the evaluation function is minus the number of opponent answers, then the greedy algorithm can choose the optimal move *Qa7+*, but a second move *Qa8+* (very bad) can be also chosen because this move has the same value (i.e. Both moves leave a single answer to the opponent.).

A basic greedy algorithm can not be applied on MASH. Indeed, the evaluation function corresponds to return the reward, but without prior knowledge, the evaluation function is necessarily the reward. The reward carries

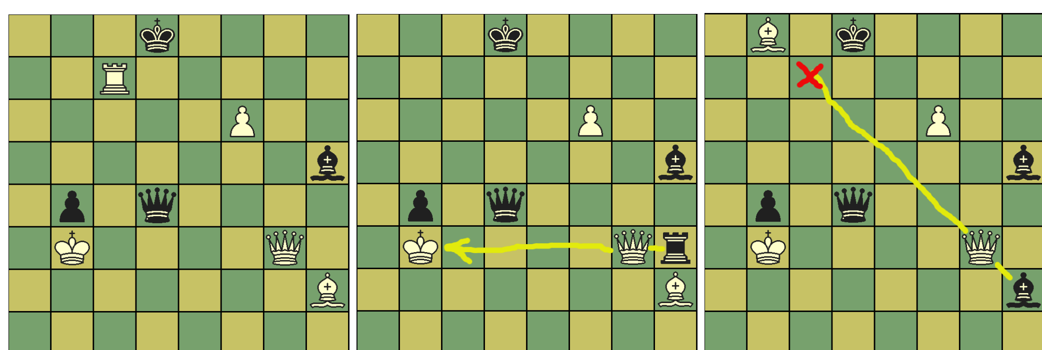


Figure 1.8: White to Play. All conditions are filled too. But here, the mate in 2 moves is impossible.

little information. Moreover, MASH is model-free, we can not undo; applying the decision is required in order to retrieve the reward. For the application of Go, an efficient evaluation function requires a lot of "handcrafted-works".

1.4.4 Dynamic Programming

The dynamic programming (DP) [Bellman, 1957] is the computation of the V function by backwards application of Bellman's operator, i.e. until convergence apply (1.1) and (1.2) in turn:

- for each state s,

$$\pi(s) = \operatorname*{argmax}_a \sum_{s'} P_a(s,s')(R_a(s,s') + \gamma V(s')) \tag{1.1}$$

- for each state s,

$$V(s) = \sum_{s'} P_{\pi(s,s')}(R_{\pi(s)}(s,s') + \gamma V(s')) \tag{1.2}$$

The value iteration is the same as DP without storage of optimal actions (which are recomputed when needed). An enhancement of Value Iteration is Focus Topological Value Iteration [Dai et al., 2009].

Another variant is the policy iteration. We apply many steps (1.2) (until convergence) before re-applying step (1.1). Some examples of Policy Iteration are the Rollout Classification Policy Iteration [Dulac-Arnold et al., 2012] or Online least-squares policy iteration [Buşoniu et al., 2010].

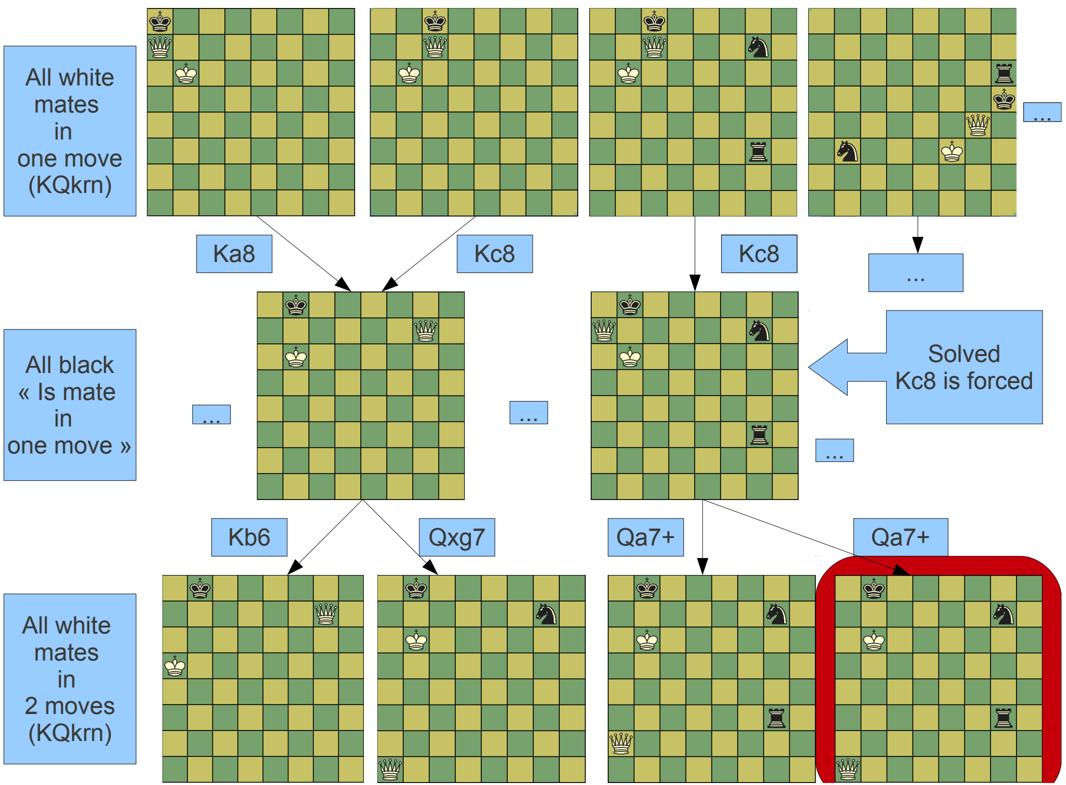


Figure 1.9: Dynamic Programming : With the material **K**ing **Q**ueen for White, search of all mates in one move and **k**ing, **r**ook, k**n**ight for Black and then search all "is mate in one move" and then search all mates in 2 moves ... until the current position is found.

The main idea of DP is to find all final states and then updates by back-propagation - as if we back in time (Fig. 1.9).

We solve our small problem (Fig. 1.5) by starting from goal states and go back to the current position. We generate all good final states (i.e. all mates in one move for White). Then we generate all positions where whatever the Black decision, White mates in one move. After this, we generate all position where White mates in 2 moves, and so on until the current position given in Fig. 1.5) is found or no new mate has been found. The second case could occur if White can not win. The algorithm can be applied because of the small number of pieces on the board.

The illustrative algorithm described before can not be applied due to:

- the number of final positions is too large for the game of Go

- the set of final states is unknown for the MASH application.

More generally, for DP, applications on MASH are model-free and can not be applied.

1.4.5 QLearning

The QLearning algorithm has been designed by Watkins [Watkins and Dayan, 1992]. One iterates updates of the value $Q(s, a)$ along trajectories following:

$$Q(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha_t(s_t, a_t)}_{\text{learning rate}} \times \left[\overbrace{\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \underbrace{\max_{a_{t+1}} Q(s_{t+1}, a_{t+1})}_{\text{max future value}}}^{\text{learned value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right] \tag{1.3}$$

Some other variants inspired of QLearning exist such as SARSA [Rummery and Niranjan, 1994] or with eligibility trace [Singh et al., 1996]. From samples of trajectories, Fitted Q Iteration [Ernst et al., 2005] approximates the Q function with a regressor such as Neural Networks or Random Forests [Breiman, 2001].

For computing the Q values, simulations are done. We illustrate the idea of the algorithm on our problem (Fig. 1.5) with 5 random simulations. Only for QLearning, the game is seen as a stochastic game coming from the fact that the opponent is not modeled. We assume that in the implementation, the solver has no idea of the Black player. These 5 "random" simulations are:
1. Qa7+ (Kc8) 2. Qxg7
1. Qxg7 (Rg4) 2. Qxg4
1. Qxg7 (Rh3)
1. Qf6 (Kc8) 2. Qf7 (Kd8) 3. Qa7 (Kc8) 4. Qc7#
1. Qa7+ (Kc8) 2. Kc5
Even if the simulation *1. Qa7+ (Kc8) 2. Qc7#* has not occurred, an algorithm of the family of QLearning can find the optimal way to win. Indeed, the situation met after *1. Qa7+ (Kc8)* is met once again in the fourth simulation after *1. Qf6 (Kc8) 2. Qf7 (Kd8) 3. Qa7 (Kc8)* in which the optimal move *Qc7#* is played. Through simulations and updates, we can assume that useful information will be propagated (for example at the 5th simulation) and the algorithm will be able to play the optimal way to win (i.e. *1. Qa7+ (Kc8) 2. Qc7#*).
The main idea is that the optimal path has been never simulated but it can

be found.
QLearning can not be applied in the game of Go; one reason is that QLearning with lookup table requires to keep all visited states and so, it takes too much memory. An extrapolation tool able to generalize on a very abstract space such as positions of Go should be defined. What is very difficult. For MASH applications, the same reason occurred. Moreover, finding trajectories which lead to a good final state is very difficult.

1.4.6 Direct Policy Search

A direct policy search (DPS) is an optimization of a parametric policy on simulated costs. There exist a lot of Direct Policy Search such as gradient methods, stochastic methods or evolutionary algorithms [Heidrich-Meisner and Igel, 2008]. When parameters are discrete without notion of order, gradient methods can not be applied.

For our problem (Fig. 1.5), a parametric policy (Alg. 1) is defined. $param1$ and $param2$ are parameters to optimize. We can assume that the Direct Policy Search is a Random Search (parameters are tried randomly). With $param1 \geq 0$ (e.g. $param1 = 1$) and $param2 = 0$, the DPS will see after simulations that the policy is better than $param1 < 0$ (e.g. $param1 = -\infty$) and $param2 > 0$ (e.g. $param2 = +\infty$) (See Fig. 1.10).

After learning, when the policy is with

- $param1 < 0$ and $param2 > 0$, it plays randomly.
- $param1 \geq 0$ and $param2 > 0$, the policy plays *Qxg7*, the only move which wins material.
- $param1 \geq 0$ and $param2 > 0$, it will play again a random move but the improvement can be seen after because if ever the opponent lets his knight without defense catchable by the opponent (e.g. after *1. Qc4 (Ra3)*), then the policy will return a good move (the capture of the Knight).

Direct Policy Search can address model-free problems and is a good candidate for solving MASH applications. For the game of Go, the difficulty lies in designing the policy. DPS could be used as a second stage for optimizing other parametric algorithms.

1.4.7 MinMax

In 2-player adversarial large games, a good player must take into account decisions by the opponent. Fortunately, for a 2-player adversarial board

Algorithm 1 An example of parametric policy to optimize. $param1$ and $param2$ are initially fixed to respectively $-\infty$ and $+\infty$. Capturing an opponent piece is good, so $param1$ should be optimally fixed to a value ≥ 0. Capturing a piece of less value is very often good $Value(s[dest]) > Value(s[loc])$, but if the value of the opponent piece is smaller than the value of the attacking piece and if this opponent piece has a protection, then the capture is often not good. So, $param2$ should be optimally fixed to 0.

```
Function Policy(s)
Let mv_out the "returned" move initialized randomly
for mv ∈ list of moves from the state s do
  Let loc the starting location of the move mv
  Let dest the destination of the move mv
  if Value(s[dest]) > param1 then
    if Value(s[dest]) > Value(s[loc]) ∨
    Control(s[dest], opponent) ≤ param2 then
      mv_out = mv
      break
Return mv_out
```

game which is sequential, symmetric with perfect information, it is easy to implement a model. When this is done, simulations can be run on the model. Chess and the game of Go are 2 examples. MASH applications are not a 2-player board game and above all, is model-free. Implementing a model is not possible; building automatically the model seems to be very hard. Algorithms MinMax, AlphaBeta, A^*, Proof Number Search and Monte-Carlo Tree Search presented hereafter requires a model and can not be used on the MASH applications. These algorithms build a subtree in the tree of possible decisions.

Fig. 1.11 illustrates the tree of possible decisions from the position in Fig. 1.5. Of course, all possible decisions are not represented but by pruning (which consists in neglecting some possible decisions), this kind of tree can be obtained. A node corresponds to a position of chess (including the player to play). The root node corresponds to the position given Fig. 1.5 and an edge is a move.

MinMax [Heineman et al., 2009] is one of the main tree search algorithms for 2 player zero-sum games. The algorithms belongs to the family of Dynamic Programming algorithms. The game tree is visited in order to move

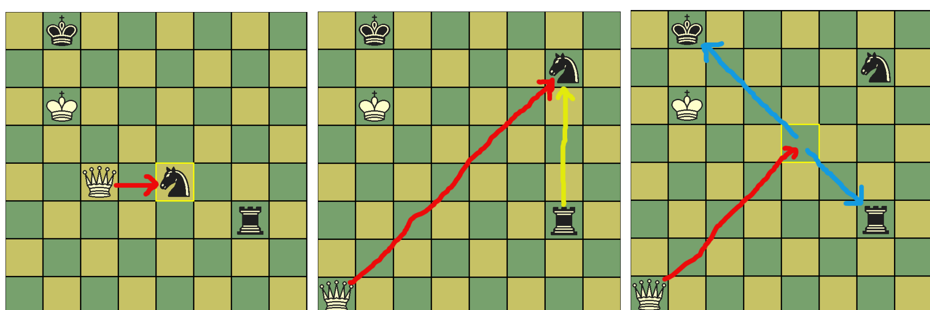


Figure 1.10: Direct Policy Search **Left:** By simulations, I will learn that capturing a piece is good. So, in Fig. 1, $param1$ will be fixed to 0 (for example). $param2$ is still fixed to its initial value $+\infty$. **Center:** But, after running new simulations, I will learn that capturing a protected piece may be bad. $param2$ will be fixed to 0. The direct policy search searches best values of $param1$ and $param2$. Perhaps, I will learn more and more complex rules such as the Example of 2 threats at **Right**. However, it's dependent of the structure of the policy and the direct policy search which is used. With the structure of policy given by Fig. 1, the rule of "2 threats" can not be learnt.

up recursively the best value α_{best} to the root of the tree. But at each step, the player to play maximises its score, whereas the opponent minimizes it. The NegaMax (in Chapter 7 of [Heineman et al., 2009]) is the MinMax algorithm reposing on the fact that $\max(a, b) = -\min(-a, -b)$. NegaMax is simpler; we present this version in Alg. 2.

The policy returns the move (i.e. one edge of the root of the tree) which leads to the value computed by MinMax.

One of the main troubles of the algorithm is the *horizon* effect. On Fig. 1.12, $depth = 1$ is not sufficient to see that capturing the Knight is bad. $depth = 2$ is better because the move is no longer chosen, but no differentiation between *Qe5+* and *Qa7+* is done. With $depth = 3$, the optimal move is found.

1.4.8 Alpha-Beta

Alpha-Beta [Knuth and Moore, 1975] is an optimization of MinMax algorithm by avoiding to study useless branches. This algorithm is not an approximation of MinMax; results given by MinMax and Alpha-Beta will be always the same. Alpha-Beta cuts some branches of the tree. The two kinds of cut-off are presented in Fig. 1.13.

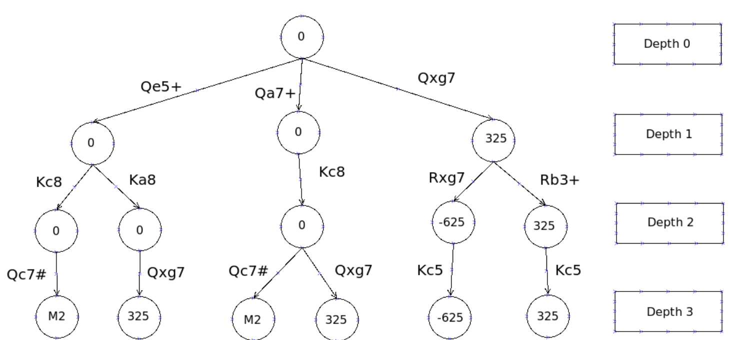


Figure 1.11: Tree of considered moves from the problem in Fig. 1.5. M2 means "Mate in 2 moves". For comparison with other values, M2 = 99,998 = 100,000 - 2 (A mate is equals to 100,000 minus the number of moves to mate). Except M2, evaluation given in the interior of a node *nd* is the cumulative sum of all winning materials for the first player from the root to the node *nd* - e.g. From the root, the move *Qxg7* wins a knight, the evaluation is 325 (value of a knight). After the move *Qxg7*, *Rxg7* wins the Queen (evaluated to 950). In the resulting node, the evaluation is of $325 - 950 = -625$. All possible decisions are not represented except the answer *Kc8* after the move *Qa7+* and the answers *Ka8* and *Kc8* after the move *Qe5+*.

The algorithm (in its "Nega" version) is shown in Alg. 3.

In Fig. 1.14, thanks to an alpha cut-off, 2 nodes are not visited in comparison with nodes visited by MinMax at $depth = 3$ (See Fig. 1.12).

The algorithm does not work very well in the Game of Go, because evaluation function is either too slow or too weak and the number of moves per position is too big (around 250 in average).

1.4.9 A*

A* uses best-first search strategy.

Best-first search

Best-first search is a search algorithm with in general a greedy strategy. Best-first search explores a graph by expanding the most promising node chosen according to a specified rule. In our example, the rule can be one of 2 examples of evaluation function given in Section 1.4.3.

Algorithm 2 MinMax algorithm (Negamax version). It moves up the best value to the root of the tree.

Function $minimax(node, depth)$
if node is a terminal node or $depth \leq 0$ **then**
  Return the heuristic value of node
$\alpha = -\infty$
for $child \in node$ **do**
  $\alpha = \max(\alpha, -minimax(child, depth - 1))$
{evaluation is identical for both players}
Return α

Algorithm 3 AlphaBeta algorithm ("Nega" version). As MinMax, it moves up the best value to the root of the tree.

function $alphabeta(node, A, B, depth)$ $\{A < B\}$
if $node$ is a terminal node $\vee$ $depth \leq 0$ **then**
  Return the value of the node $node$
else
  $best = -\infty$
  for each child of $node$ **do**
    $val = -alphabeta(child, -B, -A, depth - 1)$
    if $val > best$ **then**
      $best = val$
      if $best > A$ **then**
        $A = best$
        if $A \geq B$ **then**
          Return $best$ {The cut-off}
  Return $best$

A*

The A* [Hart et al., 1968] is a search algorithm of path in a graph between an initial node and a final node. At each timestep, it chooses a leaf of tree to expand using a best-first search strategy.

The B* algorithm, [Berliner, 1979], a variant of A* has been extended to 2-player deterministic zero-sum games. In fact, the only change is to interpret "best" with respect to the side moving in that node. So you would take the maximum if your side is moving, and the minimum if the opponent is moving. Equivalently, you can represent all intervals from the perspective of the side to move, and then negate the values during the back-up operation.

In our illustration of the principle of A* (Fig. 1.15) on the same problem (Fig. 1.5), all leaves (in state of one) with a same maximal value are expanded. An iteration corresponds to expand all leaves bounded with this best value. Because of the principle "my opponent minimizes my score which I maximise", expanded leaves have not necessarily the best value but have best value according to the MinMax principle. That's why in the third iteration in Fig. 1.15, expanded leaves are nodes after *Qe5+* and *Qa7+* with the value of 0 and not the leaf after *Qxg7-Rb3+* with a value of +325.

1.4.10 Proof Number Search

Proof Number Search (PNS) is a game tree search algorithm invented by Victor Allis [Allis et al., 1994], with applications mostly in endgame solvers, but also for sub-goals during games. Some variants are PDS-PN [Winands et al., 2002] or PN^2 [Nagai, 1998].

The following description comes from the article entitled *PDS-PN: A New Proof-Number Search algorithm* [Winands et al., 2002].
Proof-number (PN) search is a best-first search algorithm especially suited for finding the game-theoretical value in game trees [Allis, 1994]. Its aim is to prove the true value of the root of a tree. A tree can have three values: true, false or unknown. In the case of a forced win, the tree is proved and its value is true. In the case of a forced loss or draw, the tree is disproved and its value is false. Otherwise the value of the tree is unknown. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function to determine the most-promising node to be expanded next [Allis et al., 1994]. In PN search this node is usually called most-proving node. PN search selects the most-proving node using two criteria: (1) the shape of the search tree (the number of children of every internal node) and

(2) the values of the leaves. These two criteria enable PN search to treat game trees with a non-uniform branching factor efficiently.

Below we explain PN search on the basis of the AND/OR tree depicted in Fig. 1.16, in which a blue circle denotes an OR node, and a red circle denotes an AND node. The numbers to the right of a node denote the proof number (upper) and disproof number (lower). A proof number represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a disproof number represents the minimum number of leaves which have to be disproved in order to disprove the node. Because the goal of the tree is to prove a forced win, winning nodes are regarded as proved. Therefore, they have proof number 0 and disproof number inf. Lost or drawn nodes are regarded as disproved. They have proof number inf and disproof number 0. Unknown leaf nodes have a proof and disproof number of unity. The proof number of an internal AND node is equal to the sum of its children's proof numbers, since to prove an AND node all the children have to be proved. The disproof number of an AND node is equal to the minimum of its children's disproof numbers. The disproof number of an internal OR node is equal to the sum of its children's disproof numbers, since to disprove an OR node all the children have to be disproved. Its proof number is equal to the minimum of its children's proof numbers. The procedure of selecting the most-proving node to expand is the following. We start at the root. Then, at each OR node the child with the lowest proof number is selected as successor, and at each AND node the child with the lowest disproof number is selected as successor. Finally, when a leaf node is reached, it is expanded and its children are evaluated. This is called immediate evaluation. In the naive implementation (See Fig. 1.16), proof and disproof numbers are each initialised to unity in the unknown leaves. In other implementations, the proof number and disproof number are set to 1 and n for an OR node (and the reverse for an AND node), where n is the number of legal moves. A disadvantage of PN search is that the whole search tree has to be stored in memory. When the memory is full, the search process has to be terminated prematurely.

The algorithms such as AlphaBeta, A* and Proof Number Search are inefficient in the game of Go because of the number of states and the difficulty to implement an efficient evaluation function.

1.4.11 Iterative deepening depth-first search

For speaking about Iterative deepening depth-first search (IDDFS), we present 2 different search strategies: Depth-first strategy (DFS) and Breadth-first strategy (BFS) and then we argue that IDDFS is a compromise between

the two.

Depth-first search strategy

Depth-first search (DFS) [Korf, 1985] is an algorithm for traversing or searching a tree or more generally graph. One starts at the root (selecting some node as the root in the graph case) and explores as far as possible along each branch before backtracking. In Fig. 1.17, nodes are visited in the following order : A B D E C F G.
The advantage of the algorithm is to find rapidly a solution but with little probability to be optimal. However, some troubles should be treated such as the no termination of the DFS strategy (e.g. In the case of a graph, if no limit of the depth has been defined, an infinite loop can be occurred if a sequence of decisions comes back to a state already seen.).

Breadth-first search strategy

Breadth-first search (BFS) [Korf, 1985] is a strategy for searching in a tree or more generally graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph; (b) gain access to visit the nodes that neighbor the currently visited node. One starts at a root node and inspects all the neighboring nodes. Then for each of those neighbor nodes in turn, it inspects their neighbor nodes which were unvisited, and so on. In Fig. 1.17, nodes are visited in the following order : A B C D E F G. The advantage of the algorithm is to find an optimal solution but can take time if the solutions are at a large depth.

Iterative deepening depth-first search strategy

Iterative deepening depth-first search [Korf, 1985] is a state space search strategy in which a depth-limited search is run repeatedly, increasing the depth limit with each iteration until it reaches the depth of the shallowest goal state. The design of IDDFS is a compromise between Depth-first search and Breadth-first search. IDDFS has the drawback that it explores the same nodes multiple times and an advantage is that the nodes do not have to be stored in memory.

The strategy IDDFS applied to Alpha-Beta solves the problem of the choice of the parameter *depth* for Alpha-Beta by incrementing progressively the depth. From our problem (Fig. 1.5), if we extend the tree of considered moves on Fig. 1.11 to higher depths (for example 10), AlphaBeta with depth = 10 will explore a lot of nodes following the branch beginning by *Qe5+* before studying the optimal branch given by *Qa7+*. Although IDDFS will

Algorithm 4 An implementation of Iterative Deepening Depth-first Search.

Function $iddfs(root, goal)$
$depth = 0$
while $true$ **do**
  $result = dls(root, goal, depth)$
  if $result$ is a solution **then**
    Return $result$
  $depth + +$
Function $dls(node, goal, depth)$
if $depth \geq 0 \wedge node == goal$ **then**
  Return $node$
else
  if $depth > 0$ **then**
    for each child in expand(node) **do**
      $dls(child, goal, depth - 1)$
  else
    Return no solution

visit the root node and some nodes more times than AlphaBeta with $depth = 10$, IDDFS will visit fewer nodes. Some nodes are visited more times because after a new increment of the depth, the algorithm starts at the root again. Because of the mate in 2 moves, Alpha-Beta with an IDDFS strategy will stop when the intern variable $depth$ in the function $iddfs$ is equals to 3 (instead of 10) and will not explore the branch given by Qe5+ until a depth of 10.

Proof-Number Search can be implemented with an iterative deepening depth first strategy such as PN* [Seo et al., 2001].

1.4.12 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a recent Tree Search algorithm [Coulom, 2006] for finding optimal decisions by taking random decisions in the decision space and building a tree search according to the result. In MCTS, the tree is built incrementally and an asymmetric manner by doing Monte-Carlo simulations (Fig. 1.18). Each simulation starts at the root node until a leaf is reached following a rule (which we can call policy of the tree). In 2 player game, the most common rule is $argmax_{d \in D(s)} ucb(d)$ using the

Upper Confidence Bound (UCB) formula:

$$ucb(d) = \frac{Wins(s,d)}{Sims(s,d)} + \sqrt{2 \times \frac{log(Sims(s))}{Sims(s,d))}} \qquad (1.4)$$

where s is the state with which is bound a node of the tree and d a decision in the set of possible decisions $D(s)$. Monte-Carlo Tree Search with UCB formula is called UCT [Kocsis and Szepesvari, 2006] and UCB is a bandit formula. The first term $Wins(s,d)/Sims(s,d)$ which is the win rate is the exploitation and the second term $\sqrt{2 \times \frac{log(Sims(s))}{Sims(s,d))}}$ is the exploration. More generally, the exploration term is weighted by a constant and is the compromise between exploration and exploitation.

In general, when a leaf is reached, a simulation called the default policy is launched, the first state met in the default policy is added in the tree. In our illustration (Fig. 1.19), the whole simulation is kept in the tree. But for other problems, some kinds of troubles such as storage will be met. That's why more commonly, after each simulation, only one node is added in the tree.

Because of its robustness, the policy is in general the move which has been the most simulated and not the move with the best score or a score such as the win rate. Between a first decision which has been simulated 1,000,000 times with a win rate of 70% and a second decision with a win rate of 90% simulated only 10 times, it is better to take the first decision because of accuracy.

A state of the art of MCTS can be found at [Browne et al., 2012]. Monte-Carlo Tree Search is the state of the art for the Game of Go.

1.4.13 Summary

In this thesis, we have discussed below a long list of algorithms; other algorithms nonetheless exist. There are other algorithms such as Fictitious Play [Brown, 1951] or Nested Monte-Carlo Search (NMCS) [Cazenave, 2009] given in Alg. 5 which can be extended to tree searches such as Nested Rollout Policy Adaptation [Rosin, 2011] [Cazenave and Teytaud, 2012] or Nested Monte-Carlo Tree Search [Baier and Winands, 2012]. But they can not be applied to the game of Go or MASH. For example, NMCS is an algorithm for one player game and requires a model of the application.

Most of algorithms can not be applied in our context because

- on MASH, applications are model-free

Algorithm 5 Nested Monte-Carlo Search.

```
Function nested(level, node)
if level == 0 then
   ply = 0
   seq = {}
   while num_children(node) > 0 do
      CHOOSE seq[ply] = child i with probability
      1/num_children(node)
      node = child(node, seq[ply])
      ply + +
   Return (score(node), seq)
else
   ply = 0
   seq = {}
   best_score = +∞
   while num_children(node) > 0 do
      for child i of node do
         temp = child(node, i)
         (results, new) = nested(level − 1, temp)
         if results < best_score then
            best_score = results
            seq[ply] = i
            seq[ply + 1...] = new
      node = child(node, seq[ply])
      ply + +
   Return (best_score, seq)
```

- on the game of Go, modeling the principle of minmax is important in 2-player adversarial game and/or the complexity of the problem is too hard for classical algorithms.

So, the 2 interesting algorithms are the recent algorithm called Monte-Carlo Tree Search in order to implement a strong engine of Go and Direct Policy Search for solving MASH applications. In this thesis, we will bring 4 answers for solving tasks under the 4 constraints seen in Section 1.3.

- In a first part, we will see 2 online learning of Simulations (Monte-Carlo Tree Search) which correspond to the 2 first answers:
  - Simulating with the compromise between exploration/exploitation (classical MCTS)
  - Reducing the complexity of the problem by local searches (GoldenEye, a mix between A* and MCTS)
- In a second part, we will see 2 offline learning of Simulations (DPS) which correspond to the 2 last answers:
  - Building a Policy by genetic programming (RBGP)
  - Learning prior knowledge (CluVo+GMCTS)

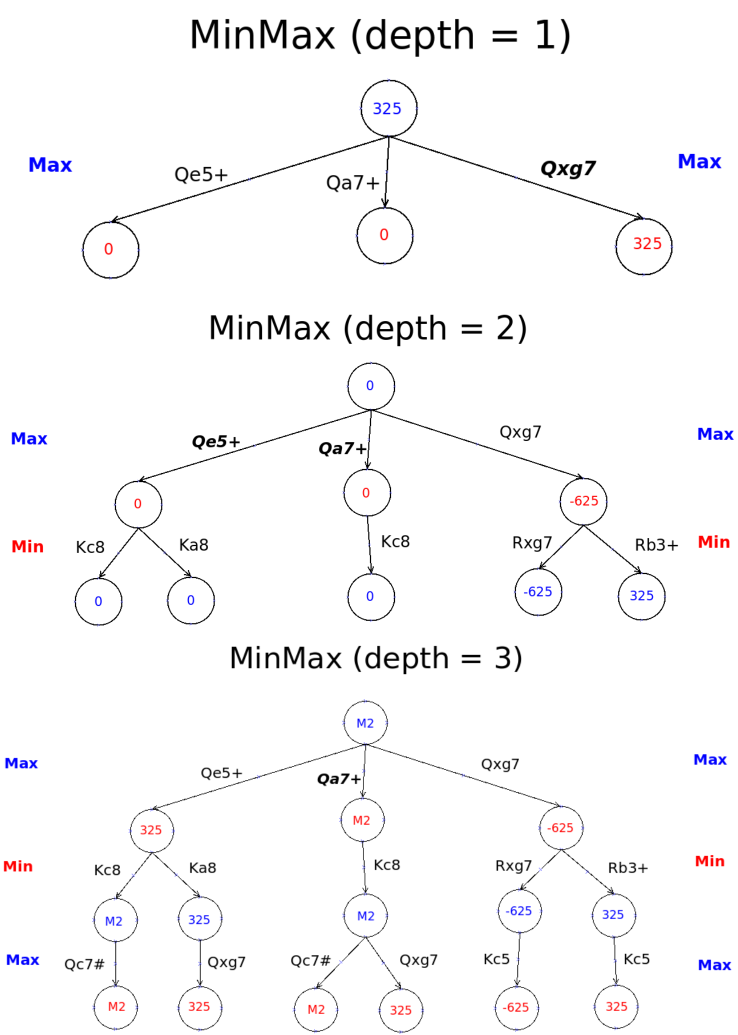


Figure 1.12: MinMax following different depths. MinMax visits all nodes. For each of depths, in **bold**, the move chosen by MinMax.

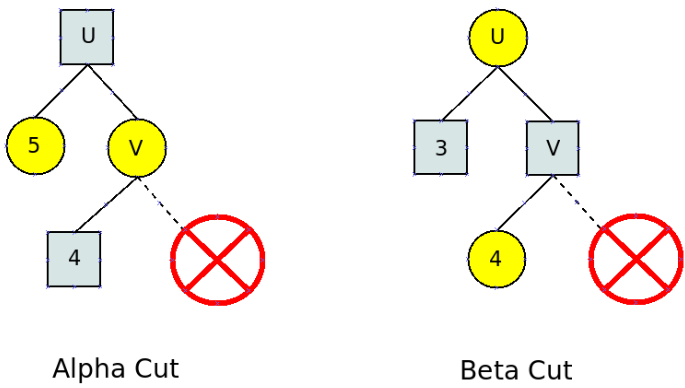


Figure 1.13: From http://fr.wikipedia.org/wiki/Elagage_alpha-beta. The 2 cut-offs of Alpha-Beta. Computing the value in the subtree modeled by the circle within a cross is useless. **Left:** An alpha cut-off. The value of the first child of the Min node V is 4. The value of the Max node U will be of 5 (maximum between 5 and a value smaller than 4). **Right:** A beta cut-off. The value of the first child of the Max node V is 4. The value of the Min node U will be of 3 (minimum between 3 and a value greater than 4).

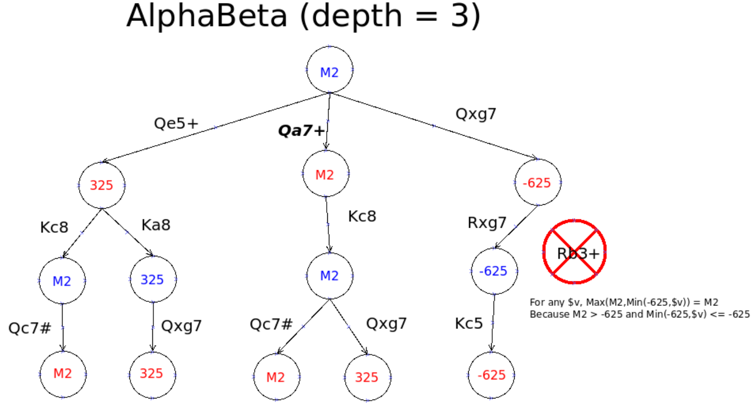


Figure 1.14: Nodes visited by Alpha-Beta. One alpha cut-off.

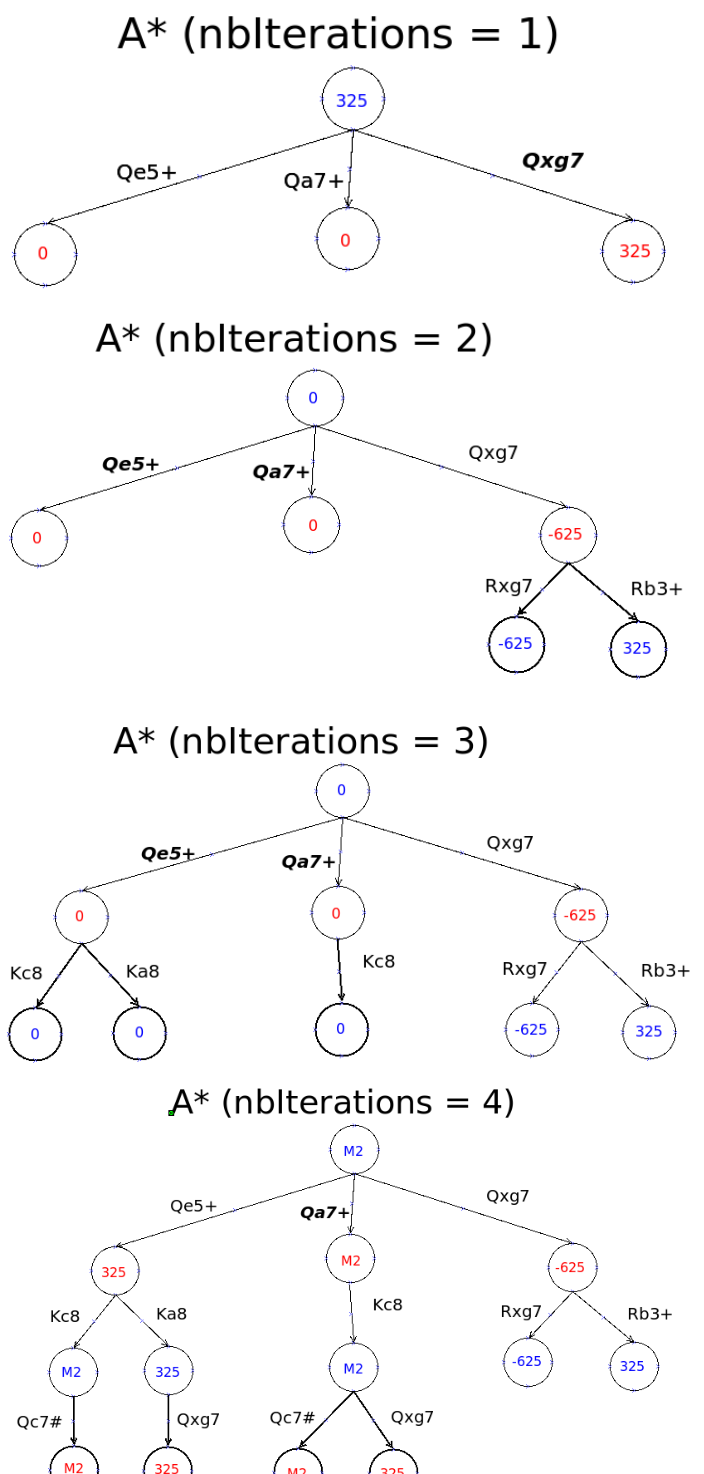


Figure 1.15: Nodes visited by an algorithm of A^* family.

Figure 1.16: Nodes visited by Proof Number Search.

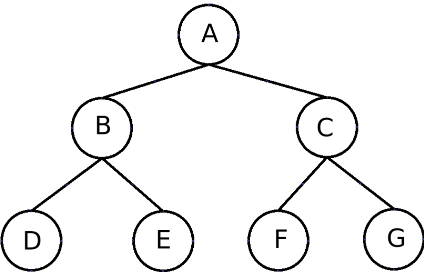


Figure 1.17: From wikipedia. A tree whose nodes are labelled in order to illustrate the route followed by different strategies.

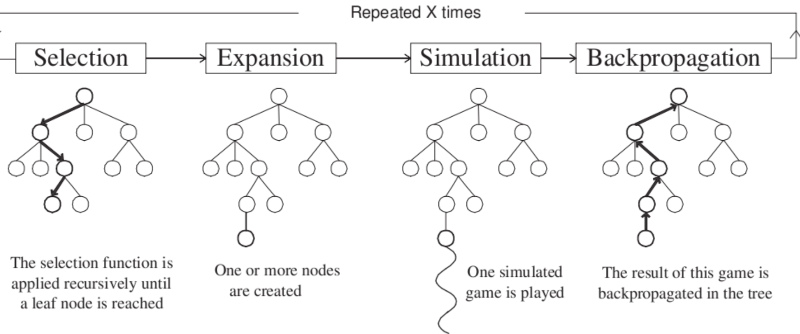


Figure 1.18: The basic MCTS process.

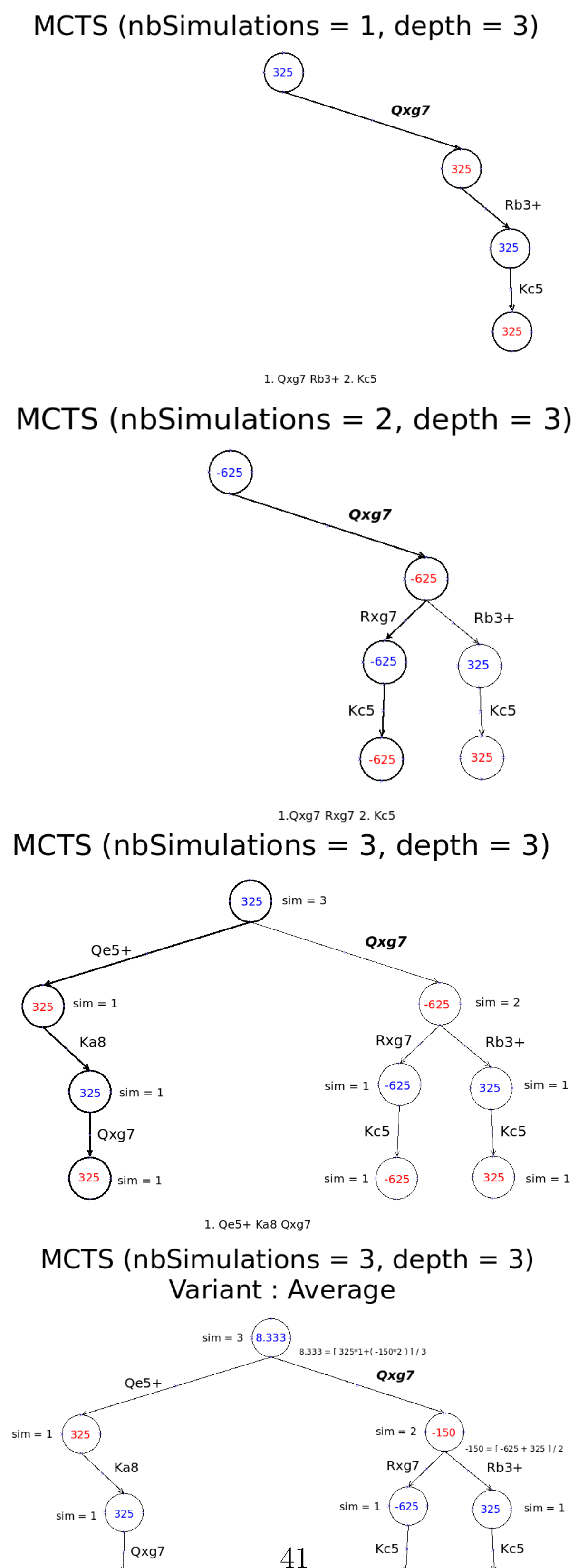


Figure 1.19: Monte-Carlo Tree Search. Right (or left) of a node, *sim* is the number of times that a node has been visited. The parameter *depth* is not a parameter of MCTS; it has been defined here for needs of the illustration.

Part I

Online learning of Simulations : Monte-Carlo Tree Search

Chapter 2

Simulating with the Exploration/Exploitation compromise (Monte-Carlo Tree Search)

This part is heavily based on Scalability and Parallelization in Monte-Carlo Tree Search (CG 2010) [Bourki et al., 2010].

2.1 Introduction

Since 2006, Monte-Carlo Tree Search (MCTS[Chaslot et al., 2006, Coulom, 2006, Kocsis and Szepesvari, 2006]) is a revolution in games and planning, with applications in many fields. It is widely said that MCTS has some scalability advantages.

It is quite natural, then, to parallelize MCTS, both on multicore machines [Wang and Gelly, 2007] and on clusters [Gelly et al., 2008, Cazenave and Jouandeau, 2007]. In this chapter, after an introduction to MCTS (Section 2.2), we (i) discuss the scalability of MCTS, showing big limitations to this scalability, and not only due to RAVE (Section 2.3); (ii) compare existing algorithms on clusters (Section 2.4).

2.2 Monte-Carlo Tree Search

We below introduce Monte-Carlo Tree Search, i.e. MCTS. MCTS is the state of the Art of the game of Go; MCTS was a revolution in 2007. All best

go engines such as MoGo, Fuego, CrazyStone, Zen use MCTS. Even some historic programs such as ManyFaces have alike adopted MCTS. MoGo is the first program[1] who has won with

- 9 handicap stones in 19x19 against a 8p player (Myungwan Kim at 2008 US Go Congress, 2008-08-07)
- 7 handicap stones in 19x19 against a professional player with the highest[2] rank (9p) (Chun-Hsun Chou at Taiwan Open 2009, 2009-02)
- 6 handicap stones in 19x19 against a 1p player (Li-Chen Chien at Taiwan Open 2009, 2009-02)

MCTS has been applied in other two player game such as Shogi [Sato et al., 2010] or connection game such as Havannah [Teytaud and Teytaud, 2009], in single-player game such as SameGame [Schadd et al., 2008], in General Game Playing [Méhat and Cazenave, 2010], in Real-Time Game (e.g. Pacman [Samothrakis et al., 2011], in partially observable game such as Poker [Ponsen et al., 2010] or UrbanRivals [Teytaud and Flory, 2011]. MCTS has been used in non-game, too, such as [de Mesmay et al., 2009] or

- Combinatorial Optimisation (e.g. in Security [Tanabe et al., 2009])
- Constraint Satisfaction [Baba et al., 2011]
- Scheduling Problems [Silver and Veness, 2010]

We here present the MCTS variant termed UCT [Kocsis and Szepesvari, 2006], which is shorter to present and very general; the formulas involved in our programs are more tricky and can be found in [Gelly and Silver, 2007, Lee et al., 2009, Gelly et al., 2008, Teytaud and Teytaud, 2009]; these details do not affect the parallelization, and UCT is a trustable algorithm in the general case of games and planning.

UCT is presented in Alg. 6. The reader is referred to [Kocsis and Szepesvari, 2006] for a more detailed presentation, and to [Gelly and Silver, 2007, Wang and Gelly, 2007, Coulom, 2006, Chaslot et al., 2007] for a more comprehensive introduction in particular for the specific case of binary rewards and two-player games.

[1] http://www.computer-go.info/h-c/index.html.

[2] CrazyStone is the first program who has won against a professional player with H7 (Kaori Aoba -4p- at UEC Cup, 2008-12-14).

Algorithm 6 Overview of the UCT algorithm for two-player deterministic games. The adaptation to stochastic cases or one-player games is straightforward. UCT takes as input a situation $s \in S$, and outputs a decision. For any situation s and any decision d, $s' = s.d$ denotes the situation s' subsequent to decision d in situation s. T is made of two mappings (initially identically 0), N_T and S_T: N_T is a mapping from S to $\mathbb{N}$ (i.e. maps situations to integers) and S_T is a mapping from S to $\mathbb{R}$. S is the set of states, S_T stands for the sum of rewards at a given state and N_T stands for the number of visits at a given state. Inspired by [Coulom, 2006, Wang et al., 2008], we propose $PW(n) = Kn^{1/4}$.

Function $UCT(s)$
$T \leftarrow 0$
while TimeLeft>0 **do**
  $PerformSimulation(T, s)$
Return r maximizing $N_T(s.r)$

Function $reward = PerformSimulation(T, s)$
if s is a final state **then**
  return the reward of s
else
  if $N_T(s) > 0$ **then**
    Choose the set of admissible decisions thanks to progressive widening PW **and the heuristic** H **as follows:**
    $R = PW(N_T(s))$ // $R \in \mathbb{N}$ is the size of the considered pool of moves
    $W = \{H(s,i); i \in [[1, R]]\}$ // W is the considered pool of moves
    Choose the move to be simulated as follows:
    if Color(s)=myColor **then**
      $\epsilon = 1$
    else
      $\epsilon = -1$
    $d = \arg\max_{d \in W} Score(\epsilon.S_T(s.d), N_T(s.d), N_T(s))$
  else
    $d = MC(d)$ /* $MC(d)$ is a heuristic choice of move */
$reward = PerformSimulation(T, s.d)$ // reward$\in \{0, 1\}$
Update the statistics in the tree as follows:
$N_T(s) \leftarrow N_T(s) + 1$
$S_T(s) \leftarrow S_T(s) + reward$
Return $reward$

Function $Score(a, b, c)$
Return $a/b + \sqrt{2\log(c)/b}$ /* plenty of improvements are published in the literature for specific problems*/

Function $H(s, i)$
Return the ith best move according to the heuristic in situation s.

A formula involved in our program is the score. The score for a decision d (i.e. a legal move) is as follows:

$$score(d) = \alpha \underbrace{\hat{p}(d)}_{Online} + \beta \underbrace{\widehat{\hat{p}}(d)}_{Transient} + (\gamma + \frac{C}{\log(2 + n(d))}) \underbrace{H(d)}_{Offline} \tag{2.1}$$

where the coefficients α, β, γ and C are empirically tuned coefficients depending on $n(d)$ (number of simulations of the decision d) and n (number of simulations of the current board).

The logarithmic formula $C/\log(2 + n(d))$ is a progressive unpruning.

The main idea of this formula is: (i) initially, the most important part is the offline learning; (ii) later, the most important part is the transient learning (RAVE values); (iii) eventually, only the "real" statistics (online values) matter.

RAVE values are defined by the following formula given in [Gelly and Silver, 2007]. Let $Qrave(s, d)$ be the rapid value estimate for decision d in state s. After each episode $s_1, d_1, s_2, d_2, ..., s_T$, the action values are updated for every state $s_t \in S$ and every subsequent decision d_{t_2} such that $d_{t_2} \in D(s_{t_1})$ the set of legal decisions following the state s_{t_1}, $t_1 \leq t_2$ and $\forall t < t_2$, $d_t \neq d_{t_2}$.

$$\begin{aligned} n(s_{t_1}, d_{t_2}) &\leftarrow n(s_{t_1}, d_{t_2}) + 1 \\ Qrave(s_{t_1}, d_{t_2}) &\leftarrow Qrave(s_{t_1}, d_{t_2}) + \frac{1}{n(s_{t1}, d_{t_2})}[R_{t_1} - Qrave(s_{t_1}, d_{t_2})] \end{aligned} \tag{2.2}$$

where $n(s, d)$ counts the number of times that decision d has been selected at any time following state s and R_{t_1} the reward given to the agent at the time t_1.

In addition to the parallelization of Monte-Carlo Tree Search [Gelly et al., 2008], some improvements of the Monte-Carlo simulations (e.g. Approach Moves in [Chaslot et al., 2009]) and the building of an opening book by using MCTS in offline mode [Audouard et al., 2009] [Gaudel et al., 2010], some of my main contribution is to have added the term $C/\log(2 + n(d))$ in Eq. 2.1, tuned the coefficient C and introduced some expertises [Chaslot et al., 2009] given in Fig. 2.2. The introductions of these expertises are described below.

$H(d)$ is the sum of two terms: patterns, as in [Bouzy and Chaslot, 2005, Chaslot et al., 2007, Coulom, 2007], and rules detailed below:

- capture moves (in particular, string contiguous to a new string in atari), extension (in particular out of a ladder), avoiding self-atari, atari (in particular when there is a ko), distance to border (optimum distance =

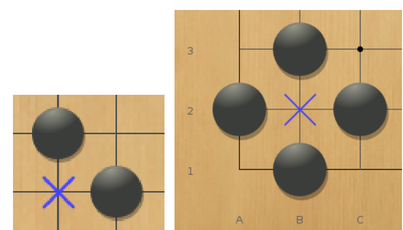


Figure 2.1: **Left:** A move producing an empty triangle for black. **Right:** A corner situation of Go. Let $w_{triangle}$ the coefficient associated with the empty triangle. If we considered the black move at the blue cross B2, the empty triangle is matched 4 times at this location; $w_{triangle}$ will be added 4 times in $H(B2)$. Typically, the empty triangle is a bad shape; in MoGo, $w_{triangle} = -1$.

3 in 19x19 Go), short distance to previous moves, short distance to the move before the previous move; also, locations which have probability nearly 1/3 of being of one's color at the end of the game are preferred.

The following rules are used in our implementation in 19x19, and improve the results:

- Territory line (i.e. line number 3), Line of death (i.e. first line), Peep-connect (ie. connect two strings when the opponent threatens to cut), Hane (a move which "reaches around" one or more of the opponent's stones), Threat, Connect, Wall, Bad Kogeima (same pattern as a knight's move in chess), Empty triangle (three stones making a triangle without any surrounding opponent's stone).

A coefficient is associated with each rule or pattern. The coefficient is typically between 0 and 1 for a good shape or between -1 and 0 for a bad shape. Sometimes, the coefficient can be greater/smaller than 1/-1 for very good/bad shape. Fig. 2.1 shows a pattern matching.

They are used both (i) as an initial number of RAVE simulations (ii) as an additive term in H. The additive term (ii) is proportional to the number of AMAF-simulations (AMAF = All Moves As First, also termed RAVE for Rapid Action-Value Estimates in the MCTS context).

These shapes are illustrated on Fig. 2.2. With a naive hand tuning of parameters, only for the simulations added in the AMAF statistics, they provide 63.9±0.5 % of winning rate against the version without these improvements. Parameters have been then automatically tuned in [Chaslot et al., 2009].

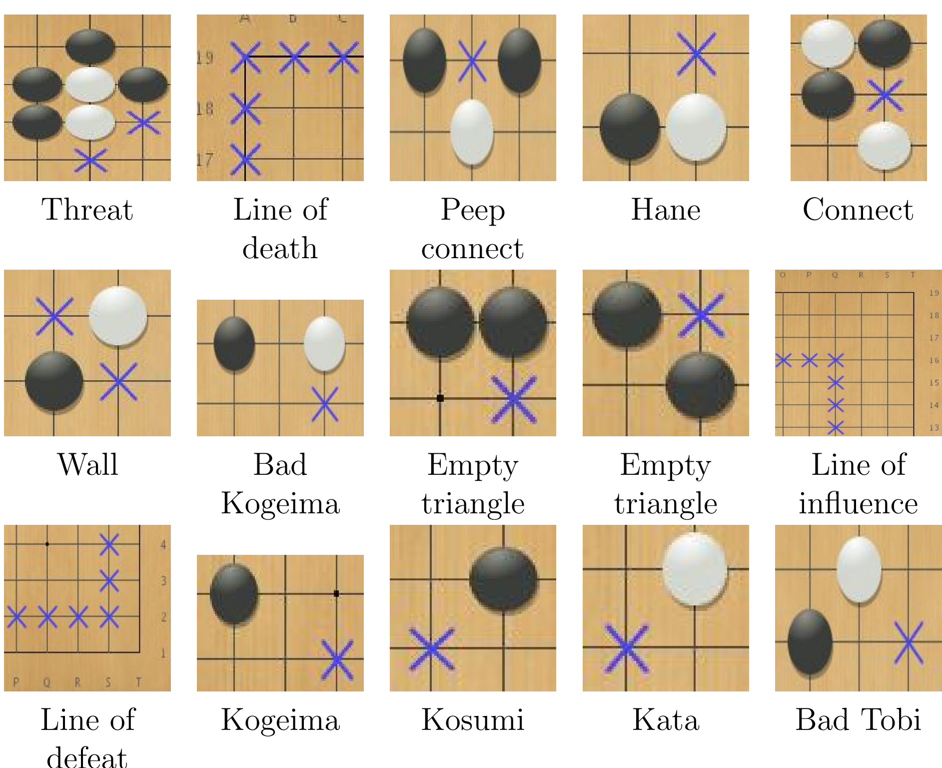


Figure 2.2: We here present shapes for which exact matches are required for applying the bonus/malus. In all cases, the shapes are presented for the black player: the feature applies for a black move at one of the crosses. The reverse pattern of course applies for white. Threat is not an exact shape to be matched but just an example: in general, black has a bonus for simulating one of the liberties of an enemy string with exactly two liberties, i.e. to generate an atari.

N =Number of simulations	Succes rate of 2N simulations against N simulations in 9x9 Go	Success rate of 2N simulations against N simulations in 19x19 Go
1 000	71.1 $\pm$ 0.1 %	90.5 $\pm$ 0.3 %
4 000	68.7 $\pm$ 0.2	84.5 $\pm$ 0.3 %
16 000	66.5 $\pm$ 0.9 %	80.2 $\pm$ 0.4 %
256 000	61.0 $\pm$ 0.2 %	58.5 $\pm$ 1.7 %

Table 2.1: Scalability of MCTS for the game of Go. These results show a decrease of scalability as computational power increases.

2.3 Scalability of MCTS

The scalability of MCTS, i.e. its ability to play better when additional computational power or time is provided, is often given as an argument in favor of it. Also, it is said that the parallelization is very efficient; the conclusion of these two statements is that with big clusters, programs should now be much stronger than humans in games in which single computers are already at the level of beginners. We will here give more informations (limitations) on this scalability.

The number of simulations per move is usually much larger in real games than in experimental results published in papers, because of limited computational power - it's difficult, even with a cluster, to have significant results corresponding to the computational power associated to realistic time settings on a big machine. In this section, we investigate the behavior of MCTS when the time per move is increased (Section 2.3.1), followed by counter-examples to scalability (Section 2.3.2).

2.3.1 The limited scalability by numbers

It is usually said that MCTS is highly scalable, and provides improvements of constant order against the baseline when the computational power is doubled. We here show that things are not so constant; results are presented in Tab. 2.1 for the game of Go. These numbers show the clear decrease of scalability as the computational power increases. This is not specific to Go; Tab. 2.2 shows that the situation is similar in Havannah. This holds even when the opponent is a MCTS also; this is not equivalent to the case of the scalability study `http://cgos.boardspace.net/study/index.html` which considers non-MCTS opponents as well; we here see that just against the same MCTS program, we have a limit in scalability; this even happens in 19x19. In Havannah with slow simulations (the operational case, with the

Number of fast simulations	Success rate
100 vs 50	68.6 ± 0.68%
1000 vs 500	63.57 ± 0.76%
2000 vs 1000	59.0 ± 1.0%
4000 vs 2000	53.9 ± 1.6%
10000 vs 5000	55.2 ± 1.6%
20000 vs 10000	54.89 ± 1.25%

Number of slow simulations	Success rate
100 vs 50	63.28 ± 0.4%
1000 vs 500	57.37 ± 0.9%
2000 vs 1000	56.42 ± 1.1%
4000 vs 2000	53.24 ± 1.42%
10000 vs 5000	52 ± 1.6%

Table 2.2: Scaling for the game of Havannah, for fast (left) and slow (right) simulations. As we can see, the success rate is not constant, but decreases when the number of simulations increases.

best performance in practice), 10 000 simulations per move give only 52% winning rate against 5 000 simulations per move (Tab. 2.2). This suggests that the scalability is smaller than expected from small scale experiments. Usually people do not publish experiments with so many simulations because it is quite expensive; nonetheless, real games are played with more than this kind of numbers of simulations and the numbers in the tables above are probably greater than the scalability in realistic scenarios.

A particularity of these numbers is that they are in self-play; this provides a limitation even in the ideal case in which we only consider an opponent of the same type; it is widely known that the improvement is much smaller when considering humans or programs of a different type. Interestingly [Kato, 2009] has shown that his MCTS implementation reaches a plateau against GnuGo when the number of simulations goes to infinity. This shows limited scalability, to be confirmed by situations (practically) unsolved by Monte-Carlo Tree Search, presented in section below.

2.3.2 Counter-examples to scalability

Heuristic refers to experience-based techniques for problem solving, learning, and discovery. Where the exhaustive search is impractical, heuristic methods are used to speed up the process of finding a satisfactory solution.

The RAVE heuristic ([Bruegmann, 1993, Gelly and Silver, 2007]) is known to be very efficient in several games: it introduces a bias in H. It is nonetheless suspected that RAVE is responsible for the bad asymptotic behavior of some MCTS programs. We below recall some known counter-examples when RAVE is included, and then give a detailed presentation of other counter-examples which do not depend on RAVE.

Counter-examples based on RAVE values. Martin Müller posted in the computer-Go mailing list the situation shown in Fig. 2.3(left, `http://fuego.svn.sourceforge.net/viewvc/fuego/trunk/regression/sgf/rave-problems/`) in which their MCTS implementation Fuego does not find the good move due to RAVE (discussed in Section 2.3.2), because the only good move is good only if played first (the RAVE value[Gelly and Silver, 2007] does not work in this case) - such cases are clearly moderately sensitive to computational time or computational power, and this has impacts in terms of scalability.

Other counter-examples. Importantly, Fig. 2.3(right) from [Berthier et al., 2010] shows that there are some bad behaviours even without RAVE values. Below, we propose new clear examples of limited speed-up,

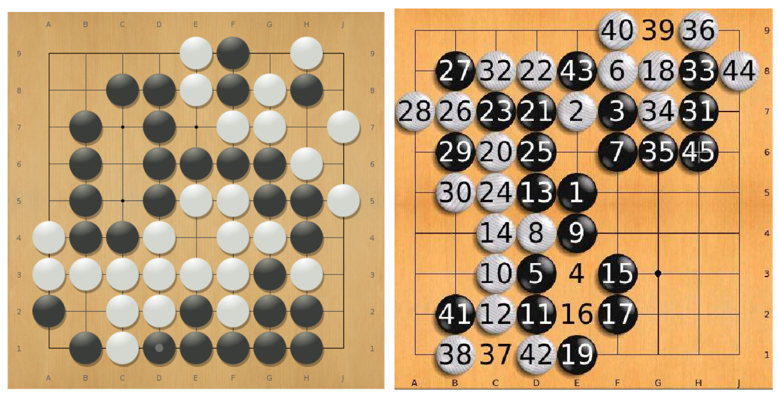


Figure 2.3: Left: white to play, an example by M. Müller of bad scalability due to RAVE. RAVE gives a very bad value to the move B2 (second row, second column), because it only makes sense if it's the first move, whereas this is the only move avoiding the seki (otherwise, black A5 and the two black stones A2 and B1 are alive). Right, white to play: an example of bad behavior shown in [Berthier et al., 2010], independently of Rave values: in many cases (yet not always, this depends on the first simulations), MoGo is almost sure that he is going to win as white by playing C1, whereas it is a loss for white.

that have the following suitable properties:

- These situations are extremely easy for human players. Even a beginner can solve them.
- These counter-examples are independent of RAVE, as shown in our experiments.

Such situations are given in Fig. 2.4. These situations are semeais; it is known since [Coulom, 2009, Lee et al., 2009] that MCTS algorithms are weak in such cases. We show that this weakness remains without RAVE and even with inclusion of specific tactical solvers.

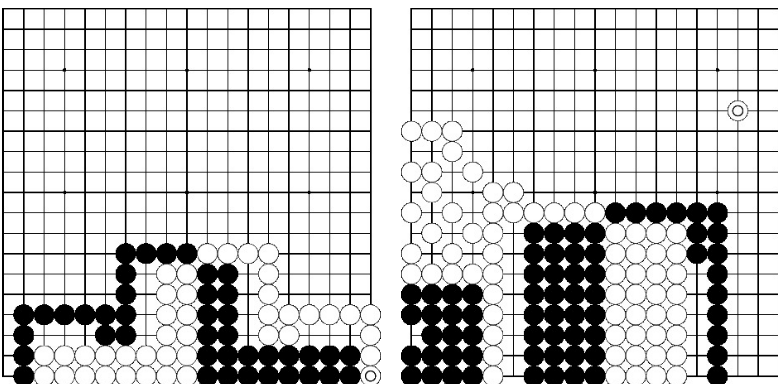


Figure 2.4: Left: black to play. It is here necessary to play in the semeai. Right: black to play: playing in the semeai is useless as the semeai is won anyway (black has two more liberties than white) - good moves are outside the semeai. MoGo often makes the mistake of playing in the semeai.

It is often said that classical solvers are able to solve semeais and therefore including expert modules should improve MCTS algorithms by including semeai solver. We have therefore tested two ways of including expertise in MCTS:

- Expertise: we introduce a bias in the score, as usually performed in MCTS algorithms [Chaslot et al., 2006, Coulom, 2006, Lee et al., 2009]. Some virtual wins are added to UCT statistics so that moves which are good according to our tactical semeai solver called GoldenEye (Chapter 3) are more simulated; the idea, detailed in [Chaslot et al., 2006, Coulom, 2006, Lee et al., 2009] consists in increasing the score of moves evaluated as necessary by the semeai solver, so that the heuristic H is more favorable to them. Only moves necessary for solving the semeai are given a bonus; no move at all if the semeai is won even if the player to play passes.
- Conditioning: then, all simulations which are not consistent with the solver are discarded and replayed. This means that when the solver predicts that the semeai is won for black (the solver is called at the end of the tree part, before the MC part), before the Monte-Carlo (MC) part, then the Monte-Carlo simulation is replayed until it gives

a result consistent with this prediction. Human experts could validate the results (i.e. only simulations consistent with the semeai solver were included in the Monte-Carlo) and the quality of the solver is not the cause for results in Tab. 2.3; the coefficients have been tuned in order to be a minimum perturbation for having a correct solving for Fig. 2.4, left: the coefficients are (i) the size of semeais considered (ii) the weight of the expertise in the function H (for versions with expertise).

The results are presented in Tab. 2.3. In order to be implementation-independent, we consider the performance for fixed numbers of simulations; the slowness of the tactical solver can't be an explanation for poor results. From these negative results, and also for many trials with various tunings, all of them leading to success rates lower than 50 % against the baseline, we include that including expert knowledge is very difficult for semeais; it is true that tactical solvers can solve semeais, but they do not solve the impact of semeais on the rest of the board: in conditioning, if simulations are accepted as soon as they are consistent with the semeai solver, then the result of the semeai will be understood by the program but the program might consider cases in which black played two more stones than necessary - this is certainly not a good solving of the semeai.

These examples of bad behavior are not restricted to MoGo. Fig. 2.5 is a game played by Fuego and Aya in the 56th KGS tournament (february 2010); Fuego (a very strong program by Univ. Alberta) played (1) and lost the game.

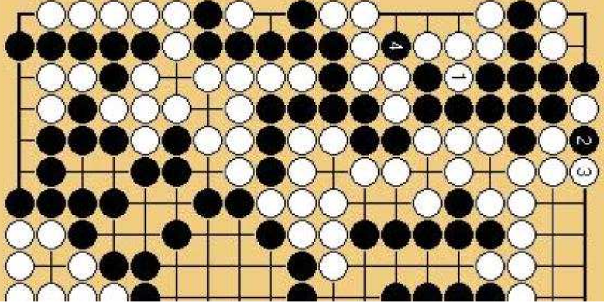


Figure 2.5: Fuego as white played the very bad move (1) during the 56th KGS tournament and lost the game. This is an example of situation very poorly handled by computers.

Version of the algorithm	Percentage of "good" moves
Situation in which the semeai should be played 1K sims per move	
MoGo	32 %
MoGo with expertise	**79 %**
MoGo with conditioning	24 %
MoGo with exp.+condit.	**84 %**
Situation in which the semeai should not be played 1K sims per move / 30K sims per move	
MoGo	**100%** / 58 %
MoGo with expertise	**95 %** / 51 %
MoGo with conditioning	**93 %** / 0 %
MoGo with exp.+condit.	**93 %** / 54 %

Table 2.3: These results are for Fig. 2.4; black should or should not play in the semeai (left or right situation in Fig. 2.4). All results are averaged over 1000+ runs. Bold is for results with more than 75 % on correct moves. We point out that the Go situations under consideration are very easy, understandable by very beginners. We see that (i) with 30K sims/move, many versions play the semeai whenever it is useless, and all versions play it with significant probability, which is a disaster since in real situations there are many time steps at which the MCTS program can have the opportunity of such a bad move and even only one such move is a disaster because it is completely wasted (ii) removing RAVE does not solve the problem (iii) adding a tactical solver can work better (moderately better) with the traditional solution of adding expertise as virtual wins, but results remain very moderate, and far from what can do even a beginner. We also tested many parameterizations in self-play and none of these tests provided more than 50 % of success rate in self-play.

2.4 Message-passing parallelization

Multi-core machines are more and more efficient, but the bandwidth is nonetheless limited, and the number of cores is much bigger when we consider clusters than when we consider a single machine. This is why message-passing parallelization (in which communications are explicit and limited) must be considered. We'll see here that, in particular in 19x19, the technique is quite efficient from a parallelization point of view: the main issue for MCTS is not the computational power, but the limits to scalability emphasized in Section 2.3.

The various published techniques for the parallelization of MCTS are as

follows:

- *Fast tree parallelization* consists in simulating the multi-core process on a cluster; there is still only one tree in memory, on the master, and slaves (i) compute the Monte-Carlo part (ii) send the results to the master for updates. This is sensitive to Amdahl's law, and is quite expensive in terms of communication when RAVE values are used[Hill and Marty, 2008, Gelly et al., 2008].

- *Slow tree parallelization* consists in having one tree on each computation node, and to synchronize these trees slowly, i.e. not at each simulation but with frequency e.g. three times per second [Gelly et al., 2008]. The synchronization is not on the whole tree; it is typically performed as follows:

  - Select all the nodes with
    - at least 5% of the total number of simulations of the root;
    - depth at most d (e.g. $d = 3$);
  - Average the number of wins and the number of simulations for each of these nodes.

  This can be computed recursively (from the root), using commands like *MPI_AllReduce* which have a cost logarithmic in the number of nodes. A special case is **slow root parallelization:** this is slow tree parallelization, but with depth at most $d = 0$; this means that only the root is considered.

- *Voting schemes*. This is a special case of tree parallelization advocated in [Chaslot et al., 2008], that we will term here for the sake of comparison with other techniques above **very slow root parallelization:** this is slow root parallelization, but with frequency $f = 1/t$ with t the time per move: the averaging is only performed at the end of the thinking time. There is no communication during the thinking time, and the drawback is that consequently there is no load balancing.

It is usually considered that fast tree parallelization does not perform well; we will consider only other parallelizations. We present in Tab. 2.4 the very good results we have in 19x19 and the moderately good results we have in 9x9 for slow tree parallelization.

We can compare **slow root parallelization to the "voting scheme" very slow root parallelization:** with 40 machines and 2 seconds per move in 9x9 and 19x19, the slow root parallelization wins clearly against the version

Configuration of game	Winning rate in 9x9	Winning rate in 19x19
32 against 1	75.85 ± 2.49 %	95.10±01.37 %
32 against 2	66.30 ± 2.82 %	82.38±02.74 %
32 against 4	62.63 ± 2.88 %	73.49±03.42 %
32 against 8	59.64 ± 2.93 %	63.07±04.23 %
32 against 16	52.00 ± 3.01 %	63.15±05.53 %
32 against 32	48.91 ± 3.00 %	48.00±09.99 %

Table 2.4: Experiments showing the speed-up of "slow-tree parallelization" in 9x9 and 19x19 Go. We see that a plateau is reached somewhere between 8 and 16 machines in 9x9, whereas the improvement is regular in 19x19 and consistent with a linear speed-up - a 63% success rate is equivalent to a speed-up 2, therefore the results sill show a speed-up 2 between 16 and 32 machines in 19x19. Experiments were reproduced with different parameters with strong difference; in this table, the delay between two calls to the "share" functions is 0.05s, and x is set to 5%. The numbers with high numbers of machines will be confirmed in Tab. 2.5.

Framework	Success rate against voting schemes
9x9 Go	63.6 % ± 4.6 %
19x19 Go	94 % ± 3.2 %

Table 2.5: The very good success rate of slow tree parallelization versus very slow tree parallelization. The weakness of voting schemes appears clearly, in particular for the case in which huge speed-ups are possible, namely 19x19.

with very slow root parallelization, as shown by Tab. 2.5. with a frequency 1/0.35 against the very slow root parallelization. As a rule of thumb, it is seemingly good to have a frequency such that at least 6 averagings are performed; 3 per second is a stable solution as games have usually more than 2 seconds per move; with a reasonably cluster 3 times per second is a negligible cost.

We now compare **slow tree parallelization** with depth $d = 1$, to the case $d = 0$ (slow root parallelization) advocated in [Cazenave and Jouandeau, 2007]. Results are as follows and show that $d = 0$ is a not so bad approximation:

Time per move	Winning rate of slow-tree-parallelization (depth=1) against slow-root-parallelization
2	50.1 ± 1.1 %
4	51.4 ± 1.5 %
8	52.3 ± 1 %
16	51.5 ± 1 %

These experiments are performed with 40 machines. The results are significant but very moderate.

2.5 Conclusion

We revisited scalability and parallelism in MCTS.

The scalability of MCTS has often been emphasized as a strength of these methods; we show that when the computation time is already huge, then doubling it has a smaller effect than when it is small. This completes results proposed by Hideki Kato[Kato, 2009] or the scalability study `http://cgos.boardspace.net/study/index.html`; the scalability study was stopped at 524288 simulations, and shows a concave curve for the ELO rating in a framework including different opponents; Hideki's results show a limited efficiency, when computational power goes to infinity, against a non-MCTS algorithm. Seemingly, there are clear limitations to the scalability of MCTS; even with huge computational power, some particular cases can't be solved. We also show that the limited speed-up exists in 19x19 Go as well, and not with much more computational time than in 9x9 Go. In particular, cases involving visual elements (like big yose) and cases involving human sophisticated techniques around liberties (like semeais) are not properly solved by MCTS, as well as situations involving multiple unfinished fights. Our experiments also show that the situation is similar in Havannah with good simulations. The main limitation of MCTS is clearly the bias, and for some situations (as those proposed in Fig. 2.4) introducing a bias in the score formula is not sufficient; even discarding simulations which are not consistent with a tactical solver is not efficient for semeai situations or situations in which liberty counting is crucial.

Several parallelizations of MCTS on clusters have been proposed. We clearly show that communications during the thinking time are necessary for optimal performance; voting schemes ("very" slow root parallelization) don't perform so well. In particular, slow tree parallelization wins with probability 94 % against very slow root parallelization in 19x19, showing that the slow tree parallelization from [Gelly et al., 2008] or the slow root parallelization from [Cazenave and Jouandeau, 2007] are probably the state of the art.

Slow tree parallelization performs only moderately better than slow root parallelization when MCTS is used for choosing a single move, suggesting that slow root parallelization (which is equal to slow tree parallelization simplified to depth= 0) is sufficient in some cases for good speed-up - when MCTS is applied for proposing a strategy (as in e.g. [Audouard et al., 2009] for opening books), tree parallelization naturally becomes much better.

Chapter 3

Reducing the complexity by local solving

The work presented in this chapter has not been already published.

3.1 Introduction

A drawback of MCTS engines of Go is that sometimes, parts of the goban (e.g. corners) are badly evaluated. Some tactics situations are very badly understood by MCTS. The semeai is an example. A semeai (or capturing race) is a mutual capturing contest. This is a tactical situation created in positions when both players have groups striving to capture each other, in some area of the board. Typically it is not possible for each side to create a safe group with two eyes. Fig. 3.1 and Fig. 3.4 show examples.

We propose to solve some capturing races. First, some terms of Go game

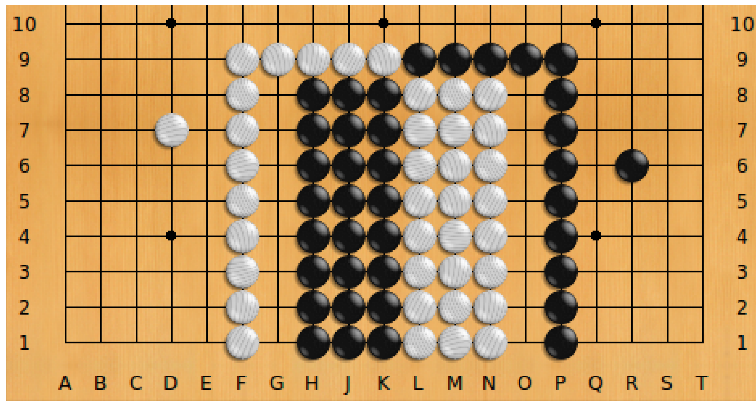


Figure 3.1: A basic semeai.

are explained, then a state of the art is presented and a semeai solving algorithm is given. Finally, some results are shown.

3.2 Glossary of Go terms

We explain some terms of the game of Go such as ko, seki, shicho and tsumego.

Liberty: A liberty of a group G is a free intersection neighbouring the group G.

Atari: A group is in atari if the group has only one liberty. The opponent threatens to capture the group at his next move.

Ko: Players are not allowed to make a move that returns the game to the previous position. This rule, called the ko rule, prevents unending repetition. Fig. 3.2 depicts 2 kinds of ko. The ko makes harder the Go game, because a local situation becomes global. Indeed, a player can not recapture a stone but it can threat something in another place of the goban; the second player has the choice:

- he can answer to the threat but then the first player can retake the ko (the situation has changed)
- or win the ko (e.g. by protecting his stone in atari) but in counter-part, the first player will execute his threat.

Seki: Seki is a Japanese go term which means mutual life. In its simple form, it is a kind of symbiosis where two live groups share liberties which neither of them can fill without dying (Fig. 3.2).

Shicho: Shicho is a technique for capturing a group of stones. The number of liberties of the attacked group alternates between 1 and 2 at each move, until this group has no more liberty and up to be captured. Fig. 3.2 gives an example.

Tsumego: Like semeai, a tsumego (or death/life problem) is a tactical situation met on a part of the goban. The life of a group is in pending. A tsumego concerns the life of one group whereas the semeai is about several groups. Fig. 3.2 and Fig. 3.25 give examples.

Approach move: A move that must be played before taking an opponent liberty, because doing so directly would put us in atari or leave us with a critical weakness (Right in Fig. 3.12). In effect, an approach move increases the actual number of liberties of the opponent group.

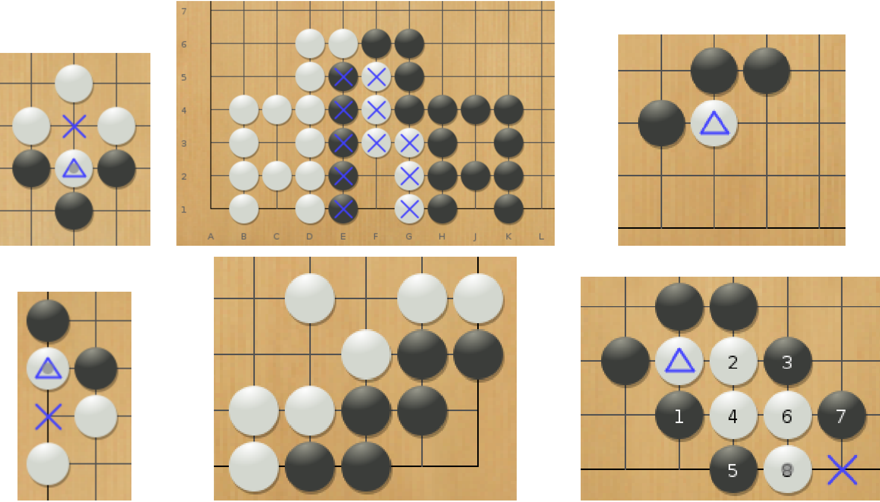


Figure 3.2: **Left:** The ko rule. Black to Play. In the game of Go, it is forbidden to come back to positions already seen. The last move (the white stone with a blue triangle) has captured a black stone in the blue cross intersection. Black can not play into the blue cross intersection because black captures the white stone with a blue triangle and we come back to a precedently met situation. *Top Left* / *Bottom Left* depicts a ko respectively in the center / on the border of the goban. **Top Center:** Seki. If Black/White plays F1 (or F2), then White/Black captures the group stones by playing F2 (or F1). The 2 groups E1 and G1 (with cross) are alive by seki. **Right:** Shicho. *Top Right:* Black to Play. The starting of the Shicho. The white triangle stone will be captured. *Bottom Right* shows how. Black plays at the cross and captures the white group. **Bottom Center** is a tsumego. The statement can be either "White to play and kills the black group" or "Black to play and lives".

3.3 State of the art

For solving a capturing race in any position of Go, this situation should be found. Once detected, another algorithm should be able to give the solution. First, we present briefly some techniques of detection and secondly, we present some algorithms for solving capturing races.

3.3.1 Detection of the Semeai

For finding semeai, the most common mechanism is analysis of groups. First, we determine the safety of groups. Unsafe and enclosed groups are searched. A semeai is likely to occur when 2 such groups of both players are neighbours. GNU Go, a software of Go game, uses this technique (See Chapter 12 of GNU Go's documentation - http://gnu.cs.pu.edu.tw/software/gnugo/gnugo_toc.html). In 1999, Martin Mueller proposes a similar method in [Müller, 1999]. But the analysis is prone to error of coding.

3.3.2 Resolution of the Semeai

There are 2 kinds of resolution:

- A static resolution by using analysis and mathematical rules
- A dynamic resolution by tree search.

Without tree

Sometimes, simple semeais can be solved statically by counting liberties. Solvers have been developed by analyzing eye shape [Vilà and Cazenave, 2003], as well as external and common liberties [Nakamura, 2008] [Müller, 2002]. Wolf studies semeais with approach moves [Wolf, 2012]. More involved rules also exist in the literature for human players [Hunter, 2003].

No simulation is necessary; it solves really quickly semeais which can be very big. However, a lot of rules is required and frontiers should be well-defined.

With tree

Solvers given in this section are mainly applied to tsumegos. Algorithms for solving tsumegos and semeais are similar, except they require specific

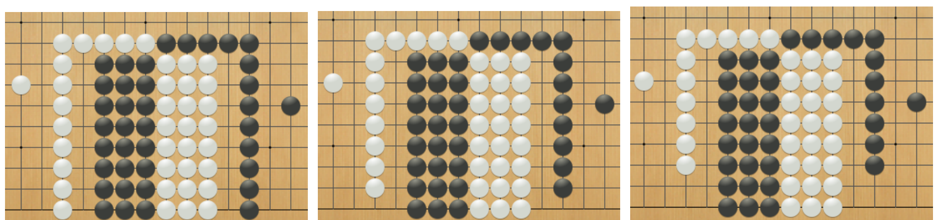


Figure 3.3: **Left:** A closed semeai. **Center:** A little open semeai. **Right:** An open semeai.

knowledge (e.g. the shape of eyes for tsumegos or counting liberties for semeais).

GoTools [Wolf, 1994] implemented by Wolf in 1994 solve tsumegos. The algorithm which solves tsumego can be applied to semeais [Wolf, 1994]. But it is not its primary function. GoTools uses a classical alpha-beta with hash tables.

Proof Number Search is another common algorithm for solving small tactical problems such as tsumego [Saito et al., 2007]. It is used for instance in Explorer.

Explorer is a software of Go with a solver specialized in one-eye tsumego problem [Kishimoto and Müller, 2003]. Explorer uses Depth-First Proof Number Search.

Open problems: A problem of Go is said open if the considered area of the problem is not completely surrounded by stones. In closed problems, solvers will consider all the legal moves in the enclosed area. In open problems, finding the set of moves to be considered is a challenge. Fig. 3.3 depicts open and closed problems.

A first solution could be to add automatically stones for closing the problem. A risk is to change the nature of the problem.

Pruning is another solution. [Chen and Zhang, 2006] solves open capture problems with AlphaBeta and a highly selective heuristic of moves.

[Niu and Müller, 2006] presents a solution for solving open tsumegos, with pruning, and especially with some mechanisms, based on automatic responses, to prevent the search from expanding where the boundary is not perfectly defined. But it is difficult to have an exhaustive list of all these mechanisms. The problem can not be too opened and groups surrounding the problem must be safe.

Abstract Proof Search (APS) [Cazenave, 2000] uses another approach. It is able to generate small sufficient sets of moves by assuming the problem can be won under a given number n of moves. Those move generations rely on

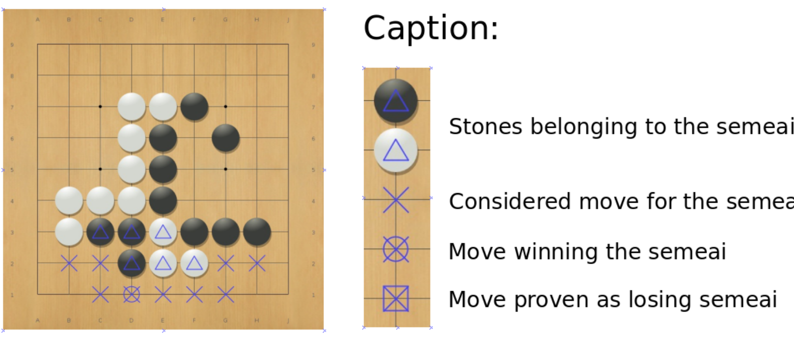


Figure 3.4: Caption for semeais. **Left:** a semeai. **Right:** the caption.

ad hoc functions. The algorithm has been mainly used to solve open capture problems in the game of Go. APS can be applied for solving semeais, but it's not its primary target and these ad hoc functions become complex when n increases.

Another solution consists in gradually expanding the sets of moves. Abstract Proof Search has been extended with an iterative widening approach [Cazenave, 2001] [Cazenave, 2004].

Monte-Carlo Tree Search [Zhang and Chen, 2008] is another way for solving open problems.

3.4 GoldenEye algorithm: combining A* and MCTS

The algorithm called GoldenEye decomposes in two parts (Fig. 3.5) :

- the detection of the semeai
- the resolution of the semeai

3.4.1 Statistical detection

In this section, the method of detection is presented. From some results, parameters of the detection are fixed. Finally, we introduce some ways for improving the detection.

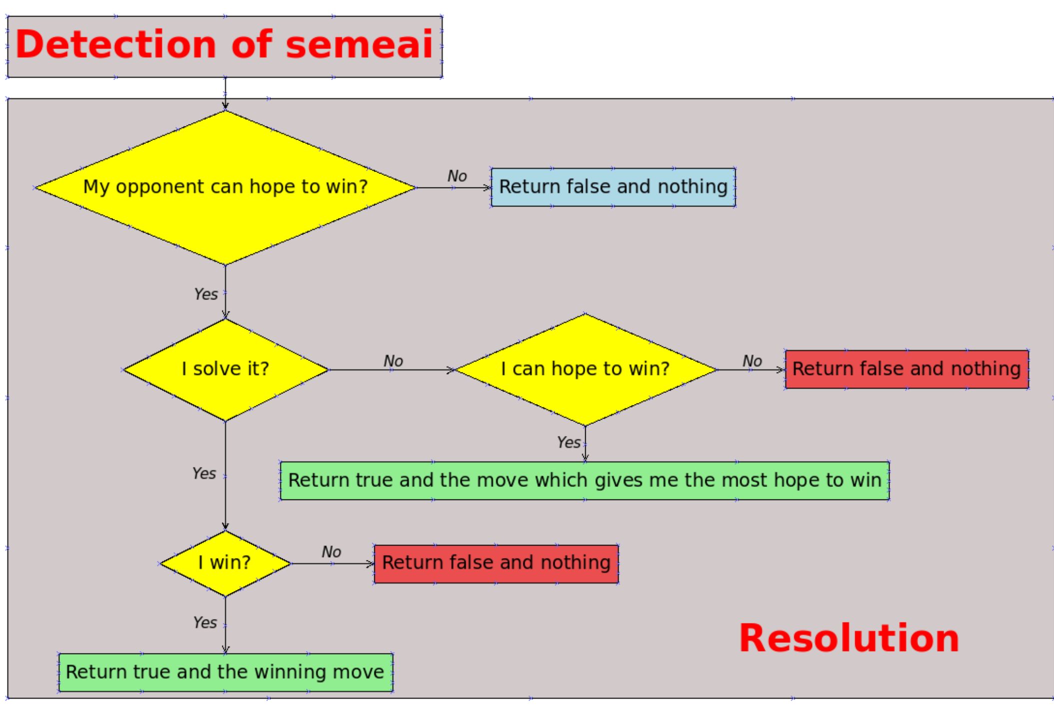


Figure 3.5: Algorithm for really choosing the move. The outputs are blue, green and red rectangles. A yellow diamond-shape is a condition. A blue rectangle means a winning case with nothing to do: the best case. Like blue rectangle, a red rectangle means nothing to do but in the bad case: it is the worst case. A green rectangle means something to do for winning. We say that the opponent can hope a win if (i) either we have proved that it is a win for the opponent if he is the first to play in the semeai; (ii) or we got an estimation $> 50\%$ for the opponent, in case we don't play in the semeai.

The approach

From the current position, pure random simulations are done. Some statistics are collected such as the number of times that a stone is dead whereas another is alive at the end of a simulation.

Let Sb a black stone and Sw a white stone.
$AD_{\text{nbSims}}(s1, s2)$ is defined as the frequency of simulations where s1 is alive and s2 is dead at the end, through $nbSims$ simulations. We define

$$sem_{\text{nbSims}}(Sb, Sw) = AD_{\text{nbSims}}(Sb, Sw) + AD_{\text{nbSims}}(Sw, Sb) \quad (3.1)$$

Two stones Sb and Sw are said to be in semeai if

$$sem_{nbSims}(Sb, Sw) >= \rho \quad (3.2)$$

where $\rho \in [0; 1]$ is a fixed parameter and $nbSims$ the number of random simulations.

For avoiding semeais where the fight is too unequal, a second condition has been added. The condition is

$$\min(AD_{\text{nbSims}}(Sb, Sw), AD_{\text{nbSims}}(Sw, Sb)) > \sigma \quad (3.3)$$

where $\sigma \in [0; 1]$ is a fixed parameter.

Semeais are built by aggregation of stones said to be in semeai.

The aggregation stone by stone[1] works well in practice in the sense that results are consistent with the connexity of groups. It rarely occurs that a stone of a group is marked in semeai but not the whole group (Fig. 3.7 features one example of this situation).

On the computer-go mailing-list, a very similar idea has been exposed by Jonas Kahn on January 13, 2011: "*[...] A semeai is then two groups of stones with anti-correlation P((white stones live and black stones live) OR (white stones die and black stones die)) almost 0, whereas P(B lives) and P(W lives) both nonzero and non-one.*" The anti-correlation and the function *sem* are both indicators of the existence of a semeai and they are almost equivalent.

The detection of semeais restricts candidate area(s) of the goban where the engine should focus. Some works using statistics such as [Coulom, 2009] are another mean for finding these areas. [Coulom, 2009] is moreover a tool for deciding the importance of a semeai and/or may detect some candidate areas in which a semeai could be found.

[1] instead of group by group.

Choosing parameters

3 parameters are used in the detection:

1. the number of random simulations $nbSims$
2. the threshold ρ
3. the threshold σ.

After having observed statistical data (such as the percentage of life of a stone in situations with or without semeai) from random simulations, σ has been fixed to 0.14.

After some experiments such as in Fig. 3.6 and Fig. 3.7, we notice that 100 random simulations is enough in most cases for a correct detection of semeai. In a complex situation such as Fig. 3.7, running more simulations is useful because it stabilizes the detection. However, it is a position where the detection of semeais is difficult and a little ambiguous even for human players.

For determining ρ, several detections of semeais have been launched in the position given in Fig. 3.6. With $\rho = 0.5$, only one semeai is detected and very badly. Some stones that have been detected in the same semeai are obviously unrelated to each other. $\rho = 1$ would be the ideal value if the simulations had perfect play; however, it is too strict. Because of random simulations, some strange situations can occur (e.g. a frontier group of a semeai is killed ...) and so with $\rho = 1$, some semeais could be undetected. $\rho = 0.75$ detects too many semeais. $\rho = 0.83$ is a good compromise, the quality of detections being admissible (Fig. 3.7). That's why for all experiments, $\rho = 0.83$ and $nbSims = 100$.

Improvements

The detection is not perfect, particularly in complex situations with a lot of stones on the goban. Some improvements can be made such as:

- choosing ρ dynamically. We have found empirically that open situations have better results with lower ρ values.
- correcting the detection:
  - eliminating a group surrounded by only one opponent group does not belong to a semeai but more probably to a life and death problem (see Fig. 3.25)

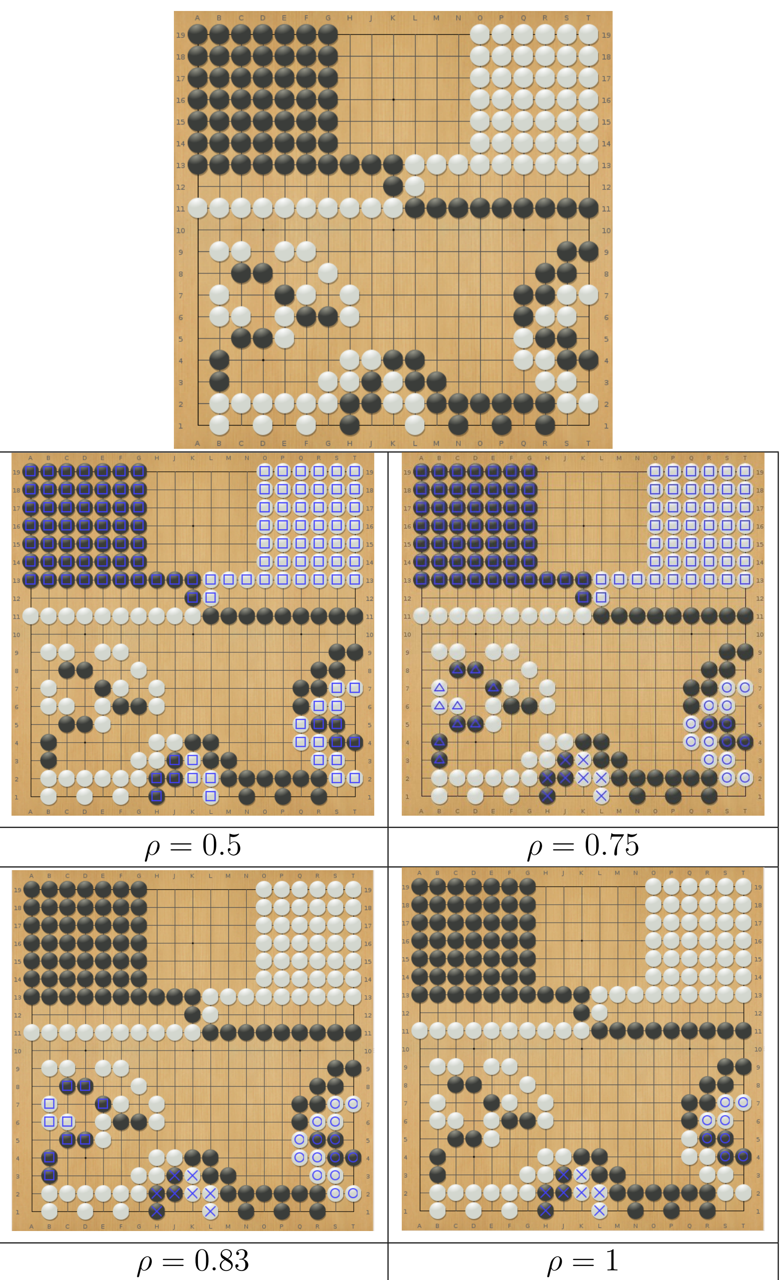


Figure 3.6: **Top:** a situation of Go for testing the detector of semeais. Top of the goban is a situation maybe too huge for considering that it is a semeai. Perhaps, the space between the 2 groups is sufficient for the life of both groups. Left of the goban is a confused situation. Right and bottom of the goban are 2 small semeais. Groups enclosing the semeai at bottom are clearly alive with 2 eyes. Groups T2 and T9 enclosing the semeai at right are not alive; even the white group T2 is in danger in comparison to the black group T9. **Bottom:** Semeais detected following ρ after 100 random simulations.

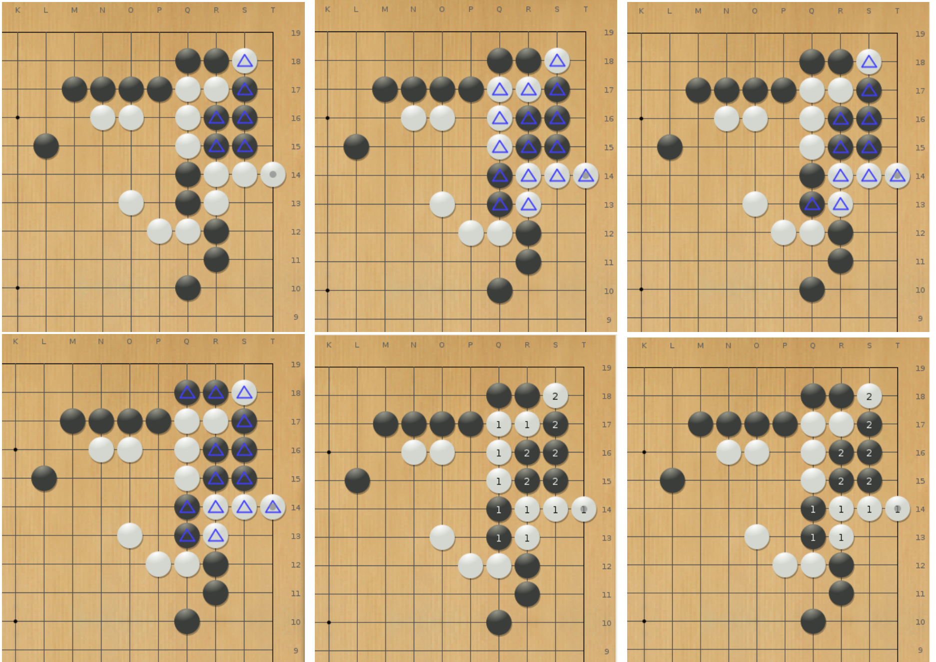


Figure 3.7: Different detections of semeais with $\rho = 0.83$ and 100 random simulations. **Top right:** In the group of black stone, the stone Q15 is in the semeai whereas the stone Q16 has not been included. **Bottom left:** The only case where the group Q18 has been included in the semeai. 2 semeais are detected in **Bottom center** and **Bottom right**. By human view, **Bottom right** can be considered as the best detection. With 100,000 random simulations and $\rho = 0.83$, **Bottom right** is the only case seen over 10 runs.

Figure 3.8: A pattern for detecting semeai.

- avoiding the situation Achille against Goliath (e.g. one group of one stone with few liberties against a big group with many liberties is not in semeai)
- using patterns such as Fig. 3.8
- correcting groups which are poorly built (Top right in Fig. 3.7 shows an incomplete group in the semeai)
- using a distance between groups (e.g. groups which are not neighbours are not in semeai - The group A19 and the group L1 at case $\rho = 0.5$ in Fig. 3.6 are in a same semeai but have no common point.).

3.4.2 Resolution

Once a semeai has been detected, the resolution begins. When several semeais have been detected, one resolution is performed for each semeai, and the biggest one is chosen (i.e. the semeai with the greatest number of black stones and white stones), among those with something to do (green rectangles in Fig. 3.5). Note that GoldenEye is not able to evaluate the importance of a semeai in the whole goban; [Coulom, 2009] could be fruitful for that.

In Fig. 3.5, the resolution starts with the question "my opponent can hope to win? ". For answering, the passmove is played (which consists in changing the player) and an estimation for winning the semeai is computed. For estimating, an algorithm of tree search with running a small number $nbMC$ of Monte-Carlo (MC) simulations is launched. Typically, $nbMC = 2,000$. When the estimation has been performed, we go backward by cancelling the pass move. The answer is yes if the semeai is solved as a win or the estimation is strictly greater than a threshold θ, In this case, we wish to solve it. We build a new tree by using the same algorithm of tree search. Since the main goal is to solve it, we run a larger number of Monte-Carlo simulations. Typically, $nbMC = 100,000$, but the search stops as soon as a solution is found. If the semeai is not completely solved, we compute an estimation exactly as we have made for the opponent. If the estimation is strictly greater than the same threshold θ, the move with the best estimation is returned.

For each candidate move, the number of Monte-Carlo simulations $nbSimulations$ and the number of winning Monte-Carlo simulations $nbWins$ are stored in memory. The winrate is a percentage defined as:

$$winrate = \frac{nbWins + K}{nbSimulations + 2K} \tag{3.4}$$

with K is a constant strictly positive fixed to 0.5. In this formula, the numbers K and $2K$ are added so that the winrate is 50% when there is no simulation for the move. The constant K has been chosen as small as possible in order to the added terms K and $2K$ becomes insignificant in the fraction as soon as possible.

The estimation is the winrate and $\theta = 50\%$.

Thus, the semeai is solved dynamically by tree search. In the following, a first part describes the motivation for the algorithm. The structure of data is given in a second part and the principle of search is given in a third part. Then we decompose the solving by describing the Monte-Carlo simulation and then the tree search. Finally, 2 main improvements are given.

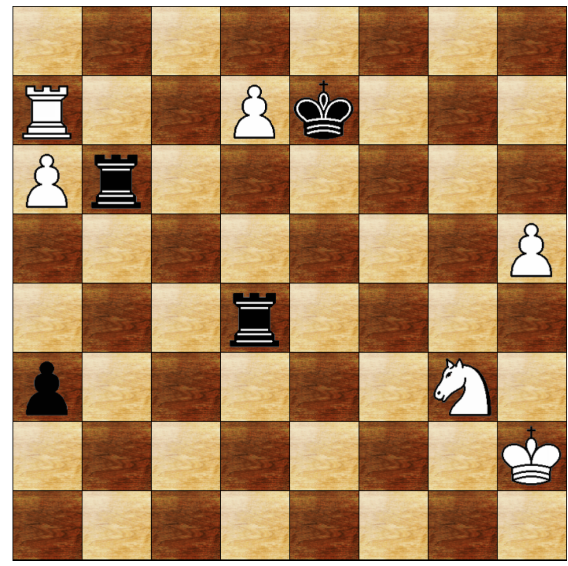
Motivation

For entertaining humans, newspapers often offer small tactical problems. The given solution is simple and short (Fig. 3.9). We wish to propose an algorithm which solves small tactical problems by building only main lines like in the solution given by the newspaper.

A main assumption is that the play is almost perfect because of much prior knowledge about the kind of problem and the simplicity of the solution (i.e. lines of the solution are not too long and there are few replies). Consequently, a small tree should be sufficient for solving the problem. Moreover, simulations are assumed to be realistic.

In Fig. 3.9, imagine the first try is 1. Nf5+ Kd8 2. Nxd4 a2 3. Nc2 Rb2 and then White loses. Either White has played the optimal way or he has committed a mistake. As Black has won and we assume that he plays almost perfect, improving the black play is useless. The main idea is to find where White has played a bad move and corrects it. Thanks to a lot of knowledge, we search the most likely error of the white player. If the error exists, the error is

- either 1. Nf5+
- or 2. Nxd4 after 1. Nf5+ Kd8
- or 3.Nc2 after 1. Nf5+ Kd8 2. Ra8+ 2. Nxd4 a2



Djaja
Donner book

1. Nf5+ Kd8 2.Ra8+
[2. Nxd4 a2 3. Rb7
(3. Nc2 Rb2 and Black
wins.) 3. ... a1Q
4. Rxb6 Qxd4 5. a7]
2. ... Kxd7 3. a7 Ra4
4. Rg8 Rba6 5. Nh6!!
and now White has
perpetual check along
the g-file !
Draw. **1/2-1/2**

White to play and draw

Figure 3.9: From the site chessbase.com. Inspiration of the algorithm. **Left:** The problem. **Right:** The solution.

We decide that the bad move is 2. Nxd4 and we decide to correct it by playing 2. Ra8+. From this position, a new simulation is launched: 2. ... Kxd7 3. a7 Ra4 4. Rg8 Rab6 5. Nh6 and the game is draw (the goal is accomplished). Now, maybe Black has played the optimal way or he has committed a mistake. We try to improve the black play and the next correction should focus on one of black moves among:

- 1. ... Kd8 after 1.Nf5+
- 2. ... Kxd7 after 1.Nf5+ Kd8 2.Ra8+
- 3. ... Ra4 after 1. Nf5+ Kd8 2. Ra8+ Kxd7 3. a7
- 4. ... Rab6 after 1. Nf5+ Kd8 2. Ra8+ Kxd7 3. a7 Ra4 4. Rg8

and so on. Note that the correction is not searched in the following line 1. Nf5+ Kd8 2. Nxd4 because Black has won in this variant. In this way, the main goal is to prove as soon as possible that the position is a draw.

Describing the architecture

Semeai: The semeai is a structure which mainly contains the list of black stones in semeai and the list of white stones in semeai. For each color, a main

stone is designed as the goal for the opponent. The semeai is won for the player who captures the main stone of the opponent. Typically, for a given color, the main stone is one stone of the largest group of that color in semeai. The structure also contains some information such as frontier groups.

The tree: The tree is a graph where a node is a position of Go and an edge is a move. For a given node nd, a child is a node obtained after the application of the move on the position of Go defined by the node nd. The root is the position of Go when the semeai is detected. A leaf is a node with no edge.

Search principle

The algorithm is an iterative best-first search (Part 1.4.9); it is akin to A^*. An iteration is composed of 3 parts (Alg. 7). A first part consists in choosing a leaf which obtains the best evaluation in the tree. The second part is the Monte-Carlo simulation starting from the chosen leaf. In this part, all the positions of the simulation are added in the tree. The third part is the update of the tree.

Algorithm 7 Building the tree. The function *playOneMCSimulation* returns the last created node at the end of the simulation.

```
Function BuildTree(tr, maxSimulations)
for sim ∈ 1..maxSimulations do
   if tr is solved then
      break
   nextLeaf = ChooseLeaf(tr.root))
   lastLeaf = playOneMCSimulation(nextLeaf)
   update the tree tr from the lastLeaf until tr.root
```

At the first iteration, the chosen leaf is the root. The problem of choosing a leaf is really raised at the second iteration. For presenting the algorithm, we propose a simple example where one iteration is sufficient for solving a semeai (Fig. 3.10 and Fig. 3.11).

The example shows several things such as:

- a Monte-Carlo simulation
- how the tree is updated

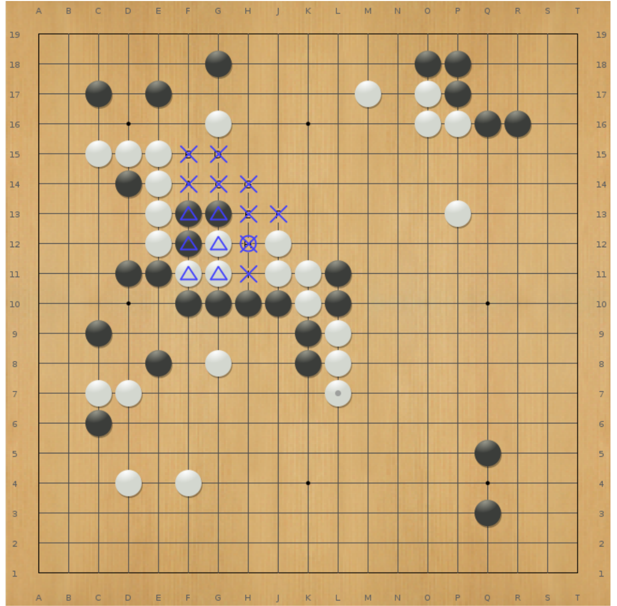


Figure 3.10: End of the game between 2 professional players O Meien and Cho Sonjin. Black to play. After H12-H11-J13-K12-K13, O Meien sees his mistake and resigns. Even if the situation is not a semeai but a shicho, the solver detects it as a semeai (groups F11 and F12) and solves it. The simulation begins by H12, the opponent plays H11 (forced move to not lose the semeai). But then, the semeai is considered as finished and lost for White because of the favorable shicho for Black. After H11, it follows J13-K12-K13-L12-M12-L13-L14-M13-N13-M14-M15-N14-O14-N15-N16-O15-N17-P15-P14-Q15-Q14-R15-S15-R14-R13-S14-S13-T14-T15-H13-T13-H12 and H14 captures the white group. Fig. 3.11 gives steps of the resolution.

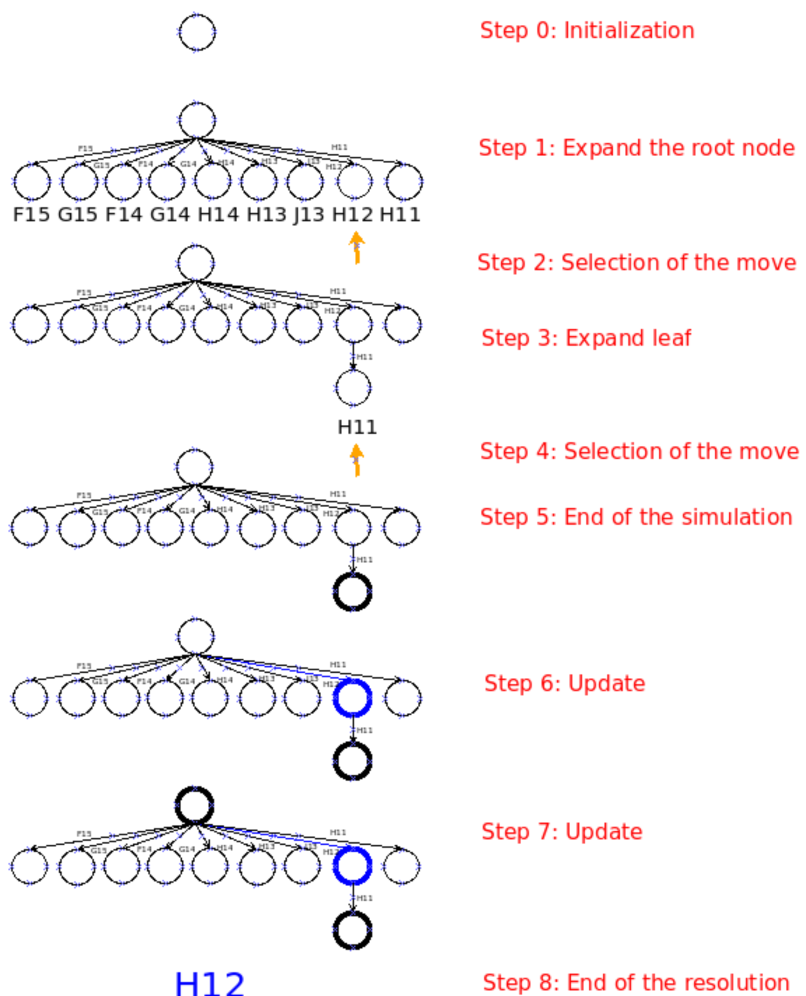


Figure 3.11: A preliminary resolution. The illustrated situation is given in Fig. 3.10. **Step 0:** creation of the root node and initialization. The simulation starts. **Step 1:** expanding the root by creating all children. **Step 2:** H12 has the best expertise (e.g. threat to win the semeai in one move); H12 is chosen. **Step 3:** the situation of the semeai is unclear; then the chosen leaf is expanded. Only one child is created because H11 is the only move which prevents to lose the semeai. **Step 4:** H11 is forced and so chosen. **Step 5:** the evaluator detects the end of the semeai by the capture of the white group with a shicho. The simulation is finished. The update of the tree begins. The last created node is so solved as a loss (thick black circle). **Step 6:** all children (here, there is only 1 child) are solved as loss, therefore the node is solved as a win (thick blue circle). **Step 7:** at least, one node is solved as a win; therefore, the node is solved as a loss. **Step 8:** the root node is solved. The resolution is finished. The semeai is a win and H12 is the winning move. A better solution may exist, but no other solution is searched.

Monte-Carlo Simulation

The Monte-Carlo simulation is given in Alg. 8.

Algorithm 8 The Monte-Carlo simulation. Arguments: lf is a leaf of the tree. The function staticEval is used to determine the end of the simulation. It returns the status of the semeai (*Win* or *Loss*) if it is able to determine it statically, or *Unknown* if it can not.

Function $playOneMCSimulation(lf)$
Let $board$ a copy of $lf.board$ ($board = lf.board$)
$nextNode = lf$
while $staticEval(board) == Unknown$ **do**
  Build the list of moves lm thanks to some rules
  for Each move $mv \in lm$ **do**
    Compute an expertise K of the move mv following the board $board$
    Add a new child ch of $nextNode$
    Store the move mv, the expertise K and the result of the application of the move mv on a copy of the board $board$ in the new child ch
  $nextNode = Argmax_{ch \in Children(nextNode)} ch.K$
  Play the corresponding move $nextNode.mv$ on the board $board$
Return $nextNode$

After presenting the main parameter of the simulation, the process for considering new moves is explained and the expansion of leaves is then presented. Some expertises are then given for doing intelligent simulations. Finally, the module which evaluates the end of the semeai (i.e. the function *staticEval*) is exposed.

Maximal length of the simulation. The horizon *horizon* of the simulation is the maximal length of a simulation, counting from the root and not from the leaf where the simulation starts.

Expansion of new considered moves. The classical expansion is for a given stone, to consider its 8 neighbours and the 4 neighbours at a distance of one jump in straight line (at Left in Fig. 3.12). When a group is captured but the simulation is not finished, all the locations where the stones have

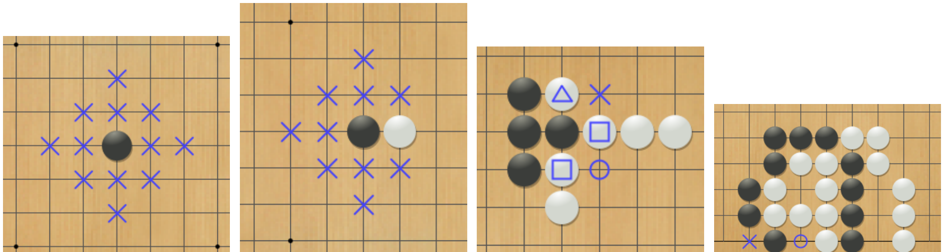


Figure 3.12: Expansion of moves around a black stone. Blue crosses are (new) considered moves. **Left:** classical expansion. **Center Left:** expansion with an opponent white stone. **Center Right:** expands or not around the black stone encircled by 3 white stones? Stones marked by blue square are strong. The move marked by a blue circle is so not considered. On the other side, stone marked by blue triangle is weak; the move marked by a cross is considered. **Right:** Approach move. If Black plays directly at blue circle, he puts itself in atari. The approach move is the blue cross.

been captured are new candidates. When a move has been considered, the move is always considered in the following sequence of the simulation (except forced cases, for instance, step 4 in Fig. 3.11). Fig. 3.12 gives main rules for expanding new moves such as approach moves. Rules are completed with specific complex conditions for preventing to add some moves evaluated as useless. For instance, the jump in straight line (*tobi*) can be not considered if the move is not directly linked to a stone in semeai.

In the phase of initialisation, considered moves are empty locations around stones in semeai (e.g. Fig. 3.4) according to rules of expansion. Each time a move is played, old considered moves from the beginning of the simulation are still candidates and new candidates are given by rules of expansion (except forced cases).

Creation of nodes. When a leaf lf is expanded, the list of considered moves is generated and for each move of the list, a child node is added to lf. We have decided to store the whole simulations in the tree (Fig. 3.11) because we expect that the size of the tree will remain sufficiently small that we won't run out of memory; also we believe that MC simulations are of adequate quality. At each node of a MC simulation, all the possible children will be added to the tree.

Expertise and policy of the simulation. MC simulation continues only on the child with the best complete expertise.

The complete expertise is a linear combination of about thirty components. These expertises are given in GoldenEye for doing realistic and almost perfect simulations. There are specific expertises for semeais. First of all, a semeai is a capturing race. Thus, filling liberties of the opponent group in semeai is really good. Increasing liberties of his own group in semeai is good too but less important, because the move delays the capture of one of the groups. On the contrary, playing a move which decreases the number of liberties of his own group is generally very bad.

There are a lot of other expertises more classical in the Go game such as saving a group in atari, patterns of connection/disconnection, approach moves, capturing big groups, creating an eye... Some expertises depend on the last move played by the opponent in the simulation, such as the expertise of blocking when the opponent tries to flee.

The policy of the MC simulation is to choose the move which has the best complete expertise K.

Evaluation of winning/losing a semeai. In the module, some techniques of captures are implemented such as shicho. The module is able to recognize if one group has built 2 eyes (alive group). It sometimes happens that one group in semeai manages to flee to the outside, and in this case the complexity of the semeai explodes. In order to avoid this bad case, the semeai is considered as a win for the player who has successfully increased the number of liberties of his main semeai group by 4.

Moreover, the player to play at the current position loses the semeai if the length of the simulation beginning at the root of the tree exceeds $horizon$ moves. The parameter $horizon$ has been fixed to 30.

The module is very efficient. For instance, the module is able to compute a long sequence for determining how the semeai finishes, for instance when the problem reduces to a shicho (Fig. 3.10). Thus, the simulations can be stopped earlier and the search tree is smaller. In counterparts, the module is computationally expensive.

The update of the tree

When the MC simulation is terminated, the update of the tree starts from the last created node in the simulation to the root of the tree. Some pieces of updated information are:

- the number of winning simulations
- the number of simulations

- whether the node is solved.

Solved node. A node is solved as “losing“ if at least, one of children nodes is solved and the result is a win. A node is solved as ”winning“ if all children nodes are solved and all results are losses. A tree is solved if and only if its root is solved (Fig. 3.11).

Other data is updated in the tree such as the result of the last MC simulation in a node.

Choosing leaf

The result res of the last MC simulation in a node is one of the main components for choosing the leaf where to start the next MC simulation. The algorithm for choosing a leaf is given in Alg. 9. The algorithm uses the depth of the leaf in the tree, the relevance of the branch from where the leaf is and the current situation on the goban linked with the leaf but the number of children or the remaining number of children in a node are not used in GoldenEye (See paragraph 3.4.2 for more details.).

One key point is that the tree is not fully covered for choosing the leaf. In particular, for a node whose last simulation has been won ($res == win$), we descend automatically to the child node where the last simulation has been won; other children are not visited at this time. It is the principle of improving the play of the actual losing player as it has been seen in part 3.4.2.

An example for choosing the next leaf is given in Fig. 3.13.

Evaluation of a leaf. The function which gives a score of the leaf lf is Alg. 10.

The evaluation of a leaf is a compromise between the expertise, the depth of the leaf and a third term which replaces the term of exploitation.

Deep leaves are penalized ($w_1 < 0$). In a leaf, the exploitation is unknown because it has not been already simulated. However, the branch from where the leaf comes is known. Nodes belonging to the branch have such statistics; a kind of term of exploitation can be computed. This evaluation will give the interest of the branch. Note that the number of children or the remaining number of children to prove for each nodes of the branch are not used for evaluating the relevance of the branch. A leaf whose branch seems to be good (i.e. well studied, with a lot of simulations) and promising (with a lot of wins) should be favored. The term is the average on exploitation terms of all nodes belonging to the branch and of the same color than the leaf. The coefficient $w_2 > 0$ weighs this ”exploitation term“. This shape of exploitation is studied in Section 3.5 and is denoted the heuristic *H0*.

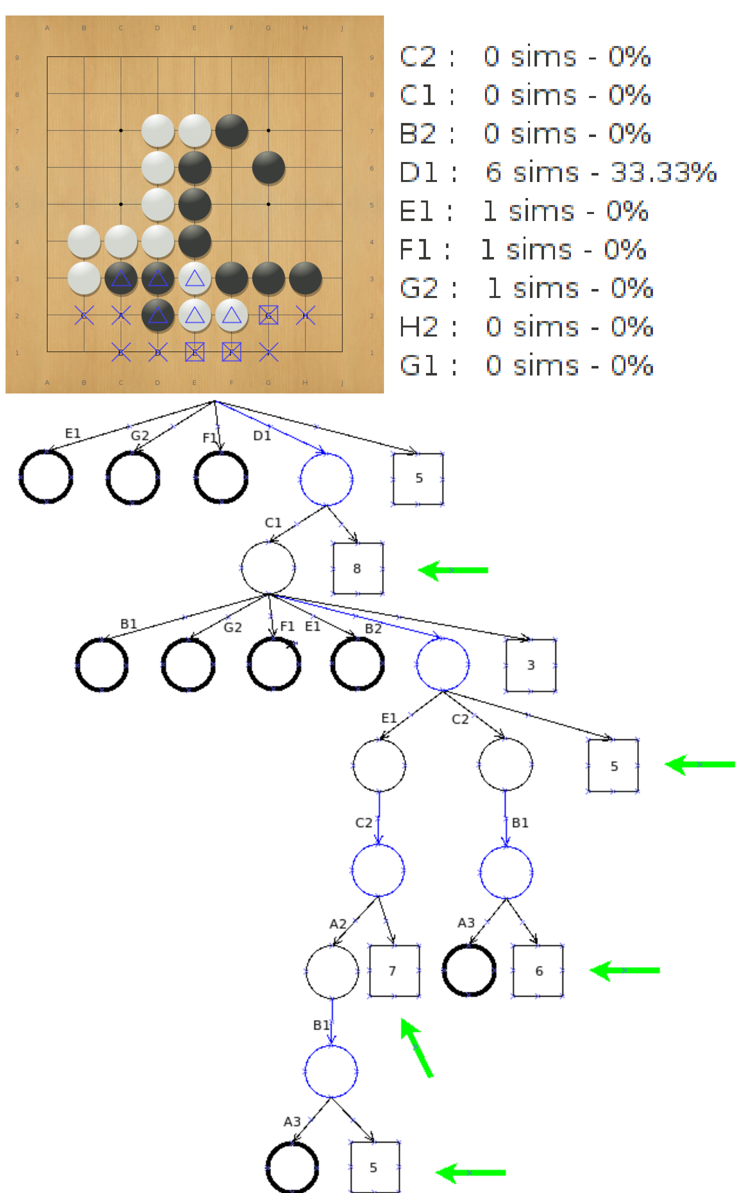


Figure 3.13: An example of solving semeai after 9 iterations. **Top Left:** The semeai (triangles) with all considered moves at beginning (blue crosses). Locations with blue square-crosses are moves proved as loss. (The winning move is D1). **Top Right:** Statistics at the root of the tree after 9 iterations. **Bottom:** The tree built after 9 iterations. A square with a number n denotes the remaining number of leaves to expand. A blue circle is a node in which the last simulation has been a win and a thick circle is a node solved as loss. At the 10th iteration, the next selected leaf will be in one of the squares pointed by a green arrow.

Algorithm 9 The recursive function in order to choose the leaf where the next MC simulation starts.

```
Function ChooseLeaf(nd)
Let res the result of the last simulation of the current
node nd.
Let lf_best initialized to an empty node (lf_best = null).
if nd is solved then
   Return null
if nd is a leaf then
   Return nd
if res is a win then
   Let wcnd be the winning child node
   Return ChooseLeaf(wcnd)
else
   for each child ch of nd do
      lf_temp = ChooseLeaf(ch)
      if lf_temp is null then
         continue
      if lf_best is null then
         lf_best = lf_temp
      else
         if evalLeaf(lf_best) < evalLeaf(lf_temp) then
            lf_best = lf_temp
   Return lf_best
```

The coefficients w_1 and w_2 are fixed parameters[2].

Some improvements in the solving

In order to solve more efficiently semeais, we present 2 generic heuristics:

- *Global Contextual Monte-Carlo*
- *Inhibition*

Global contextual Monte-Carlo. The heuristic *Global Contextual Monte-Carlo* (gcmc) is inspired by [Rimmel and Teytaud, 2010]

[2]The values of coefficients w_1 and w_2 are not specified.

Algorithm 10 Evaluation of a leaf.

```
Function evalLeaf(lf)
Let K_best be the best value of the expertise knowledge
among lf and its brothers.
Let d be the depth of the leaf lf in the tree
Let μ be the average win rate of ancestor nodes which
are of the same color than lf.
Let w_1 be the coefficient of the depth d
Let w_2 be the coefficient of the average win rate μ
Return ((lf.K − K_best) + w_1 * d + w_2 * μ)
```

For any situation, when the opponent plays a given move, the player has often an automatic answer. For example, in Fig. 3.13, if white tries to flee by playing G2, then black blocks automatically by playing H2. The idea is to find generically the automatic reply by using statistics.

It's a kind of RAVE not on one move but on a couple of moves.

2 static variables $gcmcWins$ and $gcmcSim$ depending on the color of the player, a first move and a second move are defined.

Generally, all nodes of the tree have their own variables. But in our case, for memory problems and because we need a lot of simulations in order to have significant statistics, only the root node has these 2 variables. For this reason, the heuristic is called global.

$gcmcWins$ and $gcmcSim$ are initialized to 0. These 2 variables are updated in this way: Let mv_k^i the kth move played by the ith player and a given simulation mv_1^1 mv_2^2 mv_3^1 mv_4^2 ... mv_{2j}^2 mv_{2j+1}^1 ... mv_n^i. In this simulation, the player who has won is denoted w and the player who has lost is denoted l. For all couples of moves (mv_k^l, mv_{k+1}^w) of the simulation,

$$gcmcWins[w][mv_k^l][mv_{k+1}^w]++ \tag{3.5}$$

$$gcmcSim[w][mv_k^l][mv_{k+1}^w]++ \tag{3.6}$$

For all couples of moves (mv_k^w, mv_{k+1}^l) of the simulation, only $gcmcSim$ is incremented.

$$gcmcSim[l][mv_k^w][mv_{k+1}^l]++ \tag{3.7}$$

During the choice of a leaf and the simulation, the heuristic gives a bonus in the expertise. Let $mv_{last}^{c_1}$ the last move played by the opponent c_1 and mv_{cand} the candidate move for the player c_2. The bonus is given under

the shape $coeff \times gcmcWins[c_2][mv_{last}^{c_1}][mv_{cand}]/gcmcSim[c_2][mv_{last}^{c_1}][mv_{cand}]$ with $coeff$ a tuned coefficient.

The heuristic *gcmc* is denoted *H1*.

Inhibition. The main idea is to allow the algorithm to go out of a local minimum. We wish to avoid staying in an unsolved node which has good statistics but which seems to be recently refuted. Good statistics come from a lot of winning simulations. However, the node seems to be newly bad because the k last simulations including this node have been lost. Thus, the heuristic favors the exploration.

A node is inhibited if its k last simulations have been lost. A node is reactivated if all brothers are inhibited or solved. If a node is reactivated, all inhibited brothers are reactivated, too. While a given node remains inhibited, this node and the resulting subtree are not visited anymore. In our experiments, k is fixed to 10. The heuristic *inhibition* is denoted *H2*.

The heuristic *inhibition* is a temporary pruning; the branching factor (i.e. number of children at each node) is temporarily reduced and the size of the tree to browse is smaller. Other techniques such as Progressive Widening [Coulom, 2007] and Progressive Unpruning [Chaslot et al., 2007] controls the branching factor.

Summary

GoldenEye builds incrementally a tree with a local search. The tree is initialized from the current situation on the goban. At each iteration,

1. a part of the tree is browsed in order to choose a leaf;
2. a MC simulation is then launched from the chosen leaf;
3. the tree is finally updated.

Information in the node. In a node *nd*, pieces of information are :

- *board* the board
- *mv* the move
- *res* the result of the last simulation. In the case of win, *wcnd* is the pointer on the last simulated child node. In the case of loss, *wcnd* is null.
- K the value of the expertise

- $children$ the list of children nodes (when leaf, the list $children$ is empty)
- $father$ the father of the node (when root, the pointer $father$ is null)
- d the depth of the node ($d = father.d + 1$ and when root, $d = 0$)
- $nbWins$ the number of winning simulations
- $nbSimulations$ the number of simulations
- $solv$ the result of solving. When unsolved, $solv = unknown$ and when solved, $solv = win$ or $solv = loss$.

Properties and disadvantages. As proofs are memorized by the propagation of solved nodes, GoldenEye proposes a complete resolution for open problems but only one winning move is given. When there are several winning moves, the given move may not be the best.

The scoring function is the win rate of semeais or when solved, a boolean which indicates if the semeai is won; GoldenEye does not know how many points it represents on the board.

There are aggressive but necessary prunings. The length of a simulation is bounded.

Many human expertises have been added. It allows to have efficient and natural MC simulations. With efficient simulations, we hope to build the tree as small as possible. Thereby, GoldenEye can help humans to understand a semeai and see the main variants of an open semeai. GoldenEye saves the tree built during the solving in a sgf file and a user can look into the solution with a human interface such as gogui. Another goal of this solver has been to collaborate with other go engines. Thus, the solver used in MoGo (See Section 2.3.2) is this version of GoldenEye.

3.4.3 Hierarchical solving

The algorithm can not solve some kinds of semeais because with the best defense/attack, sometimes the semeai finishes in a draw (i.e. seki) or with some conditions (i.e. ko). Fig. 3.14 shows examples.

A semeai can finish

- by the death of one group (most common)
- by the life of both groups, too (seki)
- or by the death of one group with a ko.

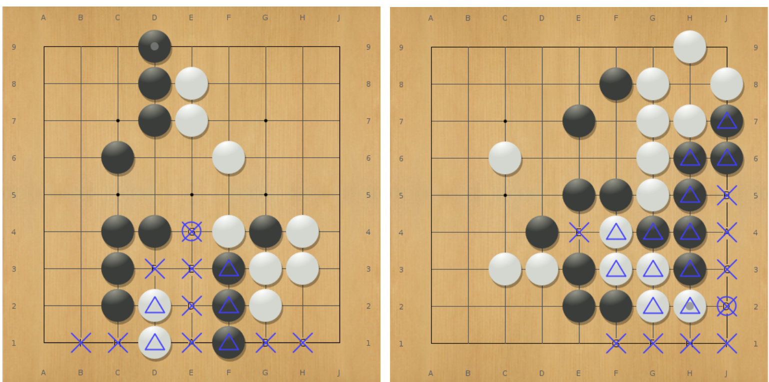


Figure 3.14: Different possible ends of the semeai. **Left:** A created situation. White to play - the semeai ends by a seki after the sequence E4-E3-D3-C1-G1. **Right:** From a game between 2 amateur players. Black to play - a ko appears in the sequence J2-J4-J3-J1-H1-G1-E4-J1(ko).

4 ways to win a semeai have been identified:

- win
- win with ko
- draw with ko
- loss with ko

In order to be able to solve all cases, several trees (4 in total) are built. These trees are built independently with their own goal. The first tree tries to win without ko. The second tree tries to win with ko. The third tree tries to draw with/without ko (seki) and the last tree tries to lose with ko.

The order - win > win without ko > draw > loss with ko - is similar to the evaluation function in [Cazenave, 2001] for solving death/life problems. But claiming that a win with ko is better than a draw without ko is debatable because it depends of the ko threats on the whole goban. However, the situation of draw is less frequent. So, we have not distinguished cases ”draw without condition“ and ”draw with condition“ and we have estimated that the win with ko is better.

The algorithm is given in Alg. 11. We switch from one tree to another, going to more ambitious when the winrate is more than 66% and less ambitious when the winrate is less than 33%. GoldenEye now solves both semeais given in Fig. 3.14.

Conditions of winning semeai is dependent of the tree. For example, in the building of the first tree, if a ko is met during the simulation, the simulation is considered as a loss whereas in the other trees, the simulation can be considered as a win.

Simulations are not reused between trees, however good moves at the root are transmitted under the form of a bonus of expertise in other trees.

The final choice of the decision is a little bit complicated to explain. For choosing the move which should be played, we must first choose the best tree among the 4 and then from this tree, the best move. The choice of the best tree depends on whether the tree is solved, and in this case it leads to a win or a loss. By assuming that no tree is solved, we take the first tree where several simulations have been made and whose winrate is more than 66% in the classical order win > win without ko > draw > loss with ko. Is it better to choose the move given by the solved tree "win with ko" or the move with a winrate of 60% given by the tree "win without ko" but unsolved? The question is open.

3.5 Results

This section shows some experiments on the GoldenEye algorithm. First, experiments are performed on a testbed of 10 semeais. Then, improvements about the exploitation, *global contextual Monte-Carlo* and *inhibition* called respectively *H0*, *H1* and *H2* are compared. A study of the parameter *horizon* is then made. Some resolutions of difficult and classical semeais are shown. A question about the accuracy of winrate will show that this information could be sparingly taken account. At the end, some failure cases will be shown.

3.5.1 Semeais from Yoji Ojima

GoldenEye has been tested a lot on semeais created by Yoji Ojima[3] in April 2008. These problems are given in Tab. 3.1. and can be found at http://comments.gmane.org/gmane.games.devel.go/15386.

Results of GoldenEye on these semeais are shown in Tab. 3.2. Through these results, we can conclude that all semeais are well-detected by GoldenEye. This is already a good result. The semeai *mc148* is an exceptional case. GoldenEye returns always the good move until 10^4 simulations. With 10^5 simulations, GoldenEye returns only 1 time over 2 the good answer. In the solving, GoldenEye is optimistic but then becomes pessimistic after a deeper

[3] the author of the strong engine Zen

study of the semeai. Certainly, GoldenEye finds a good response for the opponent but he does not see the refutation of this move or maybe, the semeai can not be won without ko. But globally, the quality of solving increases with the number of simulations. When simulations are overspread by packs on different ways to solve the semeai, all distributions seem to be equivalent.

For comparison, MoGo has been tested on these 10 semeais. Results are given in Tab. 3.3. If we compare Tab. 3.2 and Tab. 3.3, with a same number of simulations,

- when the number of simulations is small (i.e. $\leq 10^4$), GoldenEye fails around 2 times less than MoGo.
- however, when the number of simulations is higher than 10^4, numbers of failures are almost equals; GoldenEye and MoGo are equivalent.

Thus, at constant number of simulations, GoldenEye is better on these 10 semeais. For measuring at constant time, we have remarked that the execution time of a simulation of GoldenEye is 10 times slower than MoGo. With this assumption, we can conclude that at time constant MoGo and GoldenEye are equivalent for well playing the 10 semeais. An interesting point is that MoGo and GoldenEye are complementary; semeais where MoGo failed are well-solved by GoldenEye and vice versa.

3.5.2 Comparison between improvements

Now, we propose different comparisons about the exploitation term (*H0*), the *global contextual Monte-Carlo* (*H1*) and the *inhibition* (*H2*).

Experiment about the exploitation term Tab. 3.4 is a preliminary experiment about the impact of the exploitation term in the full resolution of semeai. We can see that semeais are fully solved more rapidly. Over 5 runs and whatever the semeai, the maximal number of performed simulations with *H0* is smaller than the minimal number of performed simulations without *H0*. This conclusion is not so clear in Fig. 3.15, but it seems to be confirmed for the semeai *mc147* in Fig. 3.16. For all full resolutions with *H0* (orange curve), less simulations have been performed in comparison without *H0* (blue curve). The exploitation term seems to have a positive impact in the resolution of semeais.

Study about the number of simulations for solving completely a semeai In Fig. 3.15, the first and the third semeais are solved very quickly

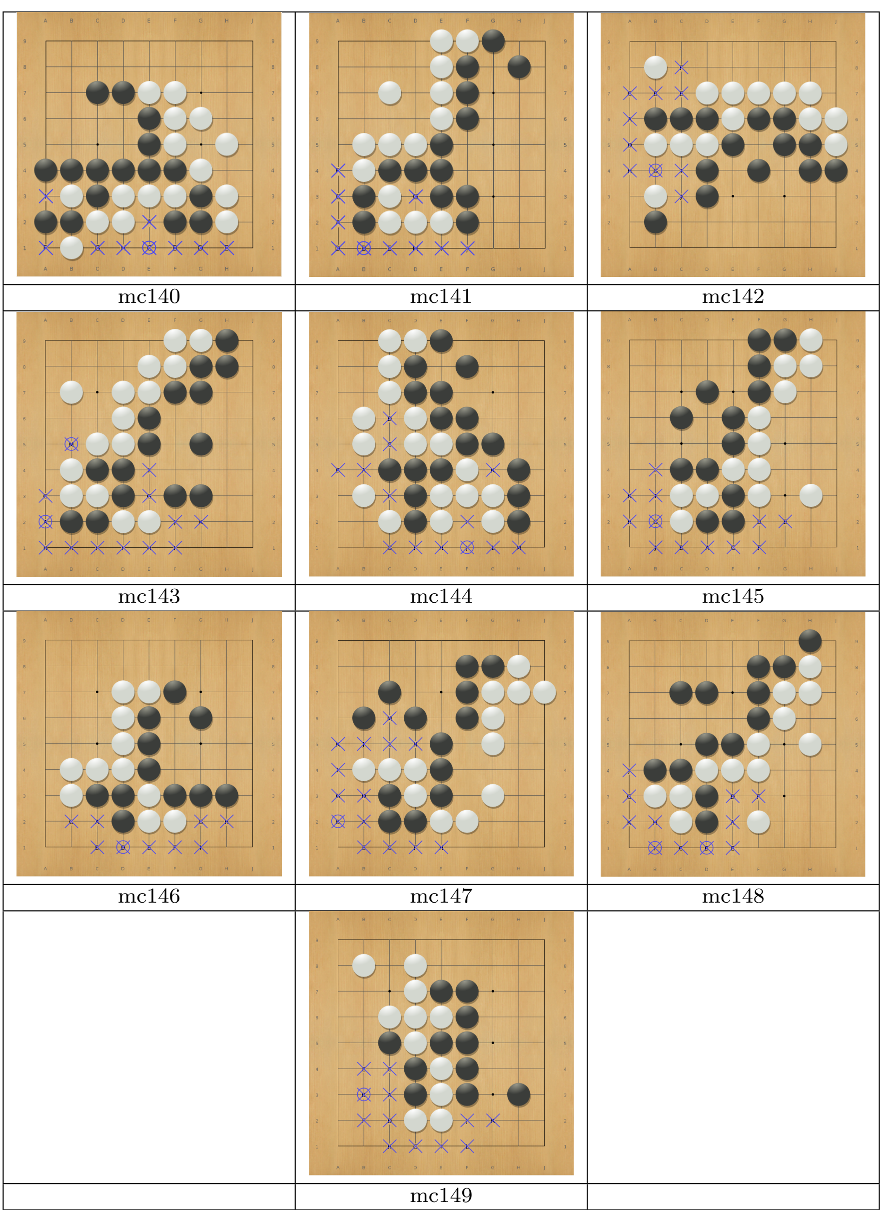


Table 3.1: The 10 semeais from Yoji Ojima. Black to play

GoldenEye	10	20	50	100	10^3	10^4	10^5	(10;10^3)	(100;100)	(10^3;10)	(10^4;1)
mc140	0	0	20	20	20	20	20	20	20	20	20
mc141	0	0	0	5	20	20	20	20	20	20	20
mc142	0	20	20	20	20	20	20	20	20	20	20
mc143	0	0	7	6	8	18	20	18	17	18	18
mc144	0	0	0	19	19	20	20	20	20	19	20
mc145	0	13	11	20	20	20	20	20	20	20	20
mc146	0	0	20	20	20	20	20	20	20	20	20
mc147	0	0	0	0	5	10	20	17	17	12	16
mc148	0	0	0	0	20	20	9	20	20	20	20
mc149	0	0	0	1	8	10	8	14	16	11	14
Failed	200	167	122	89	40	22	23	11	10	20	12

Table 3.2: Results on the semeais from Yoji Ojima following the couple (*nbPacks*;*nbSimByPacks*). For the seven first columns, $nbPacks = 1$, which means that the hierarchical solving is deactivated; only the mode "win without condition" is used. For each experiment, 20 runs have been performed. For each experiment in the four last columns, the hierarchical solving is activated ($nbPacks > 1$); a total of 10,000 simulations have been overspread differently. The last row is the total number of unsuccessful runs; the maximal number is $nbRunsPerExperiment \times nbProblems = 20 \times 10 = 200$.

MoGo	100	200	500	10^3	10^4	10^5
mc140	12	14	20	20	20	20
mc141	4	7	10	14	18	20
mc142	0	0	1	3	16	20
mc143	6	7	9	14	19	17
mc144	0	0	0	1	13	16
mc145	0	2	2	13	19	20
mc146	5	9	19	20	20	20
mc147	0	0	0	0	0	1
mc148	4	1	0	3	20	19
mc149	0	0	11	17	19	20
Failed	169	140	128	95	36	27

Table 3.3: Results of MoGo on the semeais from Yoji Ojima. For each experiment, 20 runs have been performed. The first row is the number of simulations.

mc144		mc145	
H0	no *H0*	*H0*	no *H0*
3,676	7,969	1,673	1,820
3,920	9,622	1,675	1,844
4,538	14,145	1,676	1,887
5,018	17,435	1,679	1,946
6,032	19,476	1,680	2,147

Table 3.4: Impact of the exploitation term in the full resolution of semeais *mc144* and *mc145*. Heuristics *H1* and *H2* are activated.

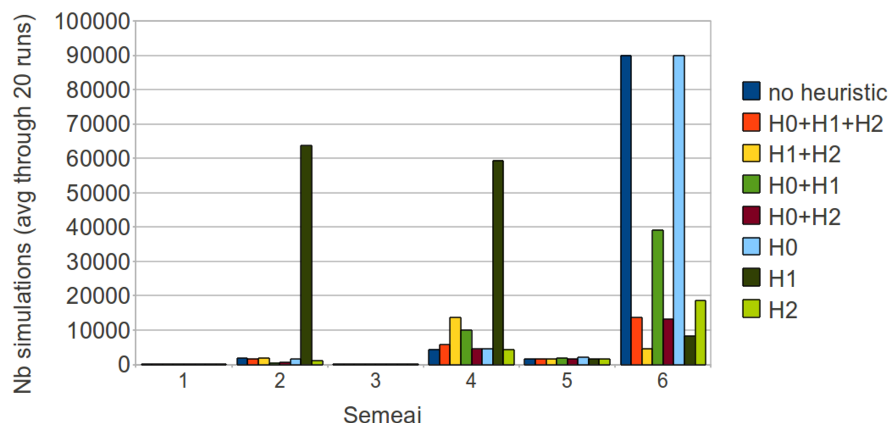


Figure 3.15: Mean number of simulations for solving completely a semeai. *H0*, *H1* and *H2* correspond respectively to $w_2 > 0$, *global contextual Monte-Carlo* and *inhibition*. When no *H0*, $w_2 = 0$. A limit of 30,000 simulations has been imposed. The y-axis is the average of (i) number of simulations when full resolution or of (ii) 3 $\times$ the limit (i.e $3 \times 30,000 = 90,000$) when no full resolution. The semeais 1,2,3,4,5 and 6 are respectively mc140, mc141, mc142, mc144, mc145, mc146. In terms of full resolutions, the heuristic *H0* alone seems to be good in the semeai mc144 but does not improve or damage in other semeais. The heuristic *H1* alone is good in the semeai mc146 but gives very bad results in the semeai mc141. Even combined with other heuristics, the heuristic *H2* gives good results in all cases.

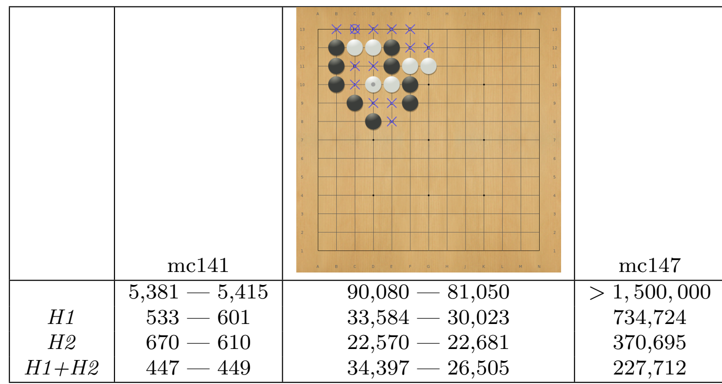
	mc141		mc147
	5,381 — 5,415	90,080 — 81,050	$> 1,500,000$
H1	533 — 601	33,584 — 30,023	734,724
H2	670 — 610	22,570 — 22,681	370,695
H1+H2	447 — 449	34,397 — 26,505	227,712

Table 3.5: Three semeais with black to play. The table gives the number of simulations done in order to have a complete resolution of the semeai. The semeai on the third column is the problem 1201 from goproblems.com. In this experiment, *H0* is always activated (i.e. $w_2 > 0$). $> x$ means that there is no complete resolution after x simulations. When there are 2 scores, it corresponds to the results of 2 runs. Through these preliminary results, Alone, *H1* or *H2* improves the rapidity of the resolution. Cumulative, we can not conclude. *H1+H2* seems to be not as good as *H2* alone on the second problem but better than *H1* alone or *H2* alone on semeais mc141 and mc147. On the semeai mc147, without heuristic, the solver has not be able to give a complete resolution after 1,500,000 simulations in the run whereas with *H1*, only 700,000 has been sufficient. With *H2*, we can divide the number of simulations by 2 and the combination of *H1* and *H2*, the number is still divided by 2 for finally solving completely the semeai with around 200,000 simulations.

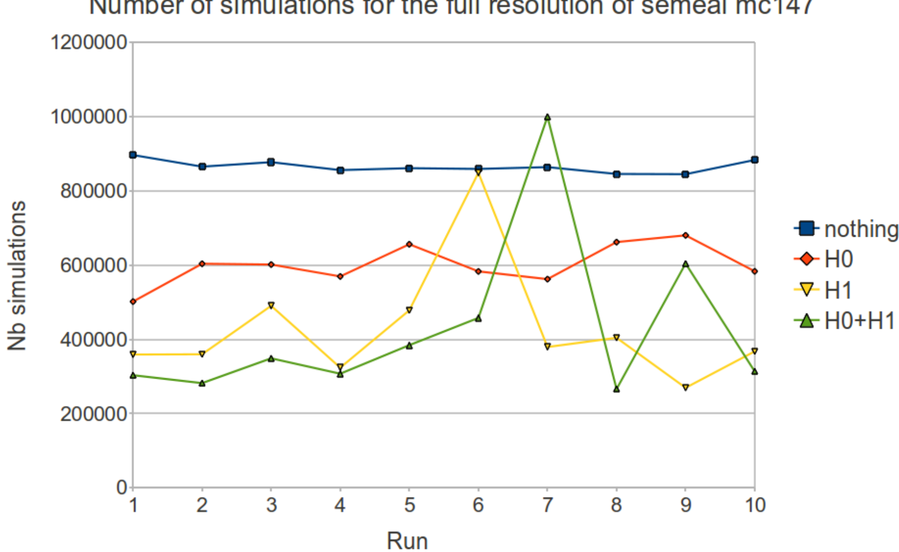


Figure 3.16: Number of simulations for solving completely the semeai mc147. In this experiment, the heuristic *H2* is always activated. When deactivated, no result can be obtained due to memory swap. A limit of 1,000,000 simulations has been imposed.

by the solver in all variants. The last semeai is the most complicated. Without heuristic, GoldenEye has been unable once time over 20 to provide a full resolution. Adding *H0* seems to brings nothing. However, with the heuristic *gcmc* alone, GoldenEye solves it very fast. With the heuristic *inhibition*, it is more true. The combination seems to be more efficient. However, the heuristic *gcmc* can introduce a very bad bias as we can see on the second semeai. The 2nd semeai is solved very quickly (around 1,000 simulations are sufficient), but with *gcmc* alone, the solver is sometimes no more able to fully solve it in less 30,000 simulations.

For all cases, the heuristic *inhibition* is the only modification with which we can claim that the heuristic improves the solver for a faster full resolution.

These results are inclined to be checked in Tab. 3.5.

In Fig. 3.16, the bad bias of the heuristic *gcmc* seems to be confirmed. But globally, in the most of runs, the semeai is solved 3 times more quickly with the heuristic *H1*. Using the Wilcoxon test, we get the following statistical confidence ($1 - p$, with p the p-value of the null hypothesis):

- p(*H0* is better than nothing) = 0.99992
- p(*H1* is better than nothing) = 0.99986

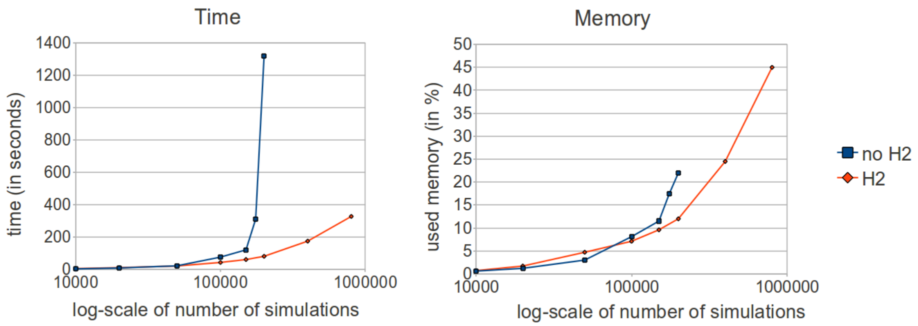


Figure 3.17: Execution time and memory used for solving the semeai *mc147*. *H0* and *H1* are deactivated.

- p(*H0*+*H1* is better than nothing) = 0.99875
- p(*H1* is better than *H0*) = 0.99875
- p(*H0*+*H1* is better than *H0*) = 0.99369
- p(*H0*+*H1* is better than *H1*) = 0.81783

The heuristic *H0* seems to be less efficient but more stable; all runs with only *H0* have given just about 600,000 simulations. *H0* or *H1* alone seems to bring something in the full resolution of semeais. The combination of *H0* and *H1* does not solve the problem of the instability of the heuristic *H1*. Following the test of Wilcoxon, the combination of both heuristics is better than *H1* alone with a probability of 81%.

When *gcmc* works (i.e. *H1* activated), semeais are solved very more rapidly (for instance, the number of simulations is divided by 3 on semeai *mc147* and on semeai *mc146*, the heuristic *H1* alone is the most efficient after the combination of *H0* and *H1*). However, *gcmc* can introduce a very bad bias in the resolution of the semeai and then the behaviour of the resolution is completely reversed (e.g. semeais *mc141* and *mc147*). Yet, next experiments are performed with *H0* and *H1* activated.

Impact of the heuristic *Inhibition* in terms of time and memory
Following a fixed number of simulations, the execution time and the used memory have been measured when the heuristic *H2* (*inhibition*) is or is not activated.

In Fig. 3.17, in terms of time, the heuristic *H2* allows to win a lot of times. When *H2* deactivated, the increase of the curve is exponential and dramatically accelerates a lot around 175,000 simulations. With 200,000 simulations,

the execution time is greater than 20 minutes, whereas the execution time is smaller than 2 minutes with *H2* activated. Certainly, the algorithm without *H2* becomes inexploitable after 200,000 simulations. The increase of time is explained by the kind of the algorithm. For remembering, the algorithm is like A^*. Even if GoldenEye does not cover the whole tree in our variant, the more the tree grows, the more there are leaves, and so the more the selection of next leaf takes time. One of the property of the heuristic *inhibition* is to makes smaller the tree to cover; some parts of the tree are not visited. The heuristic *inhibition* delays intelligently the explosion of the execution time. Even if in Fig. 3.17, we don't observe an exponential curve (orange curve), we can assume with more simulations, we will observe the same result as the blue curve.

The delay is intelligent because it inhibits some parts of the tree which becomes unpromising and economizes simulations to refute them. Without the heuristic *H2*, the good move A2 is not the move returned even after 150,000 simulations, whereas with *H2*, after 10,000 simulations, GoldenEye seriously begins to study the good move A2 and can even already consider it as the best move. One of this intelligent mechanism is that the good move seems to be found more efficiently.

Another property of the modification *inhibition* is memory savings. In Fig. 3.17, when no *H2* with a fixed number of simulations, the execution time is greater, but moreover, the used memory becomes greater after 100,000 simulations. With 200,000 simulations,

- when *H2*, only 13% of the 4 gigabytes of RAM are used
- when no *H2*, around 22% of the memory are used.

Like the explosion of the execution time, the swap of the memory is delayed, too. I have no logical explanation of that. Maybe, simulations are smaller or GoldenEye does not go in complicated variants.

The heuristic *inhibition* seems to be efficient:

- less time and less memory are used in order to solve fully the semeai
- for finding more rapidly the good move

However, these points should be verified on other semeais (in particular, memory savings which has no logical explanation).

3.5.3 Study of the parameter *horizon* of the simulation

The horizon *horizon* of the simulation is the maximal length of a simulation, counting from the root of the tree.

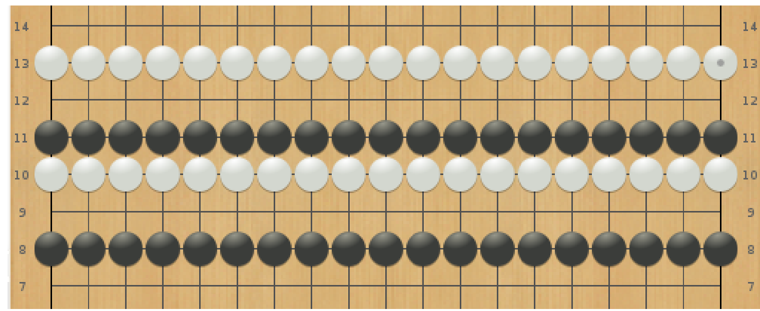


Figure 3.18: Black to play. What does it choose the *horizon* of the simulation for this problem? 37 moves (19 Black moves+18 White moves) are necessary to capture the white group on the 10th row.

Mode	Resolution with $horizon = 33$	Resolution with $horizon = 34$
only win	200 sims - T9 (10 sims - 4.54%)	1000 sims - R9 (1000 sims - 99.95%)
at least, win with ko	500 sims - T9 (13 sims - 3.57%)	no simulation
at least, draw with ko	300 sims - J9 (300 sims - 99.83%)	no simulation
at least, loss with ko	no simulation	no simulation

Table 3.6: Resolution of the semeai in Fig. 3.18 following 2 horizons. "500 sims - T9 (13 sims - 3.57%)" means 500 simulations (5 packs of 100 simulations) have been made in the tree "win with ko", the best move is T9, 13 simulations beginning by T9 have been done and 3.57% of these simulations have led to a win. These results show that the horizon of the simulation should be well-tuned. Except the parameter *horizon*, both experiments have been launched with the same parameters (i.e. $nbPacks = 10$ and $nbSimByPacks = 100$).

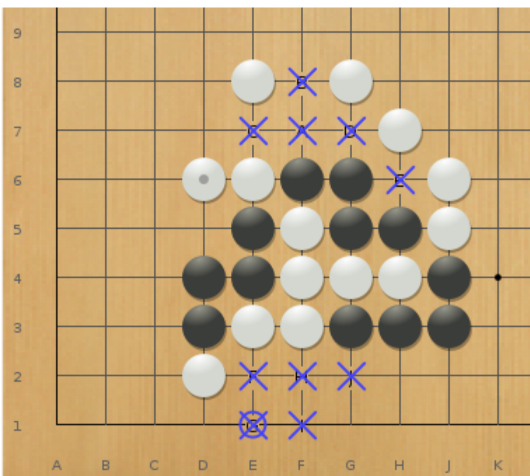


Figure 3.19: The raccoon semeai from a webpage of Denis Feldman. Black to play. E1 is the winning move.

For other problems, we have fixed the *horizon* to 30. But in the Fig. 3.18, the semeai is too big for being solved in 30 moves. In fact, at least 38 moves are necessary for capturing the white group with the best defense. So, the horizon simulation should be at least 37. However, in our case (Table. 3.6), 34 moves are sufficient because the evaluator of the end of semeai (i.e. the module which evaluates the end of the MC simulation - See paragraph 3.4.2) is able for some cases to answer several moves in advance. On the other hand, the resolution is clearly false with a horizon of 33 moves.

The second study is about the impact of the *horizon* on a complete resolution of a semeai. We have decided to take the semeai called *raccoon* (Fig. 3.19).

All results shown in Fig. 3.7 involve runs in which a good detection[4] of the semeai has been made. Results in Fig. 3.7 show that the bigger the horizon is, the more simulations are necessary for a complete resolution (more than 10,000 simulations), whereas between 1,000 and 2,000 simulations are sufficient to solve the raccoon semeai. A second point is that if the horizon is too small, it leads to a bad full resolution. If the horizon is optimal $horizon = 12$ in the sense that it corresponds to the maximal length of an optimal solution, the semeai is solved faster. In fact, less than 1000 simulations are needed. But sometimes, if a bad way in the resolution is taken, it can try another mode of win and lose precious simulations for solving them. It happens one time in the run 3. The main risk to have a horizon

[4]some tries have led to a bad detection or no detection of the raccoon semeai due to the unclear status of the black group D3 in Fig. 3.19.

		5	10	12	15	30	60
	W	100 - F2(18%)	2100 - F7(23%)	763 - E1(win)	2541 - E1(win)	1254 - E1(win)	11137 - E1(win)
1	WK	595 - ?(loss)	97800 - F7(1.2%)	0	0	0	0
	DK	22 - F2(win)	76 - F2 (win)	0	0	0	0
	LK	0	0	0	0	0	0
	W	200 - C2(30%)	1500 - F2(23%)	523 - E1(win)	1692 - E1(win)	1667 - E1(win)	13600 - E1(98%)
2	WK	1456 - ?(loss)	98400 - F7(0.4%)	0	0	0	0
	DK	188 - C2(win)	76 - F2(win)	0	0	0	0
	LK	0	0	0	0	0	0
	W			6146 - E1(win)	2531 - E1(win)	1908 - E1(win)	1537 - E1(win)
3	WK			798 - E1(win)	300 - F2(73%)	0	0
	DK			0	0	0	0
	LK			0	0	0	0
	W			567 - E1 (win)	2959 - E1(win)	51662 - E1(win)	10^4 - E1(99.9%)
4	WK			0	0	0	0
	DK			0	0	0	0
	LK			0	0	0	0
	W			1742 - E1(win)	2066 - E1(win)	1542 - E1(win)	
5	WK			0	0	0	
	DK			0	0	0	
	LK			0	0	0	

Table 3.7: Resolutions on the semeai called ”raccoon” (Fig. 3.19) following different horizons. The first row is the value of the parameter *horizon*. The first column is the number of the run. W, WK, DK and LK are respectively the mode ”win“, ”win with ko“, “draw” and “loss with ko”. For reading results, there are different cases. Case 1: the cell containing “100 - F2(18%)“ means “ With the *horizon* = 5, for the run 1, 100 simulations have been done with the mode ”win“, the best move is F2 with a winrate of 18%. Case 2: ”798 - E1(win)“ means ”With *horizon* = 12, for the run 3, after 798 simulations, the semeai has been solved in the mode “win with ko”; the solution is a win and E1 is the winning move. Other cases, “0” means no simulation and when the resolution leads to a loss, no move is proposed. All experiments have been launched with $nbPacks = +\infty$ (in theory) and $nbSimByPacks = 100$.

too huge is to lose the solver in deep and useless variants. The simulation given at left in Fig. 3.21 is an example; many useless moves are played and the simulation seems to go towards the center of the goban. The simulation becomes inefficient; this situation should be avoided.

Thus, the choice of the parameter $horizon$ is a compromise between a correct full resolution (i.e. $horizon$ not to small) and efficient Monte-Carlo simulations (i.e. $horizon$ not to huge). We suggest to take $horizon = 30$ but $horizon = 20$ is sufficient for many cases. However, if you wish to fix the parameter dynamically, you can count liberties of groups of semeai. It gives an idea of a lower bound lb_{hor} for the $horizon$. In our experiment (Exp. 3.5.3), 30 is clearly not enough (because both white group and black groups in semeai are 19 liberties); 38 moves are at least necessary for capturing one group. Because of the variety of moves played in a semeai such as approach moves or moves increasing the number of liberties of one group, we suggest to add 10 for taking account such moves. We suggest dynamically to take $horizon = lb_{hor} + 10$.

3.5.4 Difficult semeais

In this subsection, we present some semeais with a high level of difficulty solved by GoldenEye.

Fig. 3.20 and Fig. 3.22 present these semeais. All resolutions have been launched with $horizon = 30$, $nbPacks = +\infty$ (in theory) and $nbSimByPacks = 100$.

First Semeai: a win with ko

The resolution of problem 378 (Fig. 3.20) has been tested on different sizes of goban.

Tab. 3.8 presents the resolution of this semeai on different sizes of the goban. For each size, the total number of simulations has been around 100,000 simulations (e.g. $84,400 + 14,321 + 1,200 \approx 100,000$ in 9x9). The 3 experiments have been launched with $nbPacks = 100$ (i.e. $+\infty$ in theory). The resolution is faster in 9x9 and in 13x13 than in 19x19. In 19x19, a lot of simulations (around 40,000) are spent uselessly in modes ”draw” and ”loss with ko”, whereas 20,000 simulations are sufficient for solving in mode ”win with ko“.

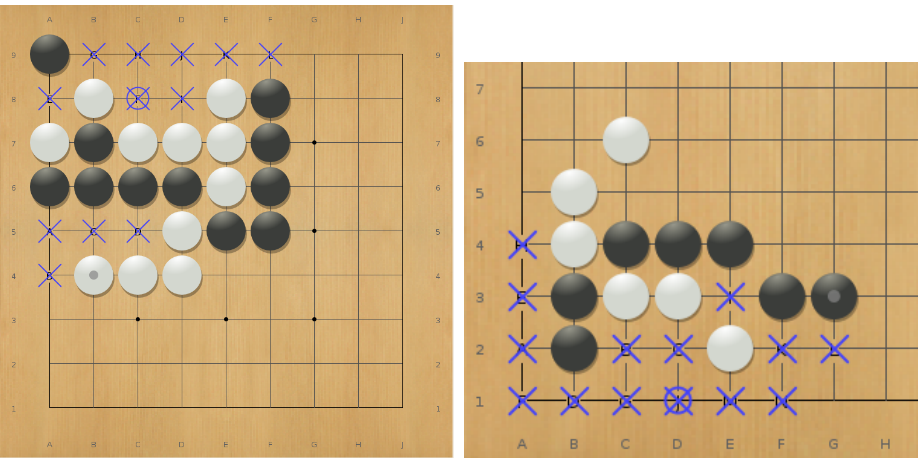
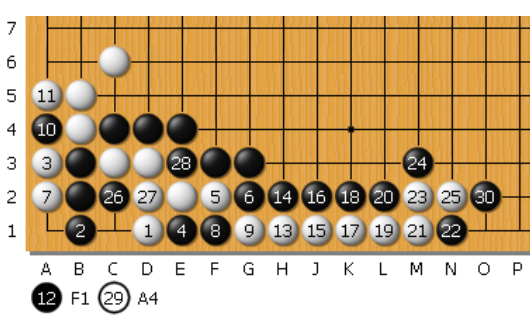


Figure 3.20: 2 semeais of level 5d. **Left:** Problem 378 from the site goproblems.com. Black to play. The algorithm solves it well as a “win with ko“. The ko appears in the following line: C8-C9-E9-D8-F9-D9-A8-A5-B9-A7-B9-A9-A8(ko)-B5. **Right:** Problem from the site wikipedia.fr. White to play. Win without condition. The winning move is D1.

	9x9	13x13	19x19
W	84400 - C8(9608 - 5.82%)	88100 - C12(8921 - 6.58%)	43700 - C18(4590 - 9.07%)
WK	14321 - C8(win)	11887 - C12(win)	19541 - C18(win)
DK	1200 - C9(624 - 95.76%)	no simulation	17435 - C18(win)
LK	no simulation	no simulation	19166 - C18 (win)

Table 3.8: Resolution of the problem 378 (Fig. 3.20) on different sizes of goban. In 9x9, 13x13 and 19x19, the winning move is respectively C8, C12 and C18. The interesting lines are WK, DK and LK. The line WK shows the number of simulations for a full resolution in the mode win with ko. Lines DK and LK show the number of uselessly spent simulations. The line W is not interesting because on 3 sizes, the win without condition (line W) has not been proved as impossible (loss) or possible (win).



Move	Nb Sim	Winrate
A2	1944	16.04%
C2	6649	70.91%
D2	3993	53.14%
B1	2373	25.32%
A3	1786	11.25%
A1	737	0.67%
C1	1624	5.48%
A4	490	7.34%
E3	1616	6.68%
D1	553393	98.64%
F2	6724	57.43%
G2	1296	5.94%
E1	2038	11.18%
F1	2421	18.95%

Figure 3.21: Statistics after the full resolution of the semeai at right in Fig. 3.20. **Left:** one of the most simulated sequences. **Right:** number of simulations done and winrate for all considered moves in the initial situation (Right in Fig. 3.20)

Second Semeai: an open semeai

The semeai has been tested on a 19x19 goban. The algorithm has solved it in 589445 simulations.

Other semeais

Fig. 3.22 shows 2 other difficult semeais tested on 19×19 goban and the resolution of GoldenEye. The first semeai (at left) is almost fully solved. GoldenEye has seen a win with a ko, but he has not finished to prove that the semeai can not be won without a ko. The second semeai (at right) is not fully solved but GoldenEye is very optimistic about the win. The engine returns the good move Q1.

3.5.5 The accuracy of the score

When the resolution is not finished, the score of the move is given by its winrate (Eq. 3.4).

C2 is a move which loses the semeai in the semeai given at right in Fig. 3.20. However, in the table given in Fig. 3.21, after 6649 simulations beginning by C2, the winrate is still high (around 70%). When the resolution is not finished, can we trust the winrate?

Before solving a semeai, it is logical to have a winrate closed to 90%. The curve (in particular, the case "win is ko") indicates that it is not true. Moreover, there is no clue such as an increase of the winrate in order to know if the semeai will be won. The curve "win with ko" has decreased

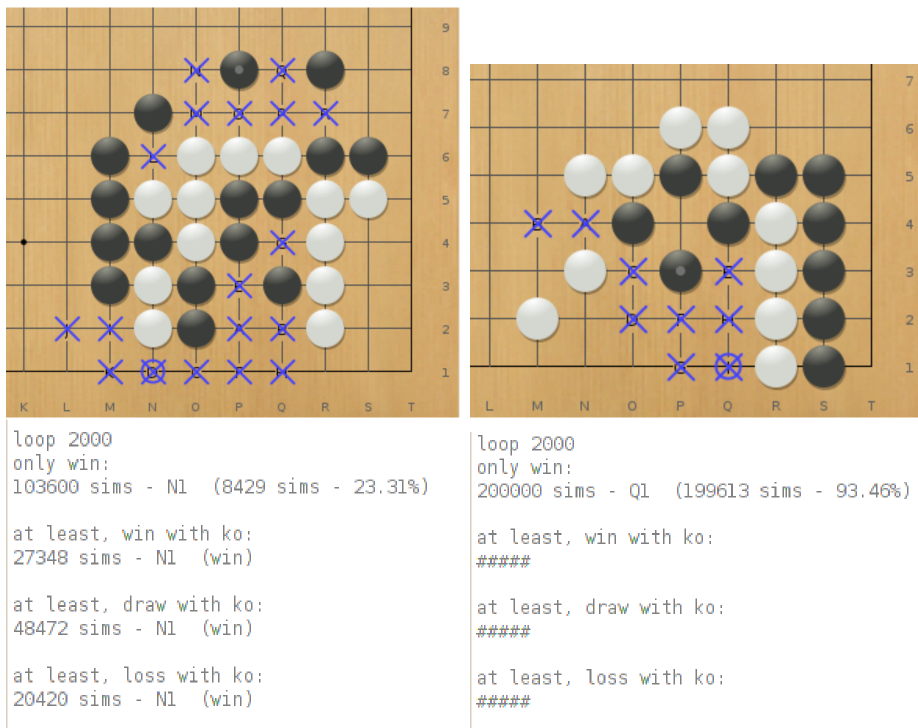
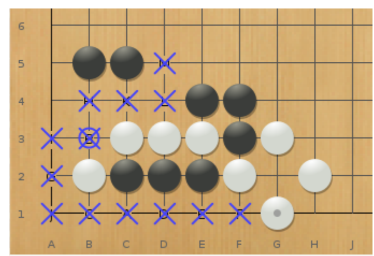


Figure 3.22: 2 difficult problems from the site goproblems.com (**Top**) and logs of GoldenEye about the resolution after 2000 $\times$ 100 simulations (**Bottom**). White to play. **Left:** problem 196 (level 6d). N1 is the winning move. **Right:** problem 122 (level 6d), classical semeai called "pear and ant". Q1 is the winning move. Q1 is a very difficult and illogical move: (i) a bad shape (empty triangle) appears, (ii) the move is located on the first row and moreover, (iii) the number of liberties of the white group R1 does not increase after Q1.



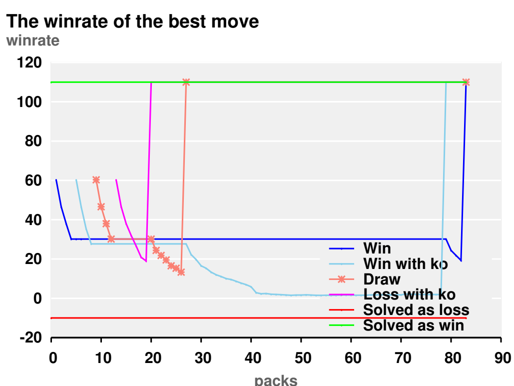


Figure 3.23: Accuracy of the score **Left:** The tested semeai. Black to play. The winning move is B3. **Right:** Score of the best move (i.e. winrate in %) for each mode of win For each mode of win, the score begins moderately (around 60%) but after some simulations, the score decreases until 1.5% for one case. However, against all odds, the semeai is solved as a win.

to 1.5% before to be solved as a win. The semeai was easy to solve (few nodes are enough) but the bias in the resolution was globally bad: a lot of simulations have been defeats. In contrary, a winrate of 99% can be seen on losing semeais. More badly, sometimes the resolution is false.

3.5.6 Failure cases

We have presented semeais in which the solver works. However, sometimes GoldenEye fails. The first reason is a bad detection or no detection. We have seen too that GoldenEye evaluation (i.e. winrate) can be false. He is unable to see the solution because the solution is too hard. But sometimes, a full solving can be even wrong. The *horizon* may be too short. Some other cases can occur. Fig. 3.24 is 3 situations in which GoldenEye proposes a bad full resolution of semeai because of a bad prunings or a bug.

In real games, results of GoldenEye are less good. Semeais are more complicated and rare. The problem is often a combination of different problems such as connection between 2 groups and semeai. When a semeai appears, the semeai status is already clear or it is not interesting for the victory. In general, either all groups are alive and there is no semeai or no group is alive and we can consider that all groups in one semeai and the solution is very complex because the question is not about 2 groups but different set of groups which survive at the end of the game. It is perhaps one of the reasons that the solver has not been worked with MoGo. No very good results has

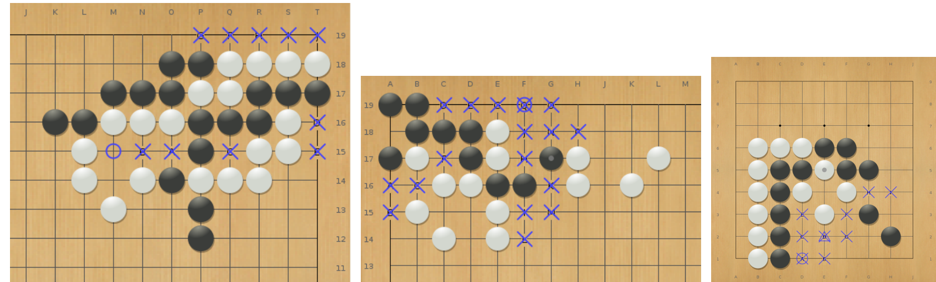


Figure 3.24: Failed full resolutions. **Left:** Black to play. GoldenEye does not consider the winning move M15 because the move is too far from the semeai. The semeai is solved as a loss; no move is given. **Center:** White to play. GoldenEye does not consider the move H19 (after White F19 and Black F18), the unique move winning the semeai, because of a rule which prevents expansion of seemingly useless moves. This rule is bad in this situation. The semeai is solved as a loss; no move is given. **Right:** Black to play. Another variant of ”pear and ant“ in 9×9. After the sequence E2-D2-D1, GoldenEye considers only the move E4. E1 is the good move in this situation. Maybe it is a bug or a misunderstood situation by GoldenEye. The semeai is solved as a win; but the given winning move is wrong (E2 instead of D1).

been found in real game.

3.5.7 Discussion

The first version of the algorithm solves efficiently small problems with few Monte-Carlo simulations. This point was one of the first motivations of the algorithm.

Even if at constant time MoGo and GoldenEye seem to be the same strength for playing semeais, at fixed number of simulations GoldenEye plays more often the good move than MoGo (Tab. 3.2 and Tab. 3.2).

Unfortunately, we have tested our program on more and more difficult problems; this version of the algorithm becomes quickly unusable on these problems (Fig. 3.17). Moreover, the winrate can be in disagree with the result given by the full resolution (Fig. 3.23). This statement is more true on harder and harder problems. Controlling efficiently the local search is an option.

For restricting the search to a small part of the goban, a lot of tricks have been developed such as

- an *horizon* not to big

- aggressive prunings
- efficient Monte-Carlo simulations with a lot of expertises.
- efficient small tactical solvers for detecting more rapidly who wins the semeai before the capture of one group (e.g. Fig. 3.10)

Despite a lot of human expertises and efficient small tactical solvers, simulations tends to go to the center of the goban (e.g. at left in Fig. 3.21). Moreover, we have seen some cases where the solving was false because of a bad pruning (Fig. 3.24) or a horizon too small (Tab. 3.7). The complexity of the problem is sometimes beneficial in order to converge on the good solution. We have already seen a semeai solved as a win with a bad move (at right in Fig. 3.24) and on a bigger goban, the semeai is not solved but its estimation is very optimistic and the answer linked to this estimation is the good move (at right in Fig. 3.22).

Instead of controlling the locality of the search, another option is to use statistics for guiding the building of the tree. Some new statistic tools have been elaborated. Thanks to the heuristic *inhibition*, GoldenEye can fully solve more difficult problems; one reason is: for a given time, we can do more simulations (Fig. 3.17). The heuristic related to the exploitation term allows to concentrate on interesting branches of the tree. In addition, the heuristic *inhibition* allows to eliminate more rapidly newly unpromising branches. Against a lot of human expertises, some statistic tools can be used. The heuristic *gcmc* seems to be unstable but is a generic solution for finding moves or sequence of moves forgotten in the human expertise. The solving is more efficient thanks to these heuristics.

For solving all cases, we should again and again add expertises and rules giving a more and more complex engine to maintain. However, GoldenEye is able to solve a lot of semeais whose some are difficult (level $>$ 1dan).

3.6 Conclusion

We have proposed a solver of semeais (GoldenEye) whose goals are:

- produce a full resolution of an open semeai in order to be used and studied by humans
- correct weaknesses of other go engines.

Unlike MoGo, GoldenEye produces proof (a full resolution of a problem). Without GoldenEye, I would have never known that there was a ko in a real

game (at right in Fig. 3.14) or I would have never created a small semeai finishing by a seki (at left in Fig. 3.14).

The solver must be automatic; which implies 2 steps: (i) detection and (ii) resolution.

The user (or the engine) does not show where semeai(s) is/are located; the solver must be able to find by itself a semeai(s). We have proposed a statistical detection by using random simulations. This part is very short in execution time and efficient on invented problems[5] but much less efficient in real game[6].

For solving semeais, we have proposed an algorithm based on a local search. The algorithm is a combination between A^* and Monte-Carlo Tree Search. A leaf is selected like in A^*, but the algorithm does not browse the whole tree for the selection of the leaf. Like Monte-Carlo Tree Search, statistics and a term of exploitation are used and Monte-Carlo simulations are performed.

The heuristic *inhibition*, a mechanism for pruning temporary, is the improvement which has brought the more good things in terms of time and memory; some difficult semeais become feasible and fully solvable by GoldenEye. Therefore, this heuristic is a very interesting hint for improving Monte-Carlo Tree Search or browsing graph algorithms in particular A^*. However, the heuristic *inhibition* is based on a boolean evaluation: the win or the loss of the simulation. When the evaluation is not a boolean, maybe we can decide to have a rule which transforms a number to a boolean (e.g. the evaluation is greater than a threshold).

As such, GoldenEye was unable to solve some particular cases of semeai. We have added different modes for winning a semeai and a mechanism of switching between them: the hierarchical resolution.

In further works, GoldenEye shall be compared with Proof Number Search (PNS) and its variants and shall be improved. GoldenEye does not use the number of children or the number of children remaining to prove. Thus, the concept of proving of PNS could be introduced as a bonus in the expertise or when GoldenEye evaluates the relevance of a branch from where a currently evaluated leaf comes. Introducing some works such as mathematical rules [Hunter, 2003] for improving the tactical solver which evaluates the end of a semeai is probably a way for making more efficient GoldenEye. Even if the semeai is completely solved, the solution is not sure because of prunings. Another algorithm which uses extension of moves such as [Cazenave, 2001] could be applied after the full resolution of the semeai. The algorithm will

[5] generally, these situations are devoid of useless stones

[6] generally complex situations, where groups are nested into each other, are generated

check the validity of the resolution by considering secondary moves and when necessary, it will correct the resolution.

GoldenEye does not use hash tables and is not parallelized. Unfortunately, it seems difficult to use hash tables and the parallelization. Maybe, with a hierarchical solving, one processor can be dedicated to a way of winning. However, they are hints for improving GoldenEye.

GoldenEye has been extended for solving death/life problems, another weakness of MoGo. Thanks to random simulations, GoldenEye uses the percentage of times that a stone is alive for determining a group in danger (detection). Then, GoldenEye searches to know if the group can live or not (resolution). The resolution is globally the same. There are some differences such as the expertise (patterns for killing/building an eye). A case where MoGo has played badly in an important game is described in Fig. 3.25; GoldenEye correctly analyzes the death/life problem. Since I have the feeling that there are more death/life problems than semeais, a further work could be to try a cooperation between MoGo and GoldenEye in a version death/life problems.

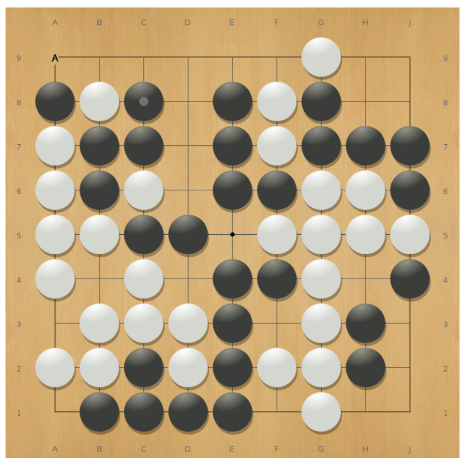


Figure 3.25: White to play. A critical situation where MoGo lost a game against a 9P professional player. The life of the white group G1 is in pending. MoGo (White) has already lost the game. However, even if we put a komi which allows to White to win, MoGo plays a bad move (H1) and then as Black, MoGo resigns. GoldenEye adapted to the death/life problems plays a good move (i.e. J2) and after the bad move played by MoGo (i.e. A9) in the game, GoldenEye plays again J2, the move, played by the 9p player, which kills the white group G1. Note that both groups G1 and H2 are an example of false semeai. Black stones are stones of sacrifice. Even after being captured, they prevent to the white group G1 to live.

Algorithm 11 The hierarchical resolution of a semeai. Based on the percentage of success, GoldenEye switches between the different ways to win. One own tree is dedicated to a way to win. GoldenEye is firstly optimist and tries to win without condition. The moment where GoldenEye can decide to switch or not between trees is called a possible switch. The algorithm has 2 parameters: $nbPacks$ is the number of possible switches and $nbSimByPacks$ is the maximal number of simulations done between two possible switches. When no full resolution, the total number of simulations is $nbPacks \times nbSimByPacks$ spread over the different trees.

Function $hierarchicalSolv(nbPacks, nbSimByPacks)$

Let tr_w, tr_{wk}, tr_{dk} and tr_{lk} 4 static variables containing respectively the tree "win without ko", "win with ko", "draw" and "loss with ko"
$tr_{cur} = tr_w$
for $iter \in 1..nbPacks$ **do**
  $BuildTree(tr_{cur}, nbSimByPacks)$
  Let $winrate$ the winrate of the root of the current tree
  if $winrate > 66\%$ {we switch the effort on a more ambitious goal} **then**
    if $tr_{cur} == tr_{wk} \wedge tr_w$ is not solved **then**
      $tr_{cur} = tr_w$
    else
      if $tr_{cur} == tr_{dk} \wedge tr_{wk}$ is not solved **then**
        $tr_{cur} = tr_{wk}$
      else
        if $tr_{cur} == tr_{lk} \wedge tr_{dk}$ is not solved **then**
          $tr_{cur} = tr_{dk}$
  if $winrate < 33\%$ {we switch the effort on a less ambitious goal} **then**
    if $tr_{cur} == tr_w \wedge tr_{wk}$ is not solved **then**
      $tr_{cur} = tr_{wk}$
    else
      if $tr_{cur} == tr_{wk} \wedge tr_{dk}$ is not solved **then**
        $tr_{cur} = tr_{dk}$
      else
        if $tr_{cur} == tr_{dk} \wedge tr_{lk}$ is not solved **then**
          $tr_{cur} = tr_{lk}$
  if tr_{cur} is solved **then**
    break

Part II

Offline learning of Simulations : Direct Policy Search

In online learning, the learning is performed in real time during the simulations, whereas in offline mode the learning is performed once and for all in a separate training phase.

Monte-Carlo Tree Search can be used in mode offline. The algorithm has been used for building an opening book of MoGo [Gaudel et al., 2010]. But for building an opening book, transitions should be deterministic and above all, the initial state should be always the same. It's not the case in MASH applications.

In this part, we will describe 2 offline learning of simulations. In a first chapter, we will see the RBGP algorithm and in a second chapter the CluVo+GMCTS algorithm.

Chapter 4

Building a Policy by Genetic Programming (RBGP)

Adding expert knowledge is boring, uneasy and biased by human ideas. We wish to automatize the generation of prior knowledge, by randomly drawing new patterns and by testing them through bandits and races. This idea leads to a new algorithm called Racing-Based Genetic Programming.

This chapter is heavily based on Progress-rate in Noisy Genetic Programming for choosing λ (EA2011) [Hoock and Teytaud, 2011].

We also use material from

- Intelligent Agents for the Game of Go (CIM Journal 2010) [Hoock et al., 2010]
- Bandit-based Genetic Programming (EuroGP 2010) [Hoock and Teytaud, 2010]

4.1 Introduction

Genetic programming (GP) consists in automatically building a program for solving a given task. The fitness function quantifies the efficiency of a program for this task.

Non-regression testing consists in testing each new version of a program, in order to check that it is at least as good as the previous version. Non-regression testing is very difficult when the fitness function is noisy, as it is uncertain. Statistical tests have to take into account the high number of tests - this is a non-trivial issue. The present work originates in the tedious non-regression testing in a highly collaborative development, often poorly performed by humans. Individual developers don't care of the global risk

due to multiple testings (Multiple Simultaneous Hypothesis Testing). Tests had to be automatized, simultaneously with the development of improvements in a search module by automatic means in order to get rid of human biases in such developments.

Noisy GP is an important problem, related to many applications. In particular, direct policy search with symbolic controllers is noisy genetic programming, as well as the design of sorting algorithms faster on average on a huge number of instances. Bandits have been investigated very early for this problem[Koza, 1992, Holland, 1973]. However, bandits address the problem of the load balancing between different possible mutations, but not the validation of the selection, i.e. the halting criterion for the offspring evaluation. Races[Mnih et al., 2008] have the double advantage of considering the load balancing and the statistical validation. [Mnih et al., 2008] considers the case in which we look for all good mutations. We might, in GP, be more interested in finding one good mutation, as μ good mutations do not necessarily cumulate to one better mutation. Yet, the case of selecting several good mutations is important also, but it was shown in [Hoock and Teytaud, 2010] that there are frameworks in which $\mu = 1$ is more relevant and we focus on this case. In the case of continuous optimization, races have been theoretically analyzed as a tool for ensuring optimal rates (within log factors) in [Rolet and Teytaud, 2010]. A partial theoretical analysis, essentially ensuring consistency, was proposed for GP in [Hoock and Teytaud, 2010]: we here extend this work by an analysis of progress rate. Importantly, the theoretical analysis proposes a choice for λ by optimisation of the progress rate.

4.2 Framework and notations

Consider a program P, and a set M of possible mutations; let $P + m$ be the program after application of mutation m. Assume that the fitness is stochastic; $f(P+m)$ is a random variable with values in $[-1, 1]$. In all the chapter we consider maximization. We consider $(1+\lambda)$ genetic programming algorithms as in Alg. 12; this is a simple $(1 + \lambda)$-algorithm[Hoock and Teytaud, 2010].

All this section is written assuming that the range of fitness values is bounded in absolute value by 1, i.e. when we compute the fitness of a point we get an answer between -1 and 1. We assume that the fitness is unbiased, *i.e.* the expected value of the measurements is equal to the real fitness. This is standard in *e.g.* Monte-Carlo sampling, or in some randomized forms of Quasi-Monte-Carlo samplings.

Hoeffding and Bernstein bounds quantify the effect of noise on empirical averages. In noisy cases, the fitness $fitness(x)$ of a point x is unknown:

Algorithm 12 One-plus-lambda Racing-Based Genetic Programming (RBGP). We assume $c > 1$ so that K is finite.

RBGP algorithm.
Parameters: a risk level δ, a pop. size λ, a coefficient $c > 1$, a threshold $\epsilon > 0$.
Let $K = 1/(\sum_{n\geq 1} 1/n^c)$
Let P be a program to be optimized
Let M be a set of possible mutations on P
Let $n \leftarrow 0$
while There is some time left **do**
  while no mutation accepted **do**
    Let $n \leftarrow n + 1$
    Randomly draw $pop = \{m_1, \ldots, m_\lambda\}$ in M
    Perform a Bernstein race with risk $\delta_n = K\delta/n^c$ on *pop* and threshold ϵ
    if A mutation m is selected **then**
      $P \leftarrow P + m$

we have only access to $y_1, \ldots, y_k$, k real numbers, if $fitness(x)$ has been approximated k times; a natural estimate is $\widehat{fitness}(x) = \frac{1}{k}\sum_{i=1}^{k} y_i$.

Hoeffding or Bernstein bounds state that with probability at least $1 - \delta$, $|fitness(x) - \widehat{fitness}(x)| < deviation$, where $deviation$ is

$$deviation_{Hoeffding} = \sqrt{\log(2/\delta)/k}. \tag{4.1}$$

[Audibert et al., 2006, Mnih et al., 2008] has shown the efficiency of using Bernstein's bound instead of Hoeffding's bound, in some settings. The deviation term is then:

$$deviation_{Bernstein} = \sqrt{\hat{\sigma}^2 2\log(3/\delta)/n} + 3\log(3/\delta)/k. \tag{4.2}$$

This equation depends on $\hat{\sigma}^2$, the empirical variance of the measurements $y_1, \ldots, y_k$. An interesting feature of this equation is that using $\hat{\sigma}^2$ and not the real variance σ^2 is not an approximation: the inequality is rigorous with $\hat{\sigma}^2$ (contrarily to many asymptotic confidence intervals).

The Bernstein race is a typical one, following [Mnih et al., 2008], except that we want to validate one and only one mutation, because in our framework it is known [Hoock and Teytaud, 2010] that two good mutations do not

necessarily cumulate, in the sense that sometimes:

$$\begin{aligned} \mathbb{E}f(P+m_1) &> \mathbb{E}f(P) \\ \mathbb{E}f(P+m_2) &> \mathbb{E}f(P) \\ \text{and yet } \mathbb{E}f(P+m_1+m_2) &< \mathbb{E}f(P). \end{aligned}$$

In a $(1+\lambda)$-GP, we look for a mutation which provides a better success rate than the current parent, i.e. a higher fitness. However, we will here translate the fitness by substracting the fitness of the parent; i.e. with a fitness function with values in $[-1,1]$ and a parent z, we define

$$fitness'(x) = (fitness(x) - fitness(z))/2;$$

this just doubles the number of calls to the fitness. Therefore, in the Bernstein race, and without loss of generality, we look for a mutation which gives a positive fitness, instead of a mutation with $fitness(x) > fitness(z)$. The Bernstein race is as presented in Alg. 13, and the *computeBounds* function

Algorithm 13 Bernstein race for selecting good individuals in a population *pop*. M is the complete set of arms (global variable; see Alg. 12).

```
BernsteinRace(pop, δ, ε)
while pop ≠ ∅ do
  for all m ∈ pop do
    Let n be the number of simulations of mutation
    m.
    Simulate m n more times (i.e. now m has been
    simulated 2n times).
        //this ensures nbTests(m) = O(log(n(m)))
    computeBounds(m, M, δ)
    if lb(m) > 0 then
      Return individual corresponding to mutation
      m.
    else if ub(m) < ε then
      pop = pop \ {m}                  m is discarded.
Return "no good individual in the offspring!"
```

is defined as shown in Alg. 14.

where # denotes the cardinal operator.

Important properties of Bernstein's races as above, and which hold with probability at least $1-\delta$, are the followings ([Mnih et al., 2008]).

Properties of Bernstein races.

Algorithm 14 Function for computing a lower and an upper bound on arm m with confidence $1-\delta$, where *pop* is the complete set of arms.

Function $computeBounds(m, pop, \delta)$
Static internal variable: $nbTests(m)$, initialized at 0.
Let $n = nbTests(m)$.
Let r be the total reward over those n simulations.
$\delta_m = \delta/(\#pop \times \pi^2 nbTests(m)^2/6)$
$nbTests(m) = nbTests(m) + 1$
$lb(m) = \frac{r}{n} - deviation_{\text{Bernstein}}(\delta_m, n)$.
$ub(m) = \frac{r}{n} + deviation_{\text{Bernstein}}(\delta_m, n)$.

- *Property 1. The number of evaluations in a Bernstein race*
  - *with population size $\#pop$;*
  - *with parameters δ and ϵ;*
  - *with a population such that, with $p = \max(\epsilon, \max_{m\in pop}\mathbb{E}f(P+m))$, all expected fitness values are in $[-\Theta(\epsilon), p]$ (possibly all fitness values ≤ 0, case in which there's no good mutation).*

  is

$$Time(pop, \delta, \epsilon) = \Theta(Th(\#pop, \delta, \epsilon, p))$$

  where

$$Th(\#pop, \delta, \epsilon, p) = \left(\log(\frac{\#pop \log(1/p)}{\delta})\right) \max\left(\sigma^2/p^2, 1/p\right) \quad (4.3)$$

  with: σ^2 is an upper bound on the variance of fitness values:

$$\sigma^2 = \sup_m \mathbb{E}(f(P+m) - \mathbb{E}f(P+m))^2.$$

- *Property 2. If a mutation is selected, then it has fitness > 0.*
- *Property 3. If there is a mutation with fitness $\geq \epsilon$ then the race will return a mutation.*

We will here focus on the case $\sigma^2 = \Theta(1)$ (which corresponds to the applicative framework in which variance does not decrease; this is consistent

with many applicative fields, in particular when optimizing the parameters of a strategy with optimal success rate $< 100\%$ - yet, other cases might be considered in a future work), and therefore Eq. 4.3 can be replaced by Eq. 4.4 without significant loss:

$$Th(\#pop, \delta, \epsilon, p) = \left(\log(\frac{\#pop \log(1/p)}{\delta}) \right) /p^2. \tag{4.4}$$

Properties of RBGP. *Eq. 4.4 and properties of Bernstein races above lead to the following properties of RBGP, which hold (all simultaneously) with probability at least $1 - \delta$: if there are*

- $n-1$ *iterations of RBGP in which all mutations have expected fitness* $\in [-\epsilon, 0]$*;*
- *and thereafter,* 1 *iteration of RBGP with at least one mutation with expected fitness* p *and all other mutations with expected fitness in* $[-\epsilon, p]$*.*

then

- *Property A: RBGP will not return a bad mutation (i.e. a mutation with fitness* ≤ 0*);*
- *Property B: If there is a mutation with fitness* $\geq \epsilon$ *then a mutation will be found;*
- *Property C: And in that case the halting time is at most*

$$O\left(\sum_{i=1}^{n-1} Th(\lambda, K\delta/i^c, \epsilon, p) + Th(\lambda, K\delta/n^c, p) \right) \tag{4.5}$$

Eq. 4.5 will be central in the progress rate analysis below.

4.3 Experiments with RBGP algorithm

Life is a Game of Go in which rules have been made unnecessarily complex, according to an old proverb. As a matter of fact, Go has very simple rules, is very difficult for computers, is central in education in many Asian countries (part of school activities in some countries) and has NP-completeness properties for some families of situations[Crasmaru, 1999], and PSPACE-hardness for others[Lichtenstein and Sipser, 1980], and EXPTIME-completeness for some versions [Robson, 1983]. It has also been chosen as a testbed for artificial intelligence by many researchers. The

main tools, for the game of Go, are currently MCTS/UCT (Monte-Carlo Tree Search, Upper Confidence Trees); these tools are also central in many difficult games and in high-dimensional planning. An example of nice Go game, won by MoGo as white in 2008 in the GPW Cup, is given in Fig. 4.1 (left). Since these approaches have been defined [Chaslot et al., 2006,

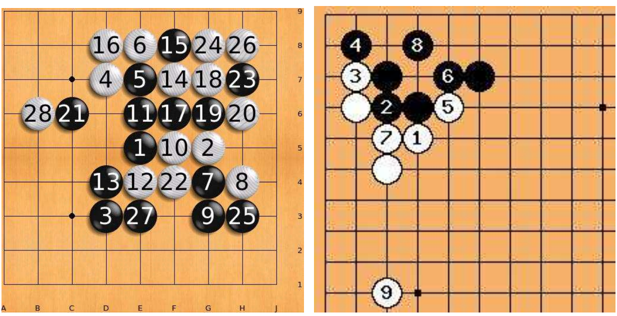


Figure 4.1: Left: A decisive move (number 28) played by MoGo as white, in the GPW Cup 2008. Right: An example from Senseis of good large pattern in spite of a very bad small pattern. The move 2 is a good move.

Coulom, 2006, Kocsis and Szepesvari, 2006], several improvements have appeared like First-Play Urgency [Wang and Gelly, 2007], Rave-values [Bruegmann, 1993, Gelly and Silver, 2007] (see `ftp://ftp.cgl.ucsf.edu/pub/pett/go/ladder/mcgo.ps` for B. Bruegman's unpublished paper), patterns and progressive widening [Coulom, 2007, Chaslot et al., 2007], better than UCB-like (Upper Confidence Bounds) exploration terms [Lee et al., 2009], large-scale parallelization [Gelly et al., 2008, Chaslot et al., 2008, Cazenave and Jouandeau, 2007, Kato and Takeuchi, 2008], automatic building of huge opening books [Audouard et al., 2009]. Thanks to all these improvements, our implementation MoGo already won even games against a professional player in 9x9 (Amsterdam, 2007; Paris, 2008; Taiwan 2009), and recently won with handicap 6 against a professional player (Tainan, 2009), and with handicap 7 against a top professional player, Zhou Junxun, winner of the LG-Cup 2007 (Tainan, 2009). Besides impressive results for the game of Go, MCTS/UCT have been applied to non-linear optimization [Auger and Teytaud,], optimal sailing [Kocsis and Szepesvari, 2006], active learning [Rolet et al., 2009]. The formula used in the bandit is incredibly complicated, and it is now very hard to improve the current best formula [Lee et al., 2009].

Here we will consider only mutations consisting in adding patterns in our program MoGo. Therefore, accepting a mutation is equivalent to accepting a pattern. A mutation is a pattern with a coefficient (Fig. 4.2). The coefficient

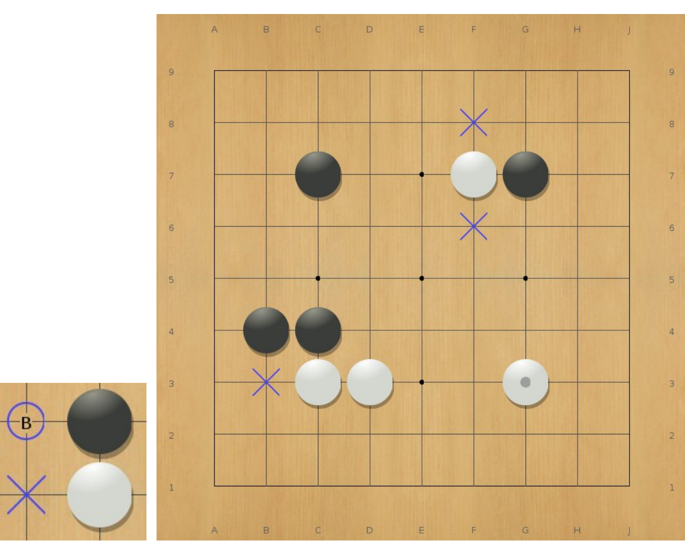


Figure 4.2: A random pattern of size 2x2 for black (**Left**) and its matchings (**Right**). The associated coefficient is either 0.75 for good shapes or -0.75 for bad shapes. At left, the blue cross is the considered decision, the letter B in the blue circle means either a black stone or an empty location. At right, blue crosses indicate matched locations. The matching considers rotation and symmetry of the pattern. For white, the pattern is reversed (e.g. the letter B in the blue circle means a white stone or an empty location).

has been always fixed to 0.75 for good shapes or -0.75 for bad shapes. The size of a pattern has been limited to 5×5. When the pattern is bigger, it matches very rarely and the impact of the pattern becomes negligeable. The introduction of one mutation in MoGo is exactly the same as the introduction of one pattern (e.g. Fig. 2.2) described in Section 2.2. We experiment random patterns for biasing UCT. The reader interested in the details of this is referred to [Lee et al., 2009]. Our patterns contain jokers, black stones, empty locations, white stones, locations out of the goban, and are used as masks over all the board: this means that for a given location, we consider patterns like “there is a black stone at coordinate +2,+1, a stone (of any color) at coordinate +3,0, and the location at coordinate -1,-1 is empty”. This is a very particular form of genetic programming. We consider here the automatic generation of patterns for biasing the simulations in 9x9 and 19x19 Go. Please note that: (1) When we speak of good or bad shapes here, it is in the sense of ”shapes that should be more simulated by a UCT-like algorithm”, or ”shapes that should be less simulated by a UCT-like algorithm”. This is not necessarily equivalent to “good” or “bad” shapes for

human players (yet, there are correlations). (2) In 19x19 Go, MoGoCVS is based on tenths of thousands of patterns as in [Chaslot et al., 2007]. Therefore, we do not start from scratch. A possible goal would be to have similar results, with less patterns, so that the algorithm is faster (the big database of patterns provides good biases but it is very slow). (3) In 9x9 Go, there are no big library of shapes available; yet, human expertise has been encoded in MoGo, and we are far from starting from scratch. Engineers have spent hundreds of hours manually optimizing patterns. The goals are both (i) finding shapes that should be more simulated (ii) finding shapes that should be less simulated.

Section 4.3.1 presents our experiments for finding good shapes in 9x9 Go. Section 4.3.2 presents our experiments for finding bad shapes in 9x9 Go. Section 4.3.3 presents our unsuccessful experiments for finding both good and bad shapes in 19x19, from MoGoCVS and its database of patterns as in [Chaslot et al., 2007]. Section 4.3.4 presents results on MoGoCVS with patterns removed, in order to improve the version of MoGoCVS without the big database of pattern.

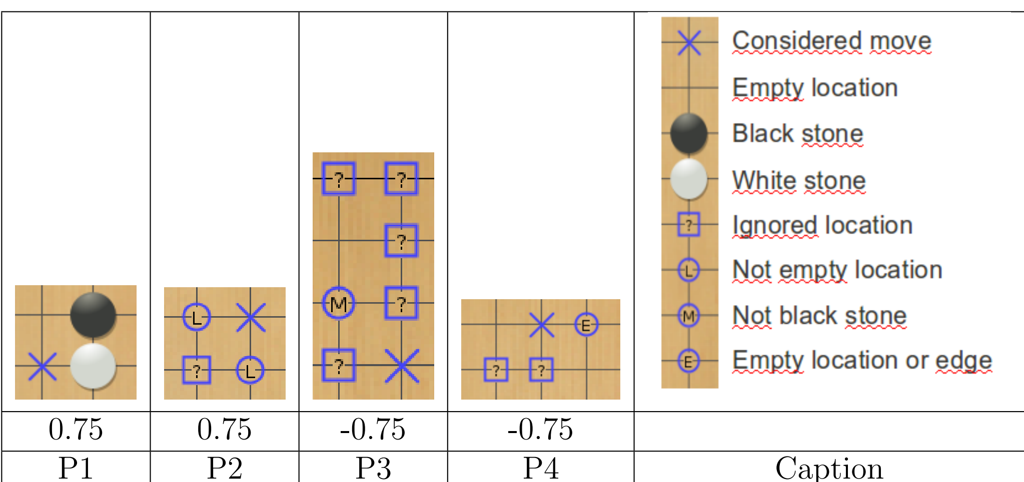
4.3.1 Finding good shapes for simulations in 9x9 Go

Here the baseline is MoGo CVS. All programs are run on one core, with 10 000 simulations per move. All experiments are performed on Grid5000. The selection rule, not specified in RBGP, is the upper bound as in UCB[Lai and Robbins, 1985, Auer et al., 2002]: we simulate s such that $ub(s)$ is maximal. We here test modifications which give a positive bias to some patterns, *i.e.* we look for shapes that should be simulated more often.

For each iteration, we randomly generate some individuals, and test them with the RBGP algorithm. For the three first iterations, 10 patterns were randomly generated; the two first times, one of these 10 patterns was validated; the third time, no pattern was validated. Therefore, we have three version of MoGo: MoGoCVS, MoGoCVS+P1, and MoGoCVS+P1+P2, where P1 is the pattern validated at the first iteration and P2 is the pattern validated at the second iteration. We then tested the relative efficiency of these MoGos as follows:

Tested code	Opponent	Success rate
MoGoCVS + P1	MoGoCVS	$50.78\% \pm 0.10\%$
MoGoCVS + P1 + P2	MoGoCVS +P1	$51.2\% \pm 0.20\%$
MoGoCVS + P1 + P2	MoGoCVS	$51.9\% \pm 0.16\%$

We also checked that this modification is also efficient for 100 000 simulations per move, with success rate $52.1 \pm 0.6\%$ for MoGoCVS+P1+P2 against



0.75	0.75	-0.75	-0.75	
P1	P2	P3	P4	Caption

Table 4.1: Mutations P1, P2, P3 and P4 found by RBGP in 9x9. The pattern P1 is a famous human pattern in the game of Go; the name of this pattern is the Hane and it is considered as a good shape. The pattern P2 favours the moves beside 2 stones. The pattern P3 avoids the moves on the fourth line. The pattern P4 avoids the moves in an empty zone.

MoGoCVS. There was no pattern validated during the third iteration, which was quite expensive (one week on a cluster). We therefore switched to another variant; we tested the case $|S_0| = 1$, *i.e.* we test one individual at a time.We launched 153 iterations with this new version. There were therefore 153 tested patterns, and none of them was validated.

4.3.2 Finding bad shapes for simulations in 9x9 Go

We now switched to the research of negative shapes, *i.e.* patterns with a negative influence of the probability, for a move, to be simulated. We kept $|S_0| = 1$, i.e. only one pattern tested at each iteration. There were 173 iterations, and two patterns P3 and P4 were validated. We verified the quality of these negative patterns as follows, with mogoCVS the version obtained in the section above:

Tested code	Opponent	Success rate
MoGoCVS + P1 + P2 + P3	MoGoCVS + P1 + P2	$50.9\% \pm 0.2\%$
MoGoCVS + P1 + P2 + P3	MoGoCVS	$52.6\% \pm 0.16\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS + P1 + P2 + P3	$50.6\% \pm 0.13\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS	$53.5\% \pm 0.16\%$

This leads to an overall success of 53.5% against MoGoCVS, obtained by RBGP. The four mutations found by RBGP are given in Tab. 4.1.

4.3.3 Improving 19x19 Go with database of patterns

In 19x19 Go, all tests are performed with 3500 simulations per move. Here also, we tested the case $|S_0| = 1$, *i.e.* we test one individual at a time. We tested only positive biases. The algorithm was launched for 62 iterations. Unfortunately, none of these 62 iterations was accepted. Therefore, we concluded that improving these highly optimized version was too difficult. We switched to another goal: having the same efficiency with faster simulations and less memory (the big database of patterns strongly slowers the simulations and takes a lot of simulations), as discussed below.

4.3.4 Improving 19x19 Go without database of patterns

We therefore removed all the database of patterns; the simulations of MoGo are much faster in this case, but the resulting program is nonetheless weaker because simulations are far less efficient (see *e.g.* [Lee et al., 2009]). Fig. 4.1 (right) presents a known (from Senseis `http://senseis.xmp.net/?GoodEmptyTriangle#toc1`) difficult case for patterns: move 2 is a good move in spite of the fact that locally (move 2 and locations at the east, north, and north east) form a known very bad pattern (termed empty triangle), termed empty triangle, and is nonetheless a good move due to the surroundings.

We keep $|S_0| = 1$, 127 iterations. There were six patterns validated, validated at iterations 16, 22, 31, 57, 100 and 127. We could validate these patterns Q1,Q2,Q3,Q4,Q5,Q6 as follows. MoGoCVS+AE means MoGoCVS equipped with the big database of patterns extracted from games between humans.

Tested code	Opponent	Success rate
MoGoCVS + Q1	MoGoCVS	$50.9\% \pm 0.13\%$
MoGoCVS + Q1 + Q2	MoGoCVS + Q1	$51.2\% \pm 0.28\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS + Q1 + Q2	$56.7\% \pm 1.50\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS + Q1 + Q2 + Q3	$52.1\% \pm 0.39\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS + Q1 + ... + Q4	$51.1\% \pm 0.20\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS + Q1 + ... + Q5	$54.1\% \pm 0.78\%$
MoGoCVS + Q1 + Q2	MoGoCVS	$53.4\% \pm 0.50\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS	$57.3\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS	$59.4\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS	$58.6\% \pm 0.49\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS	$61.7\% \pm 0.49\%$
MoGoCVS	MoGoCVS + AE	$26.6\% \pm 0.20\%$
MoGoCVS + Q1	MoGoCVS + AE	$27.5\% \pm 0.49\%$
MoGoCVS + Q1 + Q2	MoGoCVS + AE	$28.0\% \pm 0.51\%$
MoGoCVS + Q1 + Q2 + Q3	MoGoCVS + AE	$30.9\% \pm 0.46\%$
MoGoCVS + Q1 + ... + Q4	MoGoCVS + AE	$32.1\% \pm 0.43\%$
MoGoCVS + Q1 + ... + Q5	MoGoCVS + AE	$30.9\% \pm 0.46\%$
MoGoCVS + Q1 + ... + Q6	MoGoCVS + AE	$32.8\% \pm 0.47\%$

An important property of RBGP is that all validated patterns are confirmed by these independent experiments. We see however that in 19x19, we could reach roughly 30% of success rate against the big database built on human games (therefore our RBGP version uses far less memory than the other version); we will keep this experiment running, so that maybe we can go beyond 50 %. Nonetheless, we point out that we already have 60 % against the version without the database, and the performance is still increasing (improvements were found at iterations 16,22,57,100,122,127, with regular improvements - we have no plateau yet) - therefore we successfully improved the version without patterns, which is lighter (90% of the size of MoGoCVS is in the database).

4.4 Progress rate of Racing-based GP

λ has been fixed to 1 on precedent experiments. But maybe, a better value can be found. We propose a study of progress in Noisy Genetic Programming for choosing λ.

Progress rate theory is classical in continuous optimization[Auger and Hansen, 2006]. The progress rate will be here adapted to noisy GP. We will consider T (formally defined below), the number of fitness evaluations before finding a good mutation. The progress rate is then defined as $1/T$, the inverse time before finding a good mutation. We will then, following the continuous case, choose parameters that optimize

the progress rate. Note that here, the progress rate is a success probability; this is equivalent to classical criteria [Beyer, 2001] only in restricted settings (see assumptions in Section 4.4).

We consider that randomly drawn mutation have fitness[1]:

- $q > 0$ with probability $f > 0$;
- ≤ 0 otherwise.

We will study the behavior of $(1+\lambda)$-GP depending on q and f. The other relevant parameters are:

- λ (the offspring size);
- ϵ, the threshold of the Bernstein races (see Alg. 12);
- c, a parameter used in Alg. 12 and which is of moderate importance as shown below.

The different cases under analysis are (i) $q = \epsilon$ (ii) $q >> \epsilon$. We will investigate the running time, i.e. the number T of fitness evaluations before finding a good mutation with probability at least $1 - 2\delta$. It is already known[Hoock and Teytaud, 2010] that with probability at least $1 - \delta$,

- if there is a good mutation ($fitness \geq q$), it will be found;
- no bad mutation ($fitness < 0$) will be selected. Possibly, the race replies that it did not find any good mutation.

We will show (i) that a too mild rejection threshold ϵ has bad effects (section 4.4.1); (ii) that a good tuning provides significant improvements (section 4.4.2); (iii) that there is a parameter free version with ensures that there's no infinite loop (section 4.4.3).

4.4.1 Too mild rejection: $q >> \epsilon$

Here, the precision required for rejecting a bad mutation is very small in front of the quality of good mutations. The following result shows that in that case the choice of λ is crucial: λ should be of the order

$$\log(\delta)/\log(1-f) \tag{4.6}$$

Theorem 1: rejection pressure too small (ϵ too small). *Assume that $q > \epsilon$. Then,*

[1] We recall that fitnesses are translated as explained in Section 4.2 so that the parent has fitness 0 and therefore "good" mutations are mutations with value > 0.

- *If* $\lambda \geq \lceil \frac{\log(\delta)}{\log(1-f)} \rceil$*, then* $T = \Theta(\lambda \log(\lambda/(q\delta))/q^2)$.
- *If* $\lambda \leq \frac{\log(1/2)}{\log(1-f)}$*, then* $T = \Omega(\log(1/(f^2\epsilon\delta)/\epsilon^2)$.

The proof is in [Hoock and Teytaud, 2011].□

Remark: In the case $q >> \epsilon$, this theorem implies that $\lambda \geq \lceil \frac{\log(\delta)}{\log(1-f)} \rceil$ is much better than $\lambda \leq \frac{\log(1/2)}{\log(1-f)}$.

4.4.2 Well tuned parameters: $q = \epsilon$

Theorem 2: population size with well tuned parameters. *Assume that* $q = \epsilon > 0$*. Then,*

$$T = \Theta(\frac{\lambda \log(\lambda \log(\frac{1}{\epsilon})/\delta)}{(1-(1-f)^\lambda)\epsilon^2}). \tag{4.7}$$

The proof is in [Hoock and Teytaud, 2011].□

Remark: Theorem 2 provides an evaluation of the cost

$$T = \Theta(\frac{\lambda \log(\lambda \log(\frac{1}{\epsilon})/\delta)}{(1-(1-f)^\lambda)\epsilon^2}).$$

If we neglect all logarithmic factors,

- this is linear as a function of λ if $\lambda \simeq 1/n$, i.e. the overall cost is linear as a function of $1/(f\epsilon^2)$.
- this is linear as a function of $1/(f\epsilon^2)$ if $\lambda = 1$.

We therefore see that the population size does not matter a lot when the parameter ϵ is chosen so that it nearly matches q.

4.4.3 No prior knowledge

The analysis above has the weakness that it requires some knowledge on the fitnesses of possible mutations, in order to choose ϵ, the parameter used in the rejection rule. The main risk is a too strong rejection: $q << \epsilon$ would lead to the rejection of the best mutations. We here investigate results possible with no knowledge at all.

Theorem 3: population size with $q << \epsilon$. *Consider RBGP with the Bernstein race as in Alg. 15. Then, with probability* $1-\delta$*, no bad mutation is accepted, and if the probability of a good mutation is positive, then*

$$T < \infty. \tag{4.8}$$

The proof is in [Hoock and Teytaud, 2011].□

Algorithm 15 Variant of Bernstein race: a pattern is rejected as soon as its average fitness is below 0.

```
BernsteinRace(pop, δ, ε)
while pop ≠ ∅ do
  for all m ∈ pop do
    Let n be the number of simulations of mutation
    m.
    Simulate m, n more times
        (i.e. now m has been simulated 2n times).
        //this ensures nbTests(m) = O(log(n(m)))
    computeBounds(m, M, δ)
    if lb(m) > 0 then
      Return individual corresponding to mutation
      m.
    else if average fitness(m) < 0 then
      pop = pop \ {m}                    m is discarded.
Return "no good individual in the offspring!"
```

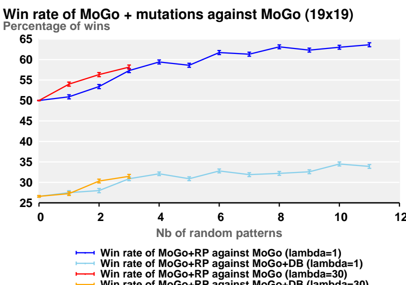
4.5 Experimental results

The theoretical results above for choosing the population size are rather preliminary; the population size can be chosen optimally only if we have many informations. On the other hand, it proposes a criterion different from classical Bernstein races: acceptance is based on the same criterion as usual Bernstein races, but rejection is based on a simple naive empirical average, and with this criterion we have T finite without any prior knowledge. We here experiment rules a bit more complicated than algorithms above.

We have already tried the algorithm on the program MoGo (a software of Go) without real success against the full version of the software (Fig. 4.3).

We have decided to compare with another testbed. The testbed is Monte-Carlo Tree Search (MCTS [Chaslot et al., 2006, Coulom, 2006]) on the game NoGo [Chou et al., 2011]. It is a two-player board game. It is a variant of the game of Go. The rule is the following : the first player which captures one or several stone(s) has lost and the pass move is forbidden. This game has been designed by the Birs seminar on games as a nice challenge for game developers. In all our experiments, we have worked on the size 7x7 of the game.

The baseline is the program NoGo (adapted from MoGo[Lee et al., 2009])



Number of mutations	Cumulated number of evals $\lambda = 1$	Cumulated number of evals $\lambda = 30$
1	219297	63952
2	267483	103261
3	288005	266541

Figure 4.3: Preliminary results of the study of λ on MoGo. No real success against the full version of the software.

without mutation. In our experiments, we have added some rules in order to reject more rapidly some bad mutations, combining Alg. 13 (which requires some knowledge on the distributions) and Alg. 15 (which requires no knowledge). This leads to Alg. 16, which is somehow a combination of these two algorithms, empirically developed for our problem.

Fig. 4.4 shows a slightly better result for $\lambda = 16$, for a fixed number of mutations compared to $\lambda = 1$ and $\lambda = 2$; but more extensive experiments (possibly on toy datasets) are required; results are nearly the same for all λ in our real-world case. In all cases, we could get a nice curve, with a very significant (almost 70%) success rate against the baseline, which is still clearly increasing (yet, in a slower manner).

Another question is about the best population size λ in our experiments. It seems that we have generally a good mutation with frequency 1/15. Using Eq. 4.6, with $\delta = 0.05$ and assuming $f \simeq 1/15$, we get $\lambda \simeq \log(0.05)/\log(1 - 1/15) \simeq 43$. Therefore our analysis suggests a population size $\lambda \simeq 43$.

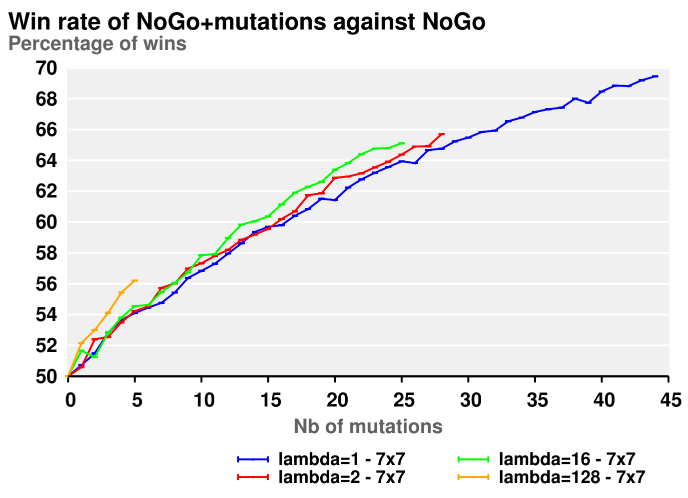
4.6 Conclusion

The use of Bernstein races for rigorously performing non-regression testing was already proposed in [Hoock and Teytaud, 2010]. We here investigate the natural question of the choice of the population size λ, and the modification of Bernstein races when no prior knowledge is available:

- **Choosing the population size.** The good news is that we find a formula for optimally choosing λ, equal to $\log(\delta)/\log(1 - f)$ where f is the frequency of good mutations and δ the risk level chosen by the user. Unfortunately, f is unlikely to be known unless the fitness improvement q that one can expect from good mutations is nearly known, and in this

Algorithm 16 Empirically modified version of Bernstein race for our problem. It is essentially Alg. 15, with a bit more of rejection for fastening the algorithm.

```
BernsteinRace(pop, δ, ε)
while pop ≠ ∅ do
  for all m ∈ pop do
    Let n be the number of simulations of mutation m.
    Simulate m n more times (i.e. now m has been simulated 2n times).
        //this ensures nbTests(m) = O(log(n(m)))
    computeBounds(m, M, δ)
    if lb(m) > 0 then
      Return individual corresponding to mutation m.
    else if average fitness(m) < 0.004 or (average fitness < 0.006 and n > 10^5) then
      pop = pop \ {m}                    m is discarded.
Return "no good individual in the offspring!"
```



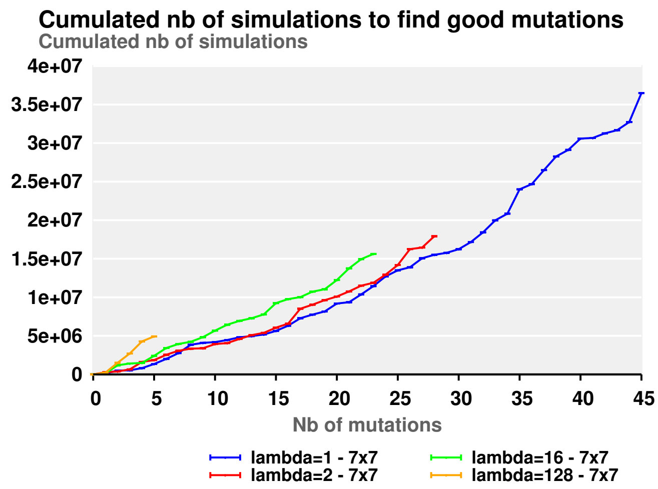


Figure 4.4: **Top:** win rate of NoGo+mutations against the baseline (i.e. NoGo without mutation). Win rates are given with a precision of +/-0.3%. **Bottom:** the “running time“ (measured by the cumulated number of simulations) for finding a good mutation.

case the user is likely to choose ϵ of the order of q, and we show that in this case there is little to win by a good choice of λ: $\lambda = 1$ performs at least nearly as well as all values of λ.

- **What if we have no prior knowledge ?** We could propose a modified Bernstein race (Alg. 15) which has the advantage that it always converges ($T < \infty$), independently of all parameters.

In the experiments, we heuristically combined our various tools for optimizing the performance, proposing Alg. 16. Importantly, we got a very significant result on a new game, NoGo, recently proposed by the Birs seminar on games - the curve shows a regular improvement, for each parametrization of the algorithm. Importantly, we made this work with applications in minds; further investigations, on toy datasets for convenience and clarity, are necessary - so that we can see experimental results with confidence intervals, bridging the gap between our maths and our real-world experiments. We have presented RBGP for optimizing Monte-Carlo Tree Search. But it can be used for building a policy; a mutation is not a pattern but a “sub-policy”. However, using RBGP for solving tasks in the MASH project is not feasible because RBGP needs a lot of simulations for validating rigorously a mutation and simulations in MASH applications are much too expensive.

Chapter 5

Learning prior knowledges

RBGP is already a way for learning prior knowledge by adding it directly in the solver or by building a policy. RBGP solves slowly a problem by optimizing the solver; we wish to learn prior knowledge for solving *quickly* a complex task.

This part is essentially based on the article Solving a Goal-planning task in the MASH project [Hoock and Bibai, 2012].

As it has been seen in Section 1.2.2, there is no information that can help us to infer knowledge for solving more efficiently the problem. But we can simulate several scenarios in order to build a knowledge. First, we define notations (Section 5.1). Then,

1. We learn a categorization of actions (cf Section 5.2); this categorization of action will induce a family of macro-actions.

2. We learn a clustering of the features (i.e., in MASH terminology, a clustering of the state variables - Section 5.3)

3. A tree of subgoals is built, by simulations (Section 5.5), using the macro-actions defined above; this tree of subgoals is related to MCTS trees, and is a form of learnt model of the problem. This tree of subgoals is the policy; the memory of the policy is a pointer to a node of this tree.

4. Voting schemes (Section 5.4) are used for making decisions when the memory points to some nodes of the tree.

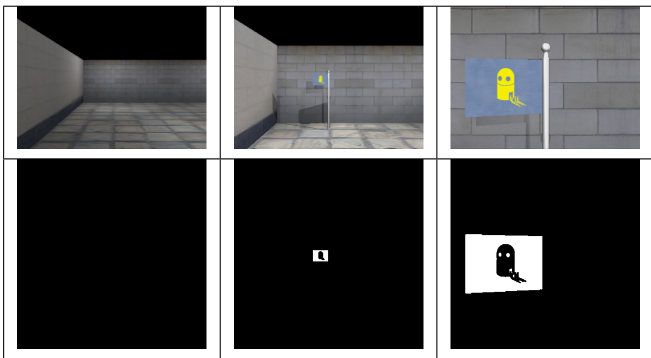


Figure 5.1: **Top:** 3 different views of the avatar. **Bottom:** Images resulting of the application of the *blue* heuristic. **Left:** The blue flag is not seen, no features are activated. **Center:** The blue flag is far, few features are activated. **Right:** The blue flag is closer, more features are activated.

5.1 Notations

An observation is a perception of the avatar on a state (which, in the fully observable case, is a state) and is denoted by o and a state is denoted by s.

From a given state s_t, the state reached after the application of one decision on the state s_t is denoted by s_{t+1}. The function which gives the observation o from a given state s is denoted by $\Omega(s)$.

The set of observations is denoted by O, the set of states by S, the set of features by F, the set of decisions by U, a feature by f and a decision by u.

Let r_t the reward and $finished_t$ the boolean variable which states if the state s_{t+1} is terminal (or final), $MakeTransition$ denotes the transition function (i.e. the application of the decision u_t on the given state s_t) and returns the new state s_{t+1}, the boolean $finished_t$ and the reward r_t: $(s_{t+1}, finished_t, r_t) = MakeTransition(s_t, u_t)$.

When a transition is applied on a state s_t, features of the observation $\Omega(s_t)$ are activated or deactivated (Fig. 5.1). In our implementation, an activated feature is boolean (or binary) and has a value 1; a deactivated feature is a binary feature and has a value 0. Given an observation o_t, $F_a(t)$ denotes the set of active features of this observation.

5.2 Learning a Categorization of the Decisions

At the beginning, the algorithm (or solver) has no idea of the meaning of decisions. But by doing simulations, the impact of a decision on an observation can be studied. In this section, we present in a first part different categories of decision and in a second part a use of this categorization in an experiment on the second MASH application (cf Section 1.2.2).

5.2.1 Categorization

Formally, a decision u is :

- periodic if $\exists k \in \mathbb{N}^*, \forall s_t \in S \ s_{t+k} = s_t$
- stationary if $\forall s_t \in S \ \exists K \in \mathbb{N}^*, \forall j \in \mathbb{N}^* \ s_{t+K} = s_{t+K+j}$
- final[1] if $\forall s_t \in S \ s_{t+1}$ is a final state.

with s_{t+x} the state s_t obtained after applying x times the decision u.

A decision u^- is the inverse of a decision u^+ if $s_t = s_{t+2}$ with $(s_{t+1}, _, _) = MakeTransition(s_t, u^+)$ and $(s_{t+2}, _, _) = MakeTransition(s_{t+1}, u^-)$ ($_$ denotes an ignored information.).

Notice that, (i) when the environment of a given application is partially observable [2], we work with a memory (Section 5.4.1) and on the observation $\Omega(s)$ instead of the state s and (ii) when we are in a stochastic application, we use in these equations approximations instead of equalities.

First, from a given state s_t, we categorize a decision u by applying it several times. Let k the number of times the decision u has been applied.

1. A first case is to reach a final state by applying the first time ($k = 1$) the decision u. The decision is categorized as a final decision.
2. A second case is the decision has no more impact on the observation ($\Omega(s_{t+k-1}) \approx \Omega(s_{t+k})$). The decision u is categorized as a stationary decision.
3. A third case is the initial observation $\Omega(s_t)$ is seen a new time ($\Omega(s_{t+k}) \approx \Omega(s_t)$). The decision u is categorized as a periodic decision.

[1] In MASH, there is no final decision, but draw of letters (cf Section 5.6.1) contains one
[2] MASH is partially observable

4. A fourth case is the initial observation $\Omega(s_t)$ is seen a new time after applying the decision u one time ($k = 1$) and another decision $u^* \neq u$ one time, too. Decisions u and u^* are categorized as inverse decisions.

Pieces of information will be very useful for simulating more efficiently.

5.2.2 Experiment on 10 flags problems

We propose a first use of the categorization of decisions. We consider the "10 flags" problem (Section 1.2.2).

The policy based on decision stumps is given in Alg. 17.

A Direct Policy Search with an evolutionary strategy is used for learning parameters. The best policy is evaluated anew and compared with a new one at each time. Each policy is evaluated on 10 tests.

The policy contains 3 parameters:

- $decision_forward$ a decision
- $decision_turn$ a decision
- $prob$ real between 0 and 1.

We compare 2 methods for learning these parameters. The first method is a random search. In the second method, parameters values are also chosen randomly, but the choice can be biased by the categorization of decisions.

More precisely, the second method is

- With probability 1/10, use the first method.
- Otherwise,
  - For the parameter $decision_forward$, draw randomly a decision which is stationary (if not, draws randomly a decision).
  - For the parameter $decision_turn$, draw randomly a decision which is periodic (if not, draws randomly a decision).
  - For the parameter $prob$, draw randomly a value between 0 and 1.

Results are shown in Fig. 5.2. The categorization of decisions allows to learn faster the best policy. Note that this version of Direct Policy Search does not work on the "blue flag then red flag" application. The reason is that the policy in Alg. 17 does not use a memory, which is required for solving the “blue flag then red flag“ problem.

Algorithm 17 the policy on decision stumps. The 3 parameters are $decision_turn$, $decision_forward$ and $prob$. From the decision u, the function $GetMaximalNbRepeat$ returns the number of times k that the decision u has been repeated in order to categorize it (Section 5.2.1); this is the maximal number of times that the decision u can be repeated.

$\pi(o)$
Let u a static variable containing the decision
Let l a static variable containing the remaining number of times that the decision u will be repeated
Let f_j the jth feature
Let v_j the value of the jth feature of the observation o.

for each components j of the observation o **do**
  if feature j is boolean **then**
    if $v_j > 0$ **then**
      $u =$ \$$decision_forward$\$
      $l = 1$
if $l > 0$ **then**
  $l--$
  Return u
Let p a value initialized randomly between 0 and 1.
if $p >$ \$$prob$\$ **then**
  $u =$ \$$decision_turn$\$
else
  u is randomly chosen
$l = 1 + rand() \mod GetMaximalNbRepeat(u)$
$l--$
Return u

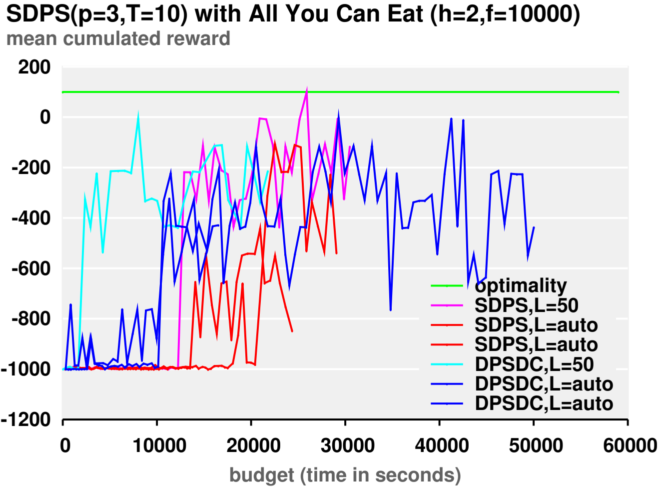


Figure 5.2: Direct Policy Search on Decision Stumps applied to All You Can Eat problem (i.e. 10 flags application). In order to compare the best policy and a new one, both policies are tested 10 times (T=10). During the learning phase, 3 parameters ($p = 3$) are learnt : $decision_forward$, $decision_turn$ and $prob$. In All You Can Eat Problem (= 10 flags problem), the 2 used heuristics ($h = 2$) are *red* and *identity* with a random sampling of 10,000 features. Red curves are obtained without the categorization of decisions and blue curves with the categorization of decisions. SDPS means Stump Direct Policy Search and DPSDC means Direct Policy Search with Categorization of Decisions. The parameter L given by the function $GetMaximalNbRepeat$ (in Alg. 17) is the maximal length of a macro-decision, i.e. a decision repeated several times (See Section 5.4); *auto* means that L is computed automatically following the decision u; by default, $L = 50$. **Conclusion: The best policy is found faster with a learning of the categorization of decisions** (blue curves).

5.3 Learning a Clustering of the Features

A feature in MASH is a binary observed state variable. Features are grouped; a group of features is termed a heuristic. Often, these features have related behavior, but the learning must work without any information on these features. The idea is to clusterize relevant features that can help the algorithm to find goal. To solve this issue, we propose an algorithm of clustering (Alg. 18) called CluVo for finding correlated features. CluVo makes no assumption about the number of clusters and is completely unsupervised (Features are not labelled.). In the MASH problem, it should, ideally, construct 2 clusters : "Features which come from the *blue* heuristic" and "Features which come from the *red* heuristic".

Algorithm 18 the complete Clustering (generation of the collection + clustering). The name CluVo comes from the contraction of words **Clu**stering and **Vo**te. The function $GenerateNewDatabase$ is described in Alg.20 and the function $Clusterize$ is described in Alg. 21.

Function $CluVo()$
Let $clusters$ a static variable containing all clusters of features.
Initialization: each cluster is composed of one feature.

for $iteration$ in $1..+\infty$ **do**
  $ldb = GenerateNewDatabase(iteration)$
  $nbClus = size(clusters)$
  $clusters = Clusterize(ldb, clusters)$
  if $nbClus \neq size(clusters)$ **then**
    $nbClus = size(clusters)$
  else
    break

In the clustering, we distinguish 3 parts : (i) The generation of collection (or database) of lists of features by doing simulations - (ii) Clustering features - (iii) The metric which is the correlation between 2 features.

5.3.1 Simulations for the generation of the collection

For the generation of the collection, following the number of iterations, simulations are done differently:

- for the first 3 iterations, decisions are chosen randomly;
- but from the 4th iteration, decisions are chosen following a probability:
  - either randomly
  - or by vote (see Section 5.4.4).

Some features are very difficult to be activated with random simulations. That's why the vote is used for guiding the simulation. Indeed, given a cluster, the vote is very efficient for activating these features but needs several updates of Eq. 5.3 before to be used efficiently. That's why the vote is not used at the first 3 iterations. Another good point of the vote is that when efficient, it activates together features which are truly correlated; from the 4th iteration, there is less risk to clusterize 2 falsely correlated features.

The generation of the collection is shown in Alg. 19 (for the first generation) and Alg. 20. A list of features is added in the collection at the end of the application of a macro-decision (see Section 5.4.2). The function $GetMacroDecisionLength$ returns the length of the macro-decision and is defined in the caption of Alg. 22. The function $GetDecisionByVote$ returns the decision chosen by a vote.

Algorithm 19 Generation of collection of lists of features for the first iteration beginning to 1. A collection of lists of features is a list of lists of features.

```
Function GenerateNewDatabase_FirstIteration()
Let PARAM_GNDFI1 a static parameter fixed to 20
Let ldb the list of lists of features
nbLoops = PARAM_GNDFI1 * #U
for i in 1..nbLoops do
    Let s a state initialized randomly
    Let u a random decision
    Let GetActiveFeatures(o) the function which returns the list of
    active features of an observation o
    l_active = GetActiveFeatures(Ω(s))
    l = GetMacroDecisionLength(Ω(s), u, NULL); finished = false
    for j in 1..l do
        s, finished, _ = MakeTransition(s, u)
        l_active = Append(GetActiveFeatures(Ω(s)), l_active)
        if finished then
            break
    if Size(l_active) < 2 then
        Decrement i
    else
        Push(l_active, ldb)
Return ldb
```

Algorithm 20 Generation of collection of lists of features (i.e. list of lists of features). *nbLoops* simulations in which macro-decisions (Section 5.4.2) are used are performed. During the application of a macro-decision, all activated features are kept in a list l_{active}. This list l_{active} is added in the collection at the end of the application of the macro-decision.

```
Function GenerateNewDatabase(iteration)
Let P_GND1 a static parameter fixed to 20
Let P_GND2 a static parameter fixed to 4
Let P_GND3 a static parameter fixed to 4
Let P_GND4 a static parameter fixed to 100
Let P_GND5 a static parameter fixed to 4
if iteration == 1 then
    Return GenerateNewDatabase_FirstIteration()
Let clusters a static variable containing all clusters of features.
Let ldb the list of lists of features
nbLoops = P_GND1 * #U
for i in 1..nbLoops do
    Let s a state initialized randomly
    Explore(s) [a]
    Let clust initialized to an empty cluster
    if iteration >= P_GND2 ∧ max_{cl∈clusters}(Size(cl)) > 1 then
        clust = ChoiceOneCluster(Ω(s), clusters)
    maxK = P_GND3
    if clust ≠ ∅ then
        maxK = P_GND4 [b]
    for k in 1..maxK do
        Let u initialized to a random decision
        Let l_active initialized to an empty list of features
        if clust ≠ ∅ ∧ k mod P_GND5 ≠ 0 then
            u = GetDecisionByVote(Ω(s), clust) [c]
        l = GetMacroDecisionLength(Ω(s), u, NULL)
        finished = false
        for j in 1..l do
            (s, finished, _) = MakeTransition(s, u)
            Increment numDec
            Append(GetActiveFeatures(Ω(s)), l_active)
            if finished then
                break
        Push(l_active, ldb) [d]
        if finished then
            break
Return ldb
```

[a] apply exploration decisions on the state s until a feature of the observation $\Omega(s)$ is activated (See Section 5.4.3).

[b] The number of macro-decisions applied during a simulation is denoted $maxK$. By default, $maxK$ is small (i.e. P_GNCD3); the risk of bad correlation between features is high due to random decisions. When a cluster of features can guide the simulation by the mechanism of vote, $maxK$ is high (i.e. P_GNCD4).

[c] returns a decision given from active features of the observation $\Omega(s)$ and belonging to *clust*. (See Section 5.4.4).If no active features, a random decision is taken.

[d] l_{active} is not added if $l_{active} == \emptyset$.

5.3.2 Clustering features

The approach of our algorithm is hierarchical. There exist 2 main families of clustering algorithms : (i) The divisive (or top-down) clustering [Savaresi et al., 2002] starts from one cluster and gradually splits clusters. (ii) The agglomerative (or bottom-up) clustering [Defays, 1977] starts from a set of small clusters and then aggregates them. Because of the system of vote (Eq. 5.3), a bottom-up clustering is surely better. The shown algorithm is therefore agglomerative. Therefore, each feature corresponds to a cluster at the beginning of the CluVo algorithm.

Algorithm 21 The Clustering phase. $SelectTwoFeatures(lclus, ldb)$ selects 2 features by using either the list of clusters $lclus$ or the collection of the list of features ldb. $Merge(clus1, clus2, lclus)$ creates a new cluster into the set $lclus$ by merging $clus1$ and $clus2$ and then removes them from the set $lclus$.

Function $Clusterize(ldb, lclus)$
Let P_C1 a static parameter fixed to 10^4
Let P_C2 a static parameter fixed to 10^7
Let $nbClus$ the number of clusters initialized to the size of ldb
$nbTry = \min(\max(P_C1, nbClus \times nbClus), P_C2)$
while $nbTry > 0 \wedge nbClus > 1$ **do**
  Decrement $nbTry$
  $(f1, f2) = SelectTwoFeatures(lclus, ldb)$
  Let $clus1$ the cluster of features to which the feature $f1$ belongs
  Let $clus2$ the cluster of features to which the feature $f2$ belongs
  if $clus1 \neq clus2 \wedge f1$ and $f2$ are correlated[a] **then**
    $Merge(clus1, clus2, lclus)$
    $nbClus = size(lclus)$
    $nbTry = \min(\max(P_C1, nbClus \times nbClus), P_C2)$
Return $lclus$

[a] See Section 5.3.3

5.3.3 Metric

This section describes the similarity measure used in Alg. 21. Let ldb the collection of the list of features as obtained by Alg. 20. Let $f1$ and $f2$ two features of the set of features obtained thanks to a uniform draw.

Let $Db(f) = \{list \in ldb; f \in list\}$ with f a feature and $Db(f1, f2) = Db(f1) \cap Db(f2)$

Let $Card(X) = \#X$ the number of elements of the set X.

The features $f1$ and $f2$ are significantly correlated if $\#Db(f1, f2)) > 5$ and $3 \times \#Db(f1, f2) >= max(\#Db(f1), \#Db(f2))$.

The numbers 5 and 3 have been fixed after experiments.

5.4 The policy

The policy π makes a decision $\pi(o, M)$ depending on o (its current observation) and M (its memory which is for us a pointer to a node in the tree of subgoals). This decision is, as far as possible, a good decision, and we will note $\pi(o)$ instead of $\pi(o, M)$ for short (yet, it depends on the memory, which is a static variable of our policy).

Sometimes, the solver should be able to realize several small tasks as subgoals for realizing the complete task[3]. A small task is called a subgoal. A lot of works on subgoals have been already produced in the literature such as [Stolle and Precup, 2002], [Schmidhuber and Wahnsiedler, 1992], [Wolfe and Barto, 2005], [Menache et al., 2002] or [Bibai et al., 2010]. In order to realize this task, the policy must be able to switch from a subgoal to another subgoal.

The first subsection (Section 5.4.1) describes the memory and subgoals given by gn in Alg. 22, the second subsection introduces macro-decisions (Section 5.4.2) and the two last subsections describe the mechanism of choosing decisions when about a given subgoal, there is:

- no observed information: the exploration (Section 5.4.3)
- observed information: the vote (Section 5.4.4).

5.4.1 Memory for switching subgoals

The policy is a tree of subgoals. A node of the tree is a subgoal; a subgoal is:

[3] e.g. in the main MASH application, the avatar must touch the blue flag (first small task) and then touch the red flag (second small task); which implies to keep the information that the blue flag has been touched.

Algorithm 22 A simplified version of our policy. The primitive $GetDecision$ is a function which returns a decision u. The primitive $GetMacroDecisionLength$ is a function which returns the number of times that the decision u is repeated (cf Macro-Decisions in Section 5.4.2). This number is fixed according to the categorization of the decision u (cf Section 5.2.1) and the active features in the observation o. Optionally, the set of active features is restricted to the list of features given by the node gn (cf subgoal in Section 5.4.1). The function $GetDecision(o, gn)$ returns either a final decision when the subgoal gn (Section 5.4.1) is a final decision, or a decision by vote if some features given by the node gn are activated in the observation o, or else a decision for exploration.

Function $\pi(o)$
Let ga a static variable containing a tree of subgoals
Let gn the current node and $root$ the root node of ga
Let u a static variable containing the decision
Let l a static variable containing the remaining number of times that the decision u will be repeated
if gn is $root \vee IsReached(gn, o)$ **then**
  $gn = ChooseNextNode(gn)$
  $l = 0$
if $l == 0$ **then**
  Build a new Macro-Decision (u, l)

- $u = GetDecision(o, gn)$
- $l = GetMacroDecisionLength(o, u, gn)$

Decrement l
Return u

- either a list of features and a minimum number of features to be activated for the subgoal to be fulfilled
- or a final decision if it exists.

The memory is a pointer to one node in this tree. At the beginning of a run of the policy, the memory points to the root node. From a given observation o, when the memory points to subgoal gn, then the algorithm applies action u chosen by the function $GetDecision(o, gn)$ in Alg. 22. The memory pointer switches from subgoal gn to subgoal gn' when at least $nb(o)$ features of node gn are activated (function $IsReached$); gn' is chosen by $ChooseNextNode()$.

5.4.2 Useful Macro-Decisions

A Macro-Decision [Mcgovern et al., 1997] (or Macro-Action) is a decision repeated several times. A Macro-Decision is modeled by the couple (u, l).

- u is the decision
- l is the number of times that the decision is repeated.

A decision can be useless because the application of the decision on the state s_t has no more impact (stationary decision), or the decision is the inverse of the last decision, or when the avatar has made a complete turn, it does not need to continue to turn (periodic decision). The set of useful decisions for a given state s_t is denoted U_{ok}.

5.4.3 Exploration for finding a subgoal

In the MASH application, the environment is partially observable. The environment must be explored. But because of the size of the environment, classical Random Search algorithm can not be applied. Macro-decisions is a tool that can help to explore more rapidly the environment[Mcgovern et al., 1997]. During the exploration, when there is no activated features in the observation $\Omega(s_t)$[4] and then a given decision activates features in the observation $\Omega(s_{t+1})$, this decision can be considered as a good one[5]. These decisions can be found statistically after a lot of explorations.

[4] In the MASH application, the flag is not seen.

[5] To find a flag, some decisions are more useful than others (e.g. the decision to turn is good).

In order to choose good decisions for finding a subgoal, we define the variable called $disc(Fclus, u)$. Let $Fclus$ a set of features

$$disc(Fclus, u) = \#\{u_t, u_t == u \wedge F_a(t) == \emptyset \wedge F_a(t+1) \cap Fclus \neq \emptyset\} \quad (5.1)$$

The formulas can be updated at each simulation after a $MakeTransition$. Then for a given node gn, the decision for exploration is given by

- Let obj the set of features given by the node gn.
- with probability proportional to
  - $disc(obj, u)$, if $u \in U_{ok}$ then return the decision u
  - 1, return a random decision.
- else return a random decision

When features have been activated, decisions can be now chosen by using these activated features.

5.4.4 Vote for reaching a subgoal

Given a cluster of features[6], a set of decisions and from an initial state s_0 whose observation $\Omega(s_0)$ contains some activated features, a sequence of decisions which leads to the activation of a maximum of features of the cluster is searched. This mechanism describes the main component (i.e. the vote) of the function *GetDecision* (cf caption of Alg. 22); it is present in the generation of the database ($GetDecisionByVote$ in Alg. 20) during the clustering, in the policy (Alg. 22) and in GMCTS (Alg. 23).

A decision can activate new features or deactivate features. Here, the goal is to activate a maximal number of features of the cluster. In order to do this, the idea is that each active feature votes for a decision which increases the number of active features of the cluster.

For each couple $(f, u) \in F \times U$ is associated a score $score(f, u)$. $score(f, u)$ is a cumulative number of activated features (cf Eq. 5.3). A vote is defined by

$$Vote(f) = Argmax_{u \in U} score(f, u). \quad (5.2)$$

Let $F_v(u) = \{f \in F_a(t) \cap FClus, u == Vote(f)\}$ the set of active features of the observation $\Omega(s_t)$ belonging to a cluster (or set of correlated features) $FClus$ and which vote for the decision u.

Then, the decision by vote is given by $Argmax_{u \in U_{ok}} \#(F_v(u))$.

[6] a cluster can be the list of features given by a subgoal.

We can interpret as: For each decision u, we count the number of active features which vote for the decision u. The decision which received most votes is selected. The vote is inspired by the meta-classifier Adaboost [Freund and Schapire, 1995] which ranks a given element in a class (among different classes) using weak classifiers. Features can be seen as weak classifiers and decisions as classes.

The function called *score* gives a cumulative number of features of one cluster which have been activated from a sample of states. More precisely, let (s_t, u, s_{t+1}) the state s, the decision u and the state s_{t+1} obtained immediately after applying the decision u on the state s.

Let $F_{a|Fclus}(t)$ the set of active features of the observation $\Omega(s_t)$ belonging to the cluster $Fclus$ ($F_{a|Fclus} = F_a(t) \cap Fclus$). Let $f_a \in Fclus$ an active feature of the observation $\Omega(s)$. Then

$$score(f_a, u) = \sum_{s_t} (\#F_{a|Fclus}(t+1) - \#F_{a|Fclus}(t)). \tag{5.3}$$

The signification of the formula is to activate for a given cluster more features in the observation $\Omega(s_{t+1})$ than activated features of the observation $\Omega(s_t)$.

Note that $f_a \in F_a(t)$ but *maybe*, $f_a \notin F_a(t+1)$. In this case, the feature f_a has been deactivated when the decision u has been applied on the state s_t.

The function called *score* can be updated at any moment when a decision is applied.

5.5 Learning a Sequence of Subgoals

The categorization of decisions and the Clustering of features are unsupervised methods, the information of the reward has not been used. Now, for building a policy, represented by a tree of successive subgoals as described in Section 5.4.1, we use the reward. This section describes this process.

The proposed algorithm (Alg. 23) is a Monte-Carlo Tree Search [Coulom, 2006] on subgoals. The algorithm is called GMCTS (Goal Monte-Carlo Tree Search). GMCTS builds incrementally the tree by doing Monte-Carlo simulations. The learnt sequence of subgoals is the most simulated sequence of subgoals in the tree.

We describe in a first subsection the Monte-Carlo simulation and in a second subsection the policy of GMCTS.

5.5.1 Monte-Carlo simulation

For each Monte-Carlo simulation, we

1. start to the root of the tree,
2. repeat
   (a) choose a subgoal
      - either we choose the most promising
      - or we add and choose a new one

   (b) achieve the subgoal by repeating
      i. we apply the voting scheme
      ii. we keep rewards
      iii. until we have accomplished the subgoal

   (c) until a final state is reached

At the end of the Monte-Carlo simulation, the tree is updated with kept rewards.

5.5.2 Policy of GMCTS

The policy of the GMCTS is given by Alg. 24

Given the node gn, the function $ChooseNewGoal$ creates a new child node of gn and returns it (exploration), whereas the function $ChooseBestGoal$ returns the most promising child node of gn (exploitation).

Let *nb_children* the number of children of gn.

- With a probability $1/nb_children$, we explore with adding a new subgoal.
- With a probability $1 - (1/nb_children)$, we exploit the most promising subgoal.

We find again a common statement of the compromise exploration/exploitation: the more we have simulated, the more we exploit. Indeed, the more we have simulated a node, the more we have added children, then the smaller the probability to add a new subgoal is, thus the more we exploit.

Following a mode, the most promising is the child node which maximizes

- either the average of cumulative rewards (mode 1)

Algorithm 23 Monte-Carlo Tree Search for searching goal. The argument $maxSimulation$ is a parameter of the GMCTS algorithm and denotes the number of simulations which will be done to build the tree ga. Macro-decisions are used in the GMCTS algorithm.

Function $Gmcts(maxSimulation)$
Let ga a static variable containing a tree of subgoals
Let $root$ the root of the tree ga.
for sim in $[\![1; maxSimulation]\!]$ **do**
  Let s initialized to the initial state.
  $(r_{cum}, finished, gn) = (0, false, root)$
  while $finished \neq true$ **do**
    $gn = ChooseNextGoal(gn)$
    $reached = false$
    while $reached \neq true \wedge finished \neq true$ **do**
      $u = GetDecision(\Omega(s), gn)$
      $l = GetMacroDecisionLength(\Omega(s), u, gn)$
      for j in $1..l$ **do**
        $(s, finished, r) = MakeTransition(s, u)$
        $r_{cum} = r + r_{cum}$
        $reached = IsReached(gn, \Omega(s))$
        if $reached \vee finished$ **then**
          break
      {this loop is the application of the Macro-decision (u, l) on the state s}
  $UpdateTree(r_{cum}, gn)$ {Pieces of information in a node which are to update are : (i) the number of simulations (ii) the average of the cumulative rewards and (iii) the cumulative reward. All nodes visited during the simulation (going from the node gn to $root$) are updated.}

Algorithm 24 The policy of GMCTS.

```
Function ChooseNextGoal(gn)
Let P_CG1 a static parameter fixed to 3
if IsLeaf(gn) then
   Return ChooseNewGoal(gn)
mode = 1
if rand() mod P_CG1 ≠ 0 then
   mode = 2
if rand() mod Size(Children(gn)) ≠ 0 then
   Return ChooseBestGoal(gn, mode)
else
   Return ChooseNewGoal(gn)
```

- or the cumulative reward of its last simulation (mode 2)

The mode 1 is chosen with a probability 1/3 whereas the mode 2 is chosen with a probability 2/3.

5.6 Experiments

Experiments with CluVo+GMCTS algorithm are presented. For the genericity, the algorithm has been applied on another application: optimization of a draw of letters. First, the letter's draw testbed is described. Then, results on this testbed and on the main application of MASH are given. Finally, results are discussed.

5.6.1 The optimization of a draw of letters

The longest word is a game where a draw of 10 letters is made and the goal is to produce the longest correct word according to a dictionary of around 300,000 French words[7]. But, in order to have interesting draws, letters are not drawn equiprobably. The goal is to find the best distribution of letters in order to have draws with long correct words (Fig. 5.4).

[7] http://www.pallier.org/ressources/dicofr/dicofr.html.

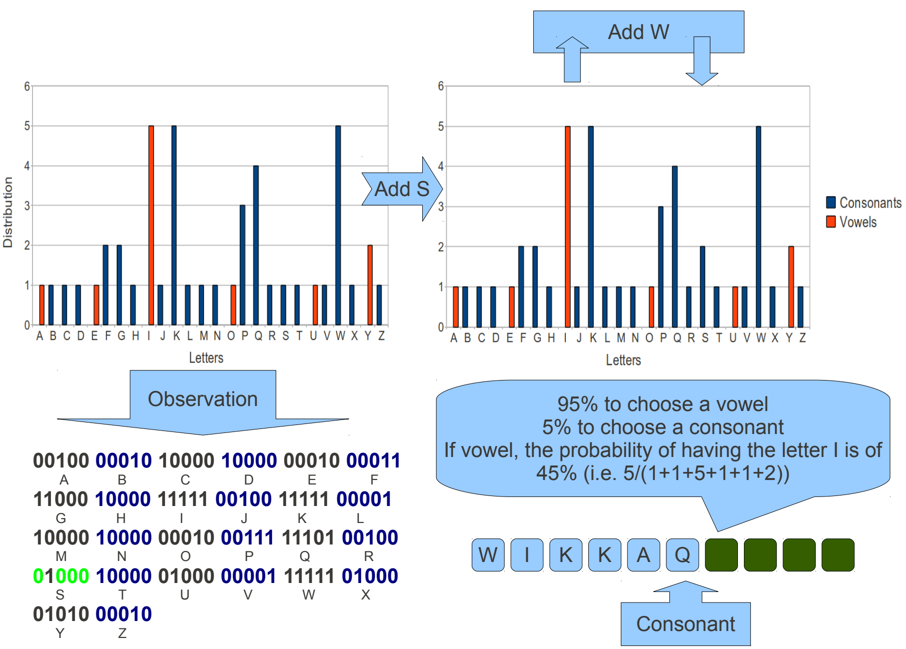


Figure 5.3: Letters distribution in LD($NbL = 5, _, _$). **Top left :** a distribution of letters. **Bottom left :** an example of the observation given to the agent which describes this distribution. **Top right :** the new distribution after adding the letter S. Adding the letter W does not change anymore the distribution because the maximal number NbL of W in the distribution is reached (stationary decision). The new observation given to the agent will be the same as **bottom left** with one green zero changed into 1. **Bottom right :** a draw in progress. This draw represents distributions at **Top**; W, K, I and Q are letters which have the greater probability to be drawn.

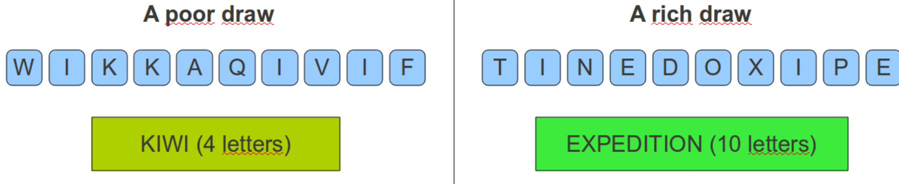


Figure 5.4: The longest word. **Left:** an example of poor draw. The longest word is only composed of 4 letters. **Right:** an example of rich draw. The longest word is composed of 10 letters. By considering these 2 draws in LD($_, _, NbDraws = 2$), the reward given to the agent is of 7 (i.e. (4+10)/2). With one of the distributions given in Fig. 5.3, the probability to have the poor draw is greater than the probability to have the rich draw.

State

The state is a vector of size $NbL \times 26$; each of the 26 groups contains NbL features. Each feature is 0 or 1. The state describes a distribution on words as follows: we randomly draw 10 letters, and we start by randomly draw a consonant - each consonant has probability proportional to the number of 1 in its features. Then, for each letter, we switch from vowel to consonant or from consonant to vowel with probability 95%; and letters are drawn among consonants or vowels with probability proportional to the number of features at 1 (Fig. 5.3). The initial distribution contains each letter one times.

Decisions

27 decisions are possible. Each decision (except the decision "not add") consists in adding one letter; the feature which is activated among the NbL corresponding features is randomly drawn.

Reward

The reward is 0 if s is not terminal. When s is terminal, the reward is computed as the average length of longest words through $NbDraws$ draws.

Terminal State

Let Mal the maximal number of letters we can add in all. A state is terminal if either Mal letters have been added or the decision "not add" has been taken.

For different variants of this problem, the letter's draw application will be denoted $LD(NbL, Mal, NbDraws)$. A difficulty of this testbed is that the reward is stochastic. Like in the MASH application, we work in a setting with no prior knowledge.

5.6.2 Results

For both applications, all tests have been made with 100 evaluations and all experiments have been exactly made with the same algorithm. No parameter has been modified.

Results on the draw of letters

Results of clustering is given in Tab. 5.1.

CluVo+GMCTS on Letters'Draw application - LD(20,200,10)
mean cumulated reward
9
8
7
6
5
4
3
0 5000 10000 15000 20000 25000 30000 35000
iteration of GMCTS
Run 1
Run 2
Min of all runs
Avg of all runs
Max of all runs

Figure 5.5: Results of GMCTS on LD(20,100,10). Run 1 and Run 2 correspond to 2 runs. The mean cumulative reward corresponds to the average length of longest words on draws of 10 letters.

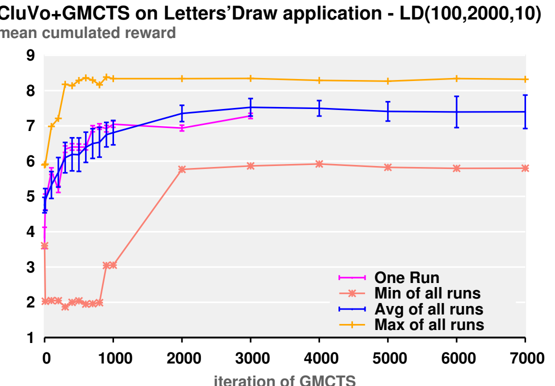


Figure 5.6: Results of GMCTS on LD(100,1000,10). The errorline of the average curve is the standard deviation computed on all remaining runs. The errorline of the curve "One Run" is the standard deviation computed through 100 evaluations.

Application	LD(20,200,10)	LD(100,2000,10)
Nb Runs	43	35
Iterations	between 3 and 4	3 (except one run with 4)
Error	2	0
Incomplete	1	0
Success	40/43	35/35

Table 5.1: Results of clustering on Letter's Draw application. The row Iterations is the number of iterations of Clustering. The row Error is the number of runs with bad correlation(s). The row Incomplete is the number of runs where the correct number of clusters has not been found. The row Success is the number of runs with complete clustering without bad correlation.

Results of GMCTS are shown in Fig. 5.5 and Fig. 5.6 in which the number of iterations is the number of simulations of the GMCTS algorithm. The mean cumulative reward corresponds to the average length of longest words.

Results on the MASH application: "blue flag then red flag"

Note that CluVo+GMCTS does not work on the "10 flags" application because the clustering always fails; results are on the "blue flag then red flag" application.

Results of the Clustering are : Over 12 runs, the number of runs with a bad correlation of features has been of 9. For the 3 other runs, 2 runs needed 5 iterations, one needed 8 iterations. At the end of the Clustering, 2 clusters have been well found.

Results of GMCTS are presented in the Tab. 5.2.

Discussions

Letter's draw optimization

The clustering works well. Although in $LD(100, 2000, 10)$ application, the size of the state space is 5 times larger and the maximal length of a simulation is 10 times bigger than in $LD(20, 100, 10)$, the clustering works always well and the GMCTS algorithm does not need more simulations to converge in average. In a similar application, the french TV game "Des Chiffres et Des Lettres", the longest word has in average a length of 8.12 letters (Average calculated over 275 draws). In Fig. 5.5 and Fig. 5.6, best runs have very good results; their score (given by the mean cumulative reward) reaches or exceeds a little 8.12. The score in average (around 7.5) is good, too.

Run	Iteration	AvgCR	Success
1	1	-13.65	
	10	4.82	
	100	9.67	
2	1	8.87	
	10	10.62	
	100	10.2	
3	1	12.09 ± 1.06	81%
	10	12.64 ± 1.03	86%
	100	11.38 ± 1.21	77%

Table 5.2: Results of GMCTS on MASH application. The column Iteration is the number of simulation of GMCTS. AvgCR means the average of cumulative reward. The optimal cumulative reward is 15. The last column Success is the percentage of evaluations where the goal "touch the blue flag and then the red flag" has been accomplished (through 100 evaluations).

For information, I give best sequences of subgoal found in the experiments LD(20,200,10) and LD(100,2000,10). A subgoal (cf Section 5.4.1 for the definition) is labelled by

- either the letter (which the subgoal describes) optionally followed between parenthesis by the proportion of features to activate (by default, the proportion is 1)
- or "finish" when we want to take a final decision.

In LD(20,200,10), the best sequence of subgoals among all runs is :

$$O(0.1) - R - A - T - E - S(0.8) - N - I(0.9) - finish$$

In LD(100,2000,10), the best sequence of subgoals among all runs is :

$$T - I(0.9) - E(0.8) - L(0.6) - R - N(0.7) - S - A - finish$$

In french, the 10 most common letters[8] are

E	S	A	I	T	N	R	U	L	O

Therefore, GMCTS has added the most common letters in French for improving the draw of letters.

MASH Application: "blue flag then red flag"

The results are slow but stable. With 12 restarts on the clustering stage (2 hours each), we get 3 successes (which were detected as successes by heuristics

[8]http://en.wikipedia.org/wiki/Letter_frequency

without further testing); then, GMCTS could be successful on each of these 3 runs (8 hours each run). So on a parallel machine all this could be done in 10 hours, or 48 hours on a sequential machine. The success rate of CluVo is 25%[9] (tested on 12 runs, and success can be detected on the fly) and the success rate of GMCTS is 100% (i.e. the goal "touch the blue flag then touch the red flag" is successfully found).

For information, like in the precedent discussion on results about the draw of letters, I give some sequences of subgoals found by GMCTS in the MASH application. The name of the subgoal (i.e. *blue* or *red*) is labelled by the color which the subgoal describes. On the second run given in Tab. 5.2, the best sequence of subgoals was:

- after 1 simulation
$$blue(0.8) - red(0.3)$$
- after 10 simulations
$$blue - red$$
- after 100 simulations
$$blue(0.8) - blue - blue(0.2) - red - red(0.5)$$

After 10 simulations, GMCTS had found the optimal sequence of subgoals (i.e. $blue - red$). Why GMCTS has not converged to the optimal sequence? In fact, touching the blue flag several times is not penalized whereas touching the red flag without touched the blue flag is penalized by -5. Thus, it is better to be sure to have touched the blue flag (even trying again and again) before going to the red flag.

5.7 Conclusion

In this chapter, we have proposed a generic algorithm CluVo+GMCTS which solves a very complex[10] task in an unknown environment of large size. In this environment, the semantics of decisions are unknown, there is no prior knowledge and pure random search gives very rarely an informative reward. Our approach combines macro-actions, categorization of actions, feature selection, subgoal learning, clustering and Monte-Carlo Tree Search. The Clustering method tries to learn some knowledge and from those results, GMCTS

[9] On n machines (one CluVo per machine), the success rate of CluVo would be around $1 - (3/4)^n$; this good parallel behavior is because we can detect successful runs of CluVo by considering empirical performance of GMCTS after the n instances of CluVo.

[10] the complexity is due to the agnostic context and without prior knowledge.

tries to find a strategy for reaching the goal. The genericity of the algorithm has been shown on 2 applications. 2 applications is not a big number, but they are very different, and no knowledge about the problem has been included. Clearly, 2 applications is not enough for showing the genericity of an approach, but further tests are in progress.

In further work, we will compare GMCTS with other algorithms such as Nested Monte-Carlo Search adapted to goal space. This work is a step in the direction of truly generic algorithms for very difficult settings, with no prior knowledge on actions, rewards, transitions. In the future we might extend the genericity by (i) testing on additional problems (ii) if necessary reworking the algorithm.

Part III

Conclusion

For solving complex tasks under 4 difficult constraints

1. the number of states is huge
2. the reward carries little information
3. small probability to reach quickly a good final state
4. no knowledge or inexploitable knowledge

several answers have been brought:

- simulating using the exploration/exploitation compromise
- reducing the complexity of the problem by local searches
- building policies using genetic programming
- learning prior knowledge

and so, we have proposed respectively 4 solvers:

- 2 model-based algorithms:
  - Monte-Carlo Tree Search
  - GoldenEye (a mix between MCTS and A*)
- 2 model-free algorithms:
  - RBGP (using bandits/races for genetic programming)
  - CluVo+GMCTS (using clustering of features, macro-actions, sub-goals and Monte-Carlo Tree Search)

We will make a point about these algorithms and then we will discuss about several open questions.

Monte-Carlo Tree Search: The Monte-Carlo Tree Search has proved its efficiency in the Game of Go. MoGo, our software of Go, won the first ever 9x9 game against a top pro[11] as black. In 2012, Zen, the best Go program, has beaten Takemiya Masaki a 9p professional player with only 4 handicap stones. The first victory of a machine against a professional Go player in an even game on the big 19×19 board is getting closer.

GoldenEye: Our GoldenEye solver handles the problem of searching local tasks in a global environment; the solving starts locally but gradually

[11] the top pro was Zhou Junxun (9p)

expands over the whole board if necessary. GoldenEye algorithm (in particular the local search) is a first step for a generic integration of exact solver in MCTS.

MCTS and GoldenEye don't work on MASH applications because they need a model.

RBGP: We have proposed a Direct Policy Search called RBGP which optimizes a Monte-Carlo Tree Search by automatically adding knowledge. RBGP validates rigorously the added knowledge by taking into account Multiple Simultaneous Hypothesis Testing. RBGP has given good results for improving MoGo.

RBGP does not work on Mash. RBGP needs many simulations for validating rigorously a mutation; simulations in MASH are much too expensive.

CluVo+GMCTS: The CluVo+GMCTS algorithm is the only algorithm which has worked on the main MASH application. It combines several methods such as Categorization of decisions, Macro-Decisions, Clustering, Monte-Carlo Tree Search[12] on subgoals and Vote.

The categorization of decisions enabled to do more intelligent simulations by avoiding useless or backward decisions and **macro-decisions** have been a tool for exploring efficiently the environment.

The **clustering** (CluVo) on features has been useful for solving a complex task with no assumption about the architecture of the state. From the results of clustering, **subgoals** have been defined. The main MASH task is simple (i.e. blue flag then red flag) but it needs a memory (due to partial observation) and it might be more complicated such as blue flag then red flag then yellow flag, the red flag again to do $x > 0$ times. Without clustering and a definition of subgoal, it seems impossible to solve this kind of goal.

Then, **GMCTS** used efficiently the results of clustering for solving tasks. GMCTS has built the memory (required for solving the main Mash problem) but has not been the main component of the policy for making a decision.

This is the **vote** which has been the main component of the policy. The vote has been efficient for

- guiding simulations used to generate database before clustering
- accomplishing a subgoal

[12] Note that MCTS can be applied because it's not the classical Monte-Carlo Tree Search - nodes are not states but goals (The 'G' of GMCTS) and edges are not actions but selections of goal.

My two main applications are the game of Go and the Mash framework. The game of Go has a model whereas MASH is model-free. We have seen algorithms for solving the game of Go, but these algorithms were based on the model and knowledge of this game.

Go without knowledge. Go can be implemented as a stochastic game without knowledge and without model. In this case, MCTS can not be applied. RBGP should give some results but the engine would remain weak; this depends on the definition of the policy and mutations. It might be possible to apply CluVo+GMCTS on this version, since a stone can be seen as a feature which votes for a move. This would require much care on how to implement the game of Go.

Building a model for using Monte-Carlo Tree Search? Even if all algorithms presented in this thesis are based on simulations, unfortunately the same exact algorithm could not be used on both applications (i.e. the Game of Go and MASH) because the game of Go has a model which tells us, given a state and an action, in which state we arrive, whereas in MASH we are obliged to execute the action (which is expensive) to find out, and we can not go back (i.e. MASH is model-free). Although GMCTS is a Monte-Carlo Tree Search, the algorithm is different to the Monte-Carlo Tree Search used in MoGo, because the decision space of GMCTS is made from subgoals.

Would it be possible to build, in a more automatic and a more generic way, a model on MASH so as to use Monte-Carlo Tree Search more directly than in the present work? Some algorithms are able to learn a model of the problem [Bongard et al., 2006] [Schmidt and Lipson, 2009]. Simulations are an efficient tool for difficult problems with a precise model, such as delayed reward and/or partially observable games (e.g. many wins in phantom-go [Cazenave and Borsboom, 2007] and [Cazenave, 2006]). My work already guides simulations with macro-actions and above all integrates a MCTS in a context model-free and slow, via the learning on the fly, in a model.

Monte-Carlo Tree Search and a lot of other algorithms could be applied and therefore, numerous new doors would be open.

Index

”Blue flag then red flag“ problem, 7
“10 flags“ problem, 9

A*, 26
Abstract Proof Search, 63
Action, 4
Activated feature, 134
Agglomerative clustering, 142
Alpha-Beta, 24
  An implementation of, 25
AMAF, 47
Approach move, 60
APS, 63
Atari, 60

Bernstein bounds, 115
Bernstein race, 116
Best-first search, 26
BFS, 29
Breadth-first search, 29
Brute force, 16

Capturing race, 59
Categorization of decisions, 135
  Final, 135
  Inverse, 135
  Periodic, 135
  Stationary, 135
Chess (game of), 14
Clustering, 139
CluVo, 139
  Generate database
    First iteration, 140
    Generation, 141
  Clustering, 142
    Metric, 143

Deactivated feature, 134
Death/life problem, 60
Decision, 4
Depth-first search, 28
DFS, 28
Direct Policy Search, 22
DP, 18
DPS, 22
Dynamic Programming, 18

Expert knowledge, 16
Explorer, 63

Fast tree parallelization, 55
Feature, 4
Feature in MASH, 139
Final decision, 135
Final state, 4

Genetic programming, 113
Global contextual Monte-Carlo, 81
GMCTS
  Algorithm, 149
  Policy of, 150
Go (game of), 6
GoldenEye, 64
  Building the tree, 73
  Choice of the leaf, 81
  Evaluation of a leaf, 82
  Monte-Carlo simulation, 76
  Hierarchical solving, 108
GoTools, 63

GP, 113
Greedy algorithm, 17

H0, 79
H1, 83
H2, 83
Heuristic, 50
Heuristic in MASH, 8, 139
History, 5
Hoeffding bounds, 115
Horizon in GoldenEye, 76

IDDFS, 28
Inhibition, 83
Inverse decision, 135
Iterative deepening depth-first search, 28
  An implementation of, 30

Ko, 60

Letters draw, 150
Liberty, 60

Macro-Decision or Macro-Action, 145
Markov decision process, 4
MASH, 6
  Applications
    Blue flag then red flag, 7
    10 flags, 9
MC, 52
MCTS, 30, 45
Monte-Carlo Tree Search, 30, 43
  on subgoals, 147
MDP, 4
Memory, 145
Message-passing parallelization, 54
MinMax, 22
  An implementation of, 25
MoGo, 6
Multiple Simultaneous Hypothesis Testing, 114

Nalimov tablebases, 13
NegaMax, 23
Nested Monte-Carlo Search, 32
NoGo, 127
Noisy GP, 114
Non-regression testing, 113

Observation, 134
Offline learning, 5
Online learning, 5
Open problems of Go, 63

Periodic decision, 135
PN, 27
PNS, 27
Policy, 5
  Example of a parametric version, 23
  Version used for solving 10 flags problems, 137
  Version used for solving the main MASH application, 144
POMDP, 5
Progress rate, 124
Proof Number Search, 27

QLearning, 20

Racing-Based Genetic Programming, 115
RAVE, 46, 50
RBGP, 115
  Bernstein race, 116
    Compute bounds, 117
    A variant, 127
    Version applied in our experiment, 129
Reward, 4

Scalability, 49
Seki, 60
Semeai, 59

SG, 5
Shicho, 60
Simulation, 4
Slow tree parallelization, 55
  Slow root parallelization, 55
State, 4
Stationary decision, 135
Stochastic game, 5
Strategy, 5
Subgoal, 143

Terminal state, 4
Tsumego, 60

UCT, 30, 45
Upper Confidence Bound, 30

Voting scheme in CluVo+GMCTS
  Exploration, 145
  Vote, 146
Voting scheme in parallelization, 55

Zero-sum game, 5

List of Figures

1.1 The famous game between MoGo and Kim MyungWan (8p). . . 9
1.2 The 3D simulator in MASH. 10
1.3 The 10 flags application. 12
1.4 Nalimov tablebases. 16
1.5 First mate in 2 moves 17
1.6 Pattern for detecting a mate in 2 moves. 19
1.7 Second mate in 2 moves. 20
1.8 No mate in 2 moves. 21
1.9 Dynamic Programming. 22
1.10 Direct Policy Search. 26
1.11 Tree of considered moves. 27
1.12 MinMax. 36
1.13 Alpha-Beta cut-off. 37
1.14 Alpha-Beta. 37
1.15 A*. 38
1.16 Proof Number Search. 39
1.17 A labelled tree. 40
1.18 The basic MCTS process. 40
1.19 Monte-Carlo Tree Search. 41

2.1 Empty triangle and pattern matching. 49
2.2 Classical shapes in the game of Go. 50
2.3 Situations bad played by a MCTS program of Go. 53
2.4 Semeais. 54
2.5 Another situation bad played by a MCTS program of Go. . . . 55

3.1 A basic semeai. 61
3.2 Basic situations of the game of Go. 63
3.3 Closed and open semeais. 65
3.4 Caption for semeais. 66
3.5 Diagram of the resolution of semeai. 67

3.6 A situation of Go for testing the detector of semeais. 70
3.7 Different detections of semeai. 71
3.8 A pattern for detecting semeai. 72
3.9 A problem of Chess with its solution. 74
3.10 End of the game between 2 professional players O Meien and Cho Sonjin. 76
3.11 A preliminary resolution of semeai. 77
3.12 Expansion of moves. 79
3.13 An example of solving semeai. 82
3.14 Different possible ends of the semeai. 87
3.15 Graphic about mean number of simulations for solving completely a semeai. 92
3.16 Graphic about the number of simulations for solving completely a semeai. 94
3.17 Graphic about the execution time and memory used for solving a semeai. 95
3.18 Semeai for evaluating a parameter. 97
3.19 The raccoon semeai. 98
3.20 2 semeais of level 5d. 101
3.21 Statistics about the resolution of a semeai. 102
3.22 2 semeais of level 6d. 103
3.23 Accuracy of the score during the resolution of a semeai. 104
3.24 3 failure cases of full resolution. 105
3.25 A critical situation lost by MoGo against a 9P player. 109

4.1 2 good moves in spite of bad shapes. 121
4.2 A random pattern and its matchings. 122
4.3 Preliminary results of the study of λ on MoGo. 130
4.4 Graphics about the evolution of NoGo+mutations against NoGo. 132

5.1 3 different views of the avatar. 136
5.2 Results of DPS with Categorization of Decisions on 10 flags application. 140
5.3 Letters distribution. 153
5.4 Poor and rich letters draw. 153
5.5 Results of GMCTS on LD(20,100,10). 155
5.6 Results of GMCTS on LD(100,1000,10). 155

List of Tables

1.1 Complexity of different applications. 15

2.1 Scalability of MCTS for the game of Go. 51
2.2 Scalability of MCTS for the game of Havannah. 52
2.3 Results of MoGo combined with a solver of semeai. 56
2.4 Experiments showing the speed-up of "slow-tree parallelization" in 9x9 and 19x19 Go. 58
2.5 The very good success rate of slow tree parallelization versus very slow tree parallelization. 58

3.1 The 10 semeais from Yoji Ojima. 90
3.2 Results of GoldenEye on semeais from Yoji Ojima. 91
3.3 Results of MoGo on semeais from Yoji Ojima. 91
3.4 Impact of the exploitation term in the full resolution of semeais. 91
3.5 Number of simulations for a full resolution of 3 semeais. 93
3.6 Resolution of a semeai for evaluating a parameter. 97
3.7 Resolutions on the raccoon semeai 99
3.8 Resolution of a semeai on different sizes of goban. 101

4.1 Mutations found by RBGP in 9x9. 124

5.1 Results of clustering on Letter's Draw application. 156
5.2 Results of GMCTS on MASH application. 157

Bibliography

[Allis et al., 1994] Allis, L. V., van der Meulen, M., and van den Herik, H. J. (1994). Proof-number search. *Artif. Intell.*, 66(1):91–124.

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

[Audibert et al., 2006] Audibert, J.-Y., Munos, R., and Szepesvari, C. (2006). Use of variance estimation in the multi-armed bandit problem. In *NIPS 2006 Workshop on On-line Trading of Exploration and Exploitation*.

[Audouard et al., 2009] Audouard, P., Chaslot, G., Hoock, J.-B., Perez, J., Rimmel, A., and Teytaud, O. (2009). Grid coevolution for adaptive simulations; application to the building of opening books in the game of go. In *Proceedings of EvoGames*.

[Auer et al., 2002] Auer, P., Cesa-Bianchi, N., and Fischer, P. (2002). Finite time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256.

[Auger and Hansen, 2006] Auger, A. and Hansen, N. (2006). Reconsidering the progress rate theory for evolution strategies in finite dimensions. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 445–452, New York, NY, USA. ACM.

[Auger and Teytaud,] Auger, A. and Teytaud, O. Continuous lunches are free plus the design of optimal optimization algorithms. *Algorithmica*, page 2009.

[Baba et al., 2011] Baba, S., Joe, Y., Iwasaki, A., and Yokoo, M. (2011). Real-time solving of quantified csps based on monte-carlo game tree search. In *IJCAI*, pages 655–661.

[Baier and Winands, 2012] Baier, H. and Winands, M. H. M. (2012). Nested monte-carlo tree search for online planning in large mdps. In *ECAI*, pages 109–114.

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition.

[Berliner, 1979] Berliner, H. J. (1979). The b* tree search algorithm: A best-first proof procedure. *Artif. Intell.*, 12(1):23–40.

[Berthier et al., 2010] Berthier, V., Doghmen, H., and Teytaud, O. (2010). Consistency Modifications for Automatically Tuned Monte-Carlo Tree Search. In *Proceedings of Lion4*, page 14.

[Beyer, 2001] Beyer, H.-G. (2001). *The Theory of Evolution Strategies*. Natural Computing Series. Springer, Heidelberg.

[Bibai et al., 2010] Bibai, J., Savéant, P., Schoenauer, M., and Vincent, V. (2010). An Evolutionary Metaheuristic for Domain-Independent Satisficing Planning. In Brafman, R., Geffner, H., Hoffmann, J., and Kautz, H., editors, *20th International Conference on Automated Planning and Scheduling-ICAPS2010*, pages 15–25, Toronto, Canada. AAAI Press.

[Bongard et al., 2006] Bongard, J., Zykov, V., and Lipson, H. (2006). Resilient machines through continuous self-modeling. *Science*, 314:1118–1121.

[Bourki et al., 2010] Bourki, A., Chaslot, G., Coulm, M., Danjean, V., Doghmen, H., Hérault, T., Hoock, J.-B., Rimmel, A., Teytaud, F., Teytaud, O., Vayssière, P., and Yu, Z. (2010). Scalability and Parallelization of Monte-Carlo Tree Search. In *The International Conference on Computers and Games 2010*, Kanazawa, Japon.

[Bouzy, 2003] Bouzy, B. (2003). Mathematical morphology applied to computer go.

[Bouzy and Chaslot, 2005] Bouzy, B. and Chaslot, G. (2005). Bayesian generation and integration of k-nearest-neighbor patterns for 19x19 go. In *G. Kendall and Simon Lucas, editors, IEEE 2005 Symposium on Computational Intelligence in Games, Colchester, UK*, pages 176–181.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine Learning*, 45(1):5–32.

[Brown, 1951] Brown, G. W. (1951). Iterative solutions of games by fictitious play. In *Activity Analysis of Production and Allocation*, pages 374–376. Wiley.

[Browne et al., 2012] Browne, C., Powley, E. J., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012). A survey of monte carlo tree search methods. *IEEE Trans. Comput. Intellig. and AI in Games*, 4(1):1–43.

[Bruegmann, 1993] Bruegmann, B. (1993). Monte carlo go. *Unpublished*.

[Buşoniu et al., 2010] Buşoniu, L., Ernst, D., De Schutter, B., and Babuška, R. (2010). Online least-squares policy iteration for reinforcement learning control. In *Proceedings of the 2010 American Control Conference*, pages 486–491, Baltimore, Maryland.

[Cazenave, 2000] Cazenave, T. (2000). Abstract proof search. In *Computers and Games*, pages 39–54.

[Cazenave, 2001] Cazenave, T. (2001). Iterative widening. In *IJCAI*, pages 523–528.

[Cazenave, 2004] Cazenave, T. (2004). Generalized widening. In *ECAI*, pages 156–160.

[Cazenave, 2006] Cazenave, T. (2006). A phantom-go program. In van den Herik, H. J., chin Hsu, S., sheng Hsu, T., and Donkers, H. H. L. M., editors, *ACG*, volume 4250 of *Lecture Notes in Computer Science*, pages 120–125. Springer.

[Cazenave, 2009] Cazenave, T. (2009). Nested monte-carlo search. In Boutilier, C., editor, *IJCAI*, pages 456–461.

[Cazenave and Borsboom, 2007] Cazenave, T. and Borsboom, J. (2007). Golois wins phantom go tournament. *ICGA Journal*, 30(3):165–166.

[Cazenave and Jouandeau, 2007] Cazenave, T. and Jouandeau, N. (2007). On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101.

[Cazenave and Teytaud, 2012] Cazenave, T. and Teytaud, F. (2012). Application of the nested rollout policy adaptation algorithm to the traveling salesman problem with time windows. In *LION*, pages 42–54.

[Chaslot et al., 2009] Chaslot, G., Fiter, C., Hoock, J.-B., Rimmel, A., and Teytaud, O. (2009). Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona, Espagne. Springer.

[Chaslot et al., 2009] Chaslot, G., Hoock, J.-B., Teytaud, F., and Teytaud, O. (2009). On the huge benefit of quasi-random mutations for multimodal optimization with application to grid-based tuning of neurocontrollers. In *ESANN*, Bruges Belgium.

[Chaslot et al., 2006] Chaslot, G., Saito, J.-T., Bouzy, B., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2006). Monte-Carlo Strategies for Computer Go. In Schobbens, P.-Y., Vanhoof, W., and Schwanen, G., editors, *Proceedings of the 18th BeNeLux Conference on Artificial Intelligence, Namur, Belgium*, pages 83–91.

[Chaslot et al., 2007] Chaslot, G., Winands, M., Uiterwijk, J., van den Herik, H., and Bouzy, B. (2007). Progressive strategies for monte-carlo tree search. In Wang, P. et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd.

[Chaslot et al., 2008] Chaslot, G., Winands, M., and van den Herik, H. (2008). Parallel Monte-Carlo Tree Search. In *Proceedings of the Conference on Computers and Games 2008 (CG 2008).*

[Chen and Zhang, 2006] Chen, K.-H. and Zhang, P. (2006). A new heuristic search algorithm for capturing problems in go. In *Computers and Games*, pages 26–36.

[Chou et al., 2011] Chou, C.-W., Teytaud, O., and Yen, S.-J. (2011). Revisiting Monte-Carlo Tree Search on a Normal Form Game: NoGo. In *EvoGames 2011*, volume 6624 of *Lecture Notes in Computer Science*, pages 73–82, Turino, Italy. Springer-Verlag.

[Coulom, 2006] Coulom, R. (2006). Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy.*

[Coulom, 2007] Coulom, R. (2007). Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands.*

[Coulom, 2009] Coulom, R. (2009). Criticality: a monte-carlo heuristic for go programs. Invited talk at the University of Electro-Communications, Tokyo, Japan.

[Crasmaru, 1999] Crasmaru, M. (1999). On the complexity of Tsume-Go. 1558:222–231.

[Dai et al., 2009] Dai, P., Mausam, and Weld, D. S. (2009). Focused topological value iteration. In *ICAPS*.

[de Mesmay et al., 2009] de Mesmay, F., Rimmel, A., Voronenko, Y., and Püschel, M. (2009). Bandit-based optimization on graphs with application to library performance tuning. In Danyluk, A. P., Bottou, L., and Littman, M. L., editors, *ICML*, volume 382 of *ACM International Conference Proceeding Series*, page 92. ACM.

[Defays, 1977] Defays, D. (1977). An efficient algorithm for a complete link method. *Comput. J.*, 20(4):364–366.

[Dubout and Fleuret, 2011] Dubout, C. and Fleuret, F. (2011). Tasting families of features for image classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 929–936.

[Dulac-Arnold et al., 2012] Dulac-Arnold, G., Denoyer, L., Preux, P., and Gallinari, P. (2012). Fast reinforcement learning with large action sets using error-correcting output codes for mdp factorization. In Flach, P., De Bie, T., and Cristianini, N., editors, *Machine Learning and Knowledge Discovery in Databases*, volume 7524 of *Lecture Notes in Computer Science*, pages 180–194. Springer Berlin / Heidelberg.

[Ernst et al., 2005] Ernst, D., Geurts, P., Wehenkel, L., and Littman, L. (2005). Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6:503–556.

[Freund and Schapire, 1995] Freund, Y. and Schapire, R. E. (1995). A decision-theoretic generalization of on-line learning and an application to boosting. In *Proceedings of the Second European Conference on Computational Learning Theory*, EuroCOLT '95, pages 23–37, London, UK, UK. Springer-Verlag.

[Gaudel et al., 2010] Gaudel, R., Hoock, J.-B., Pérez, J., Sokolovska, N., and Teytaud, O. (2010). A Principled Method for Exploiting Opening Books. In *International Conference on Computers and Games*, Kanazawa, Japon.

[Gelly et al., 2008] Gelly, S., Hoock, J. B., Rimmel, A., Teytaud, O., and Kalemkarian, Y. (2008). The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203. To appear.

[Gelly and Silver, 2007] Gelly, S. and Silver, D. (2007). Combining online and offline knowledge in UCT. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 273–280, New York, NY, USA. ACM Press.

[Hart et al., 1968] Hart, P., Nilsson, N., and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107.

[Heidrich-Meisner and Igel, 2008] Heidrich-Meisner, V. and Igel, C. (2008). Evolution strategies for direct policy search. In *PPSN*, pages 428–437.

[Heineman et al., 2009] Heineman, G. T., Pollice, G., and Selkow, S. M. (2009). *Algorithms in a nutshell - a desktop quick reference*. O'Reilly.

[Hill and Marty, 2008] Hill, M. D. and Marty, M. R. (2008). Amdahl's law in the multicore era. *Computer*, 41(7):33–38.

[Holland, 1973] Holland, J. H. (1973). Genetic algorithms and the optimal allocation of trials. *SIAM J. Comput.*, 2(2):88–105.

[Hoock and Bibai, 2012] Hoock, J.-B. and Bibai, J. (2012). Solving a Goal-planning task in the MASH project. In *The 2012 Conference on Technologies and Applications of Artificial Intelligence (TAAI 2012)*, Tainan, Taïwan, Province De Chine.

[Hoock et al., 2010] Hoock, J.-B., Lee, C.-S., Rimmel, A., Teytaud, F., Teytaud, O., and Wang, M.-H. (2010). Intelligent Agents for the Game of Go. *IEEE Computational Intelligence Magazine*.

[Hoock and Teytaud, 2010] Hoock, J.-B. and Teytaud, O. (2010). Bandit-Based Genetic Programming. In *13th European Conference on Genetic Programming*, Istanbul, Turkey. Springer.

[Hoock and Teytaud, 2011] Hoock, J.-B. and Teytaud, O. (2011). Progress Rate in Noisy Genetic Programming for Choosing λ. In *Artificial Evolution*, Angers, France.

[Hunter, 2003] Hunter, R. (2003). *Counting Liberties and Winning Capturing Races*. Slate and Shell.

[Kato, 2009] Kato, H. (2009). Post on the computer-go mailing list, october.

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

[Kishimoto and Müller, 2003] Kishimoto, A. and Müller, M. (2003). Df-pn in go: An application to the one-eye problem. In *ACG*, pages 125–142.

[Knuth and Moore, 1975] Knuth, D. E. and Moore, R. W. (1975). An analysis of alpha-beta pruning. *Artif. Intell.*, 6(4):293–326.

[Kocsis and Szepesvari, 2006] Kocsis, L. and Szepesvari, C. (2006). Bandit-based monte-carlo planning. In *ECML'06*, pages 282–293.

[Korf, 1985] Korf, R. E. (1985). Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27:97–109.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by means of Natural Evolution.* MIT Press, Massachusetts.

[Lai and Robbins, 1985] Lai, T. and Robbins, H. (1985). Asymptotically efficient adaptive allocation rules. *Advances in Applied Mathematics*, 6:4–22.

[Lee et al., 2009] Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., and Hong, T.-P. (2009). The Computational Intelligence of MoGo Revealed in Taiwan's Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in games*, pages 73–89.

[Lichtenstein and Sipser, 1980] Lichtenstein, D. and Sipser, M. (1980). Go is polynomial-space hard. *J. ACM*, 27(2):393–401.

[Mcgovern et al., 1997] Mcgovern, A., Sutton, R. S., and Fagg, A. H. (1997). Roles of macro-actions in accelerating reinforcement learning. In *In Grace Hopper Celebration of Women in Computing*, pages 13–18.

[Méhat and Cazenave, 2010] Méhat, J. and Cazenave, T. (2010). Combining uct and nested monte carlo search for single-player general game playing. *IEEE Trans. Comput. Intellig. and AI in Games*, 2(4):271–277.

[Menache et al., 2002] Menache, I., Mannor, S., and Shimkin, N. (2002). Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, pages 295–306, London, UK, UK. Springer-Verlag.

[Mnih et al., 2008] Mnih, V., Szepesvári, C., and Audibert, J.-Y. (2008). Empirical Bernstein stopping. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 672–679, New York, NY, USA. ACM.

[Müller, 1999] Müller, M. (1999). Race to capture: Analyzing semeai in Go. In *Game Programming Workshop in Japan '99, MATSUBARA, H. (ed.), Computer Shogi Association, Tokyo, Japan.*

[Müller, 2002] Müller, M. (2002). A generalized framework for analyzing capturing races in go. In *JCIS*, pages 469–472.

[Nagai, 1998] Nagai, A. (1998). A new and/or tree search algorithm using proof number and disproof number. In Frank, I., Matsubara, H., Tajima, M., Yoshikawa, A., Grimbergen, R., and Müller, M., editors, *Complex Games Lab Workshop*. Electrotechnical Laboratory, Machine Inference Group, Tsukuba, Japan.

[Nakamura, 2008] Nakamura, T. (2008). On counting liberties in capturing races of go. In Albert, M. H. and Nowakowski, R. J., editors, *Games of No Chance III,* Proc. BIRS Workshop on Combinatorial Games, July, 2005, Banff, Alberta, Canada, MSRI Publ. Cambridge University Press, Cambridge.

[Niu and Müller, 2006] Niu, X. and Müller, M. (2006). An open boundary safety-of-territory solver for the game of go. In *Computers and Games*, pages 37–49.

[Ponsen et al., 2010] Ponsen, M. J. V., Gerritsen, G., and Chaslot, G. (2010). Integrating opponent models with monte-carlo tree search in poker. In *Interactive Decision Theory and Game Theory.*

[Prim, 1957] Prim, R. C. (1957). Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401.

[Rimmel and Teytaud, 2010] Rimmel, A. and Teytaud, F. (2010). Multiple overlapping tiles for contextual monte carlo tree search. In *EvoApplications (1)*, pages 201–210.

[Robson, 1983] Robson, J. M. (1983). The complexity of go. In *IFIP Congress*, pages 413–417.

[Rolet et al., 2009] Rolet, P., Sebag, M., and Teytaud, O. (2009). Optimal active learning through billiards and upper confidence trees in continous domains. In *Proceedings of the ECML conference*.

[Rolet and Teytaud, 2010] Rolet, P. and Teytaud, O. (2010). Bandit-based Estimation of Distribution Algorithms for Noisy Optimization: Rigorous Runtime Analysis. In *Lion4*, Venice Italie.

[Rosin, 2011] Rosin, C. D. (2011). Nested rollout policy adaptation for monte carlo tree search. In *IJCAI*, pages 649–654.

[Rummery and Niranjan, 1994] Rummery, G. A. and Niranjan, M. (1994). On-line q-learning using connectionist systems. Technical report.

[Saito et al., 2007] Saito, J.-T., Chaslot, G., Uiterwijk, J. W. H. M., and Van Den Herik, H. J. (2007). Monte-carlo proof-number search for computer go. In *Proceedings of the 5th international conference on Computers and games*, CG'06, pages 50–61.

[Samothrakis et al., 2011] Samothrakis, S., Robles, D., and Lucas, S. M. (2011). Fast approximate max-n monte carlo tree search for ms pac-man. *IEEE Trans. Comput. Intellig. and AI in Games*, 3(2):142–154.

[Sato et al., 2010] Sato, Y., Takahashi, D., and Grimbergen, R. (2010). A shogi program based on monte-carlo tree search. *ICGA Journal*, 33(2):80–92.

[Savaresi et al., 2002] Savaresi, S. M., Boley, D. L., Bittanti, S., and Gazzaniga, G. (2002). Cluster selection in divisive clustering algorithms. In *SIAM Internation Conference on Data Mining*, pages 299–314.

[Schadd et al., 2008] Schadd, M. P. D., Win, M. H. M., Herik, H. J. V. D., b. Chaslot, G. M. J., and Uiterwijk, J. W. H. M. (2008). Single-player monte-carlo tree search. In *In Computers and Games, volume 5131 of Lecture Notes in Computer Science*, pages 1–12. Springer.

[Schmidhuber and Wahnsiedler, 1992] Schmidhuber, J. and Wahnsiedler, R. (1992). Planning simple trajectories using neural subgoal generators. In *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*, pages 196–199. MIT Press.

[Schmidt and Lipson, 2009] Schmidt, M. and Lipson, H. (2009). Distilling Free-Form Natural Laws from Experimental Data. *Science*, 324:81–85.

[Seo et al., 2001] Seo, M., Iida, H., and Uiterwijk, J. W. H. M. (2001). The pn*-search algorithm: Application to tsume-shogi. *Artif. Intell.*, 129(1-2):253–277.

[Silver and Veness, 2010] Silver, D. and Veness, J. (2010). Monte-carlo planning in large pomdps. In *NIPS*, pages 2164–2172.

[Singh et al., 1996] Singh, S., Sutton, R. S., and Kaelbling, P. (1996). Reinforcement learning with replacing eligibility traces. In *Machine Learning*, pages 123–158.

[Stolle and Precup, 2002] Stolle, M. and Precup, D. (2002). Learning options in reinforcement learning. In *Proceedings of the 5th International Symposium on Abstraction, Reformulation and Approximation*, pages 212–223, London, UK, UK. Springer-Verlag.

[Tanabe et al., 2009] Tanabe, Y., Yoshizoe, K., and Imai, H. (2009). A study on security evaluation methodology for image-based biometrics authentication systems. In *Biometrics: Theory, Applications, and Systems, 2009. BTAS'09. IEEE 3rd International Conference on*, pages 1–6. IEEE.

[Teytaud and Teytaud, 2009] Teytaud, F. and Teytaud, O. (2009). Creating an Upper-Confidence-Tree program for Havannah. In *ACG 12*, Pamplona Espagne.

[Teytaud and Flory, 2011] Teytaud, O. and Flory, S. (2011). Upper Confidence Trees with Short Term Partial Information. In *EvoGames 2011*, volume 6624 of *Lecture Notes in Computer Science*, pages 153–162, Turino, Italie. Springer.

[Vilà and Cazenave, 2003] Vilà, R. and Cazenave, T. (2003). When one eye is sufficient: A static classification. In *ACG*, pages 109–124.

[Wang et al., 2008] Wang, Y., Audibert, J.-Y., and Munos, R. (2008). Algorithms for infinitely many-armed bandits. In *Advances in Neural Information Processing Systems*, volume 21.

[Wang and Gelly, 2007] Wang, Y. and Gelly, S. (2007). Modifications of UCT and sequence-like simulations for Monte-Carlo Go. In *IEEE Symposium on Computational Intelligence and Games, Honolulu, Hawaii*, pages 175–182.

[Watkins and Dayan, 1992] Watkins, C. J. C. H. and Dayan, P. (1992). Q-learning. *Machine Learning*, 8(3-4):279–292.

[Winands et al., 2002] Winands, M. H. M., Uiterwijk, J. W. H. M., and van den Herik, H. J. (2002). Pds-pn: A new proof-number search algorithm. In *Computers and Games*, pages 61–74.

[Wolf, 1994] Wolf, T. (1994). The program Gotools and its computer-generated tsume go database. In *1st Game Programming Workshop in Japan, Hakone, 1994*.

[Wolf, 2012] Wolf, T. (2012). A Classification of Semeai with Approach Moves.

[Wolfe and Barto, 2005] Wolfe, A. P. and Barto, A. G. (2005). Identifying useful subgoals in reinforcement learning by local graph partitioning. In *In Proceedings of the Twenty-Second International Conference on Machine Learning*, pages 816–823.

[Zhang and Chen, 2008] Zhang, P. and Chen, K.-H. (2008). Monte carlo go capturing tactic search. *New Mathematics and Natural Computation (NMNC)*, 04(03):359–367.

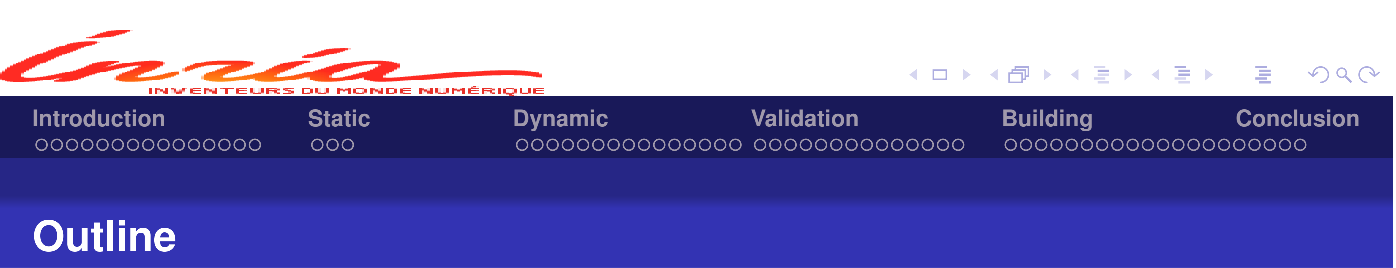
Annex :

Thesis defense slides

Contributions to Simulation-based High-dimensional Sequential Decision Making.

Generating prior knowledge in Monte Carlo Tree Search.

J-B. Hoock
Thesis defense
Under the supervision of Olivier Teytaud.
Wednesday, April 10, 2013



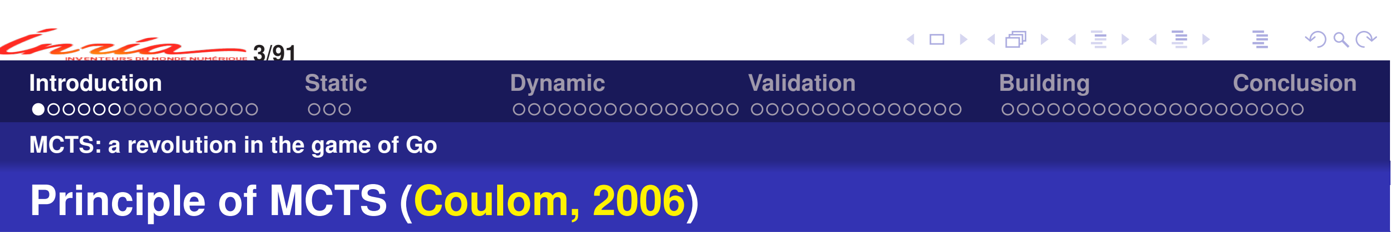
Introduction Static Dynamic Validation Building Conclusion

Outline

1. Introduction
2. Adding static prior knowledge (= patterns)
3. Adding dynamic prior knowledge (= local solver)
4. Validating prior knowledge (RBGP)
5. Building prior knowledge (CluVo+GMCTS)
6. Conclusion

Outline

1. ***Introduction***
   - Monte Carlo Tree Search
   - Parallelization
   - Different ways for generating prior knowledge in MCTS
2. Adding static prior knowledge (= patterns)
3. Adding dynamic prior knowledge (= local solver)
4. Validating prior knowledge (RBGP)
5. Building prior knowledge (CluVo+GMCTS)
6. Conclusion



Principle of MCTS (Coulom, 2006)

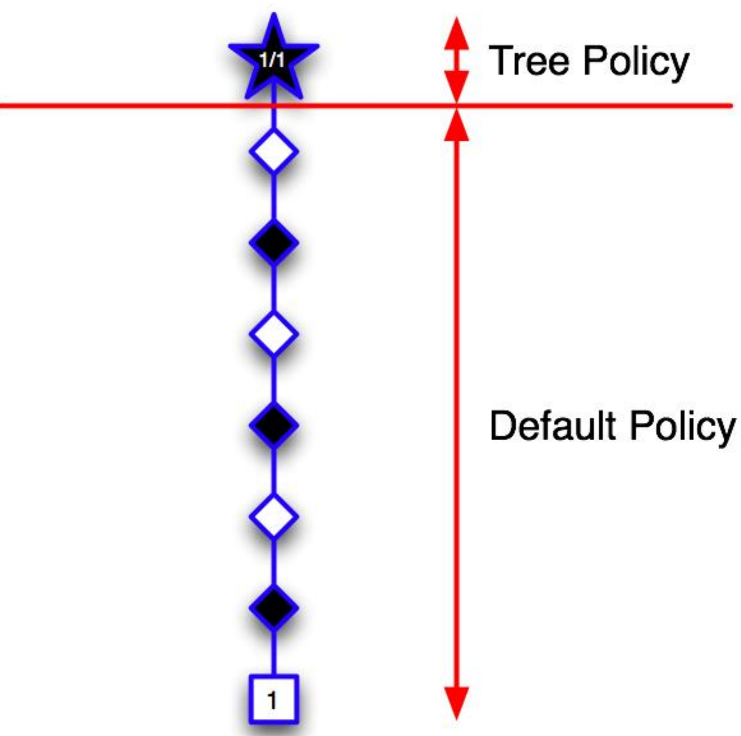
Principle

- Construction of an unbalanced subtree of possible futures
- Evaluation through Monte-Carlo simulations
- Use a formula to bias the subtree
  - typically, a bandit formula (e.g. UCB1 - Auer & al., 02)

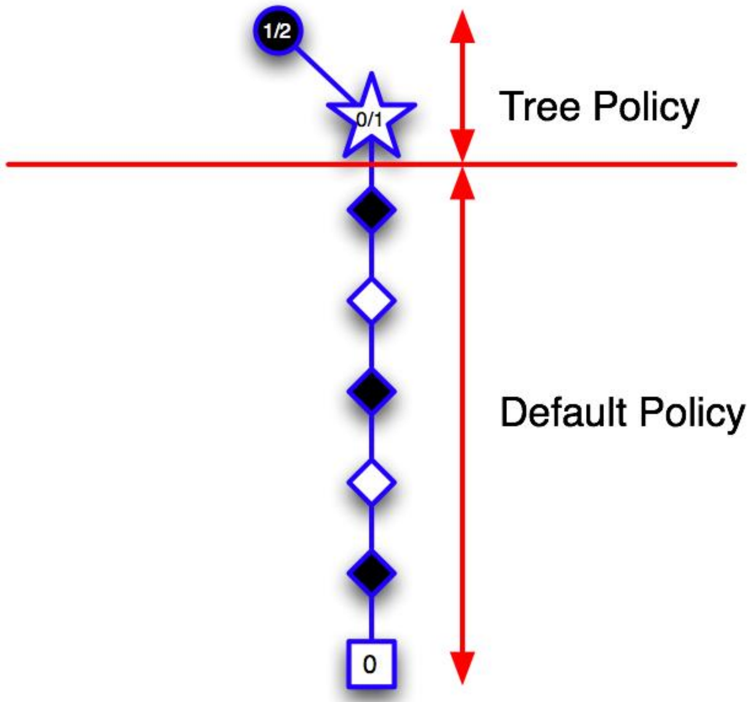
3 main steps

- Descent in the subtree
- Evaluation of the leaves
- Growth and update of the subtree

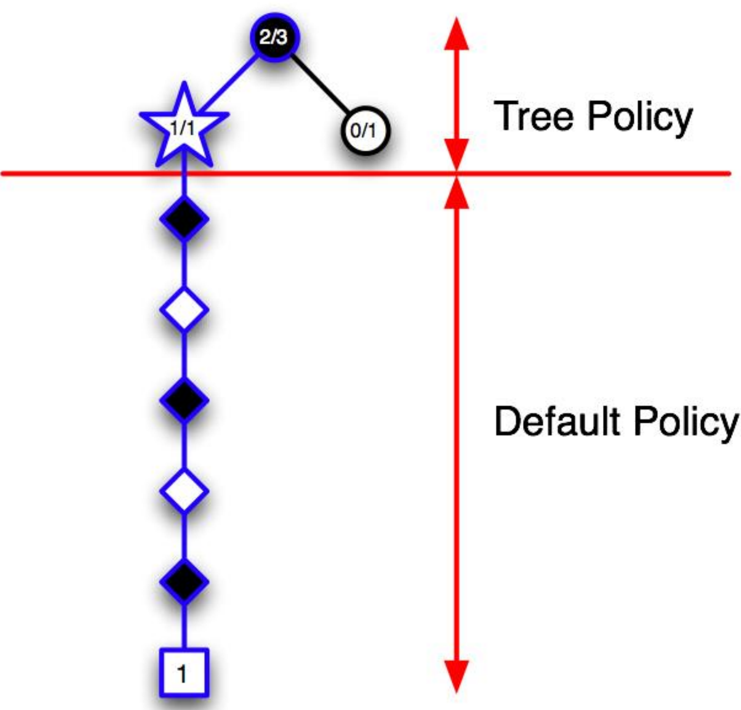
Principle of MCTS - after a first simulation



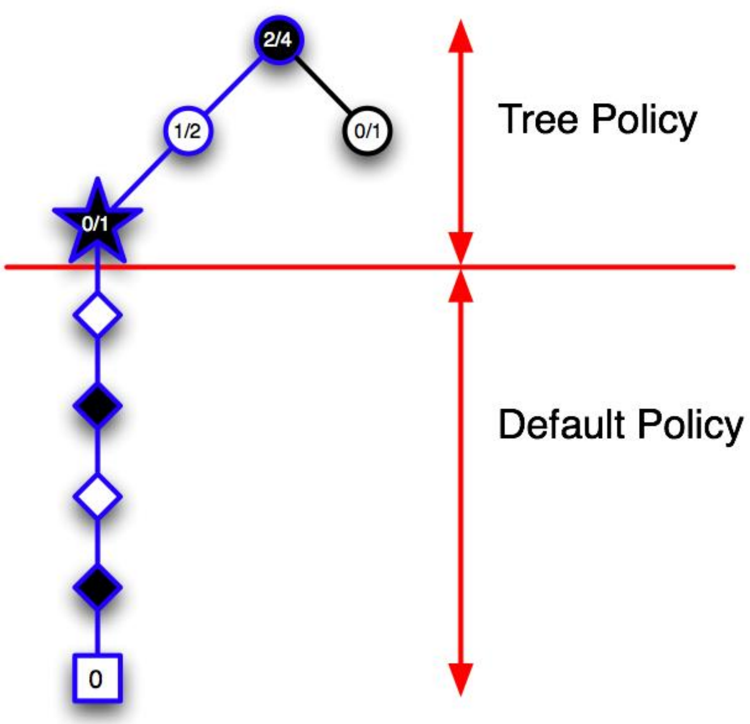
Principle of MCTS - after a second simulation



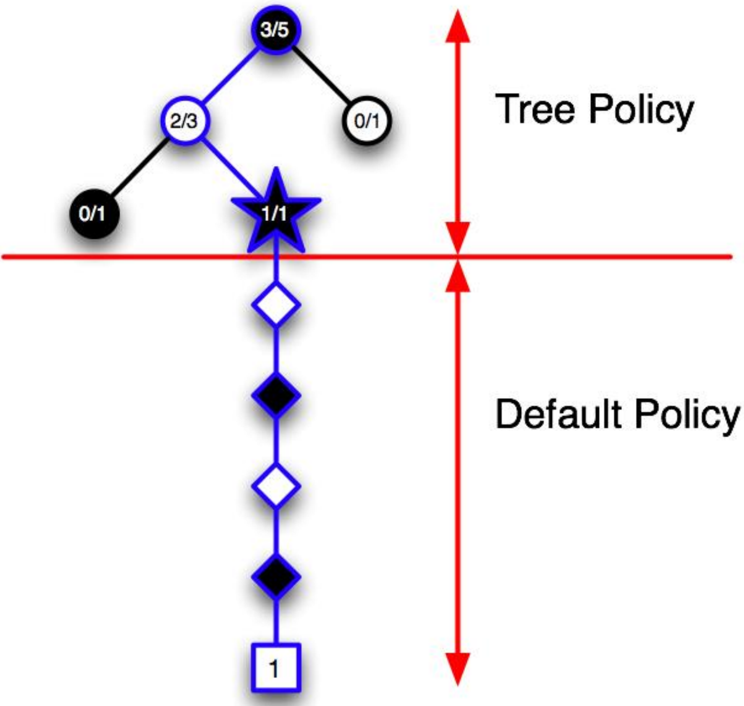
Principle of MCTS - after a third simulation

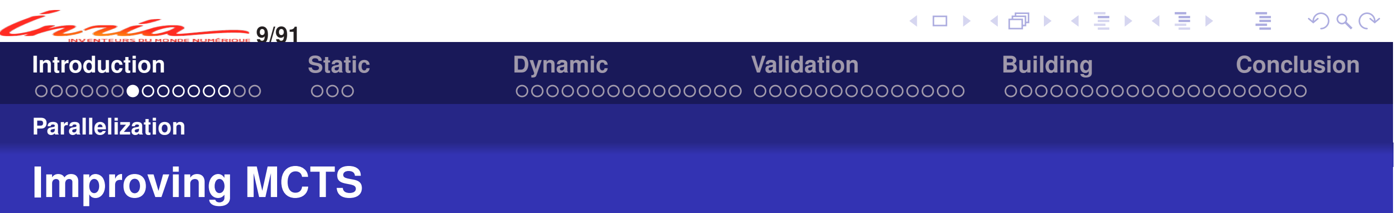


Principle of MCTS - after a fourth simulation



Principle of MCTS - after a fifth simulation





Improving MCTS

- Limited efficiency of parallelization
- Adding prior knowledge ==> not so easy (gnugo counter-example)
- This thesis: adding prior knowledge

Example: Slow tree parallelization (Gelly & al, 2008)

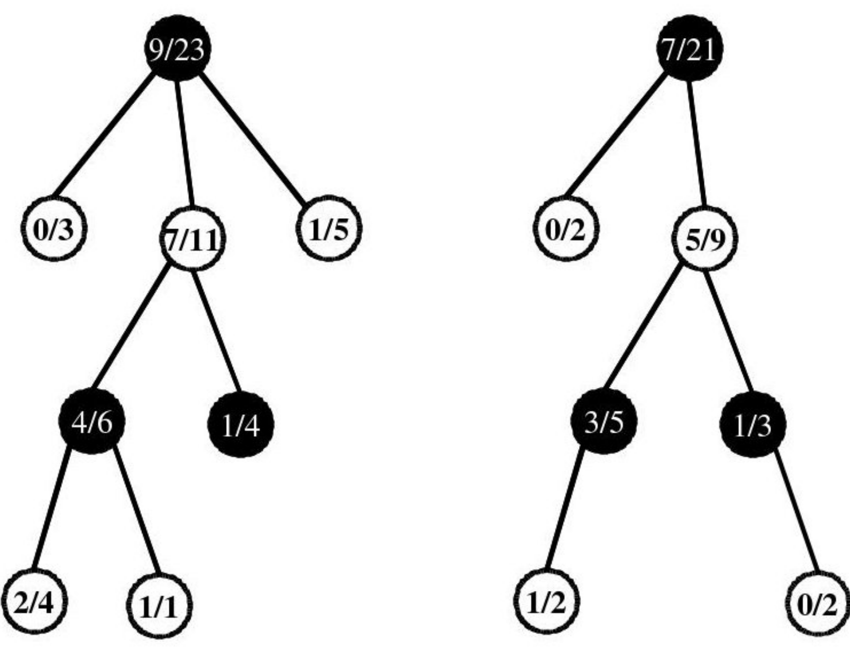
Principle (Cazenave & Jouandeau, 2007)

Each computation node builds his own tree.
Every T_0 second:

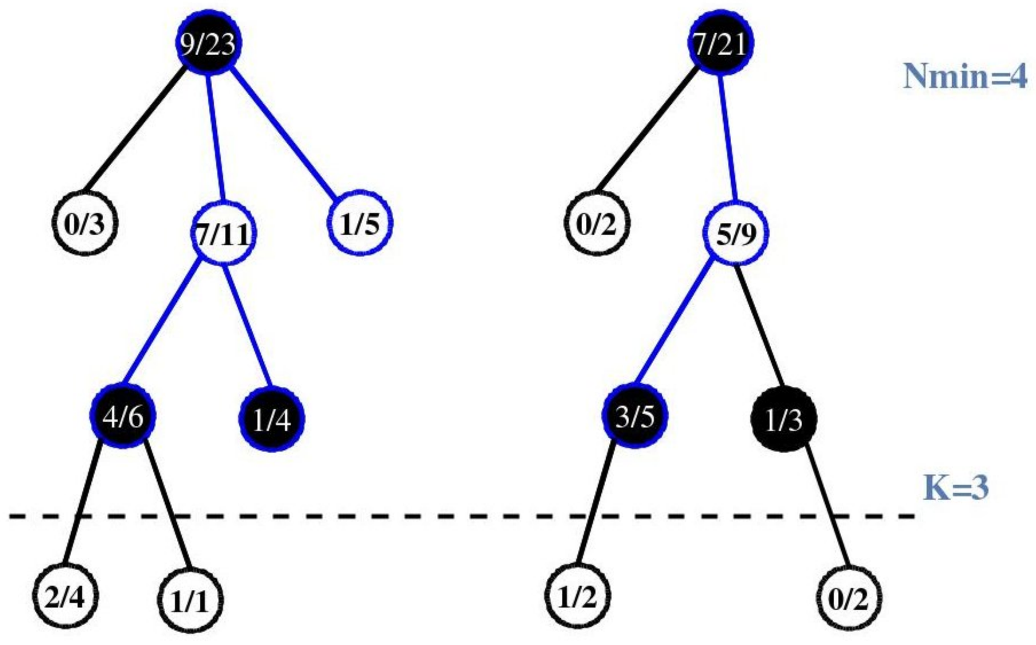
- average statistics in the tree for all nodes
  - with a depth $\leq K$
  - with at least N_{min} simulations

Variant:
slow root parallelization: only the root is considered

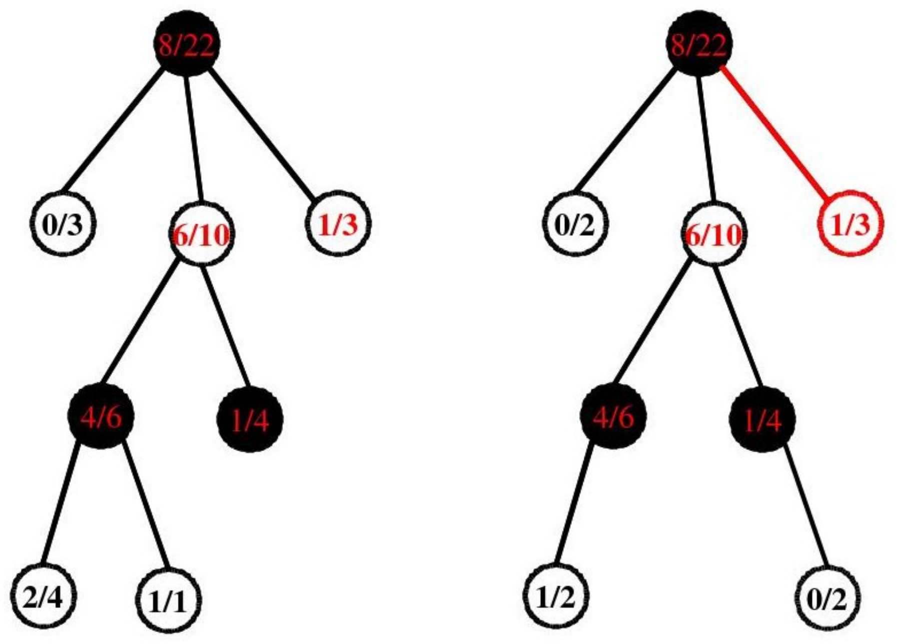
Principle of Slow tree parallelization - 2 trees



Principle of Slow tree parallelization - conditions for merging



Principle of Slow tree parallelization - results after merging

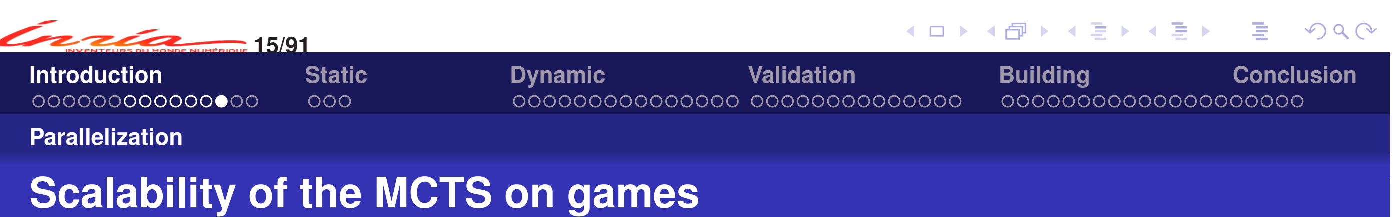


Results on MoGo, our MCTS engine of Go (Bourki & al, CG 2010)

Speed-up of slow tree parallelization

Configuration of game	Winning rate in 9x9	Winning rate in 19x19
32 against 1	75.85 ± 2.49 %	95.10±01.37 %
32 against 2	66.30 ± 2.82 %	82.38±02.74 %
32 against 4	62.63 ± 2.88 %	73.49±03.42 %
32 against 8	59.64 ± 2.93 %	63.07±04.23 %
32 against 16	52.00 ± 3.01 %	63.15±05.53 %
32 against 32	48.91 ± 3.00 %	48.00±09.99 %

- Plateau reached around 16 cores in 9x9
- Regular improvement in 19x19



Scalability of the MCTS on games

games

ability to play better when additional computational power or time is provided

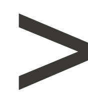
Scalability of the MCTS for the game of Go

N =Number of simulations	Success rate of 2N simulations against N simulations in 9x9 Go	Success rate of 2N simulations against N simulations in 19x19 Go
1 000	71.1 ± 0.1 %	90.5 ± 0.3 %
4 000	68.7 ± 0.2	84.5 ± 0.3 %
16 000	66.5 ± 0.9 %	80.2 ± 0.4 %
256 000	61.0 ± 0.2 %	58.5 ± 1.7 %

Summary

- The 2N vs N performance decreases with N

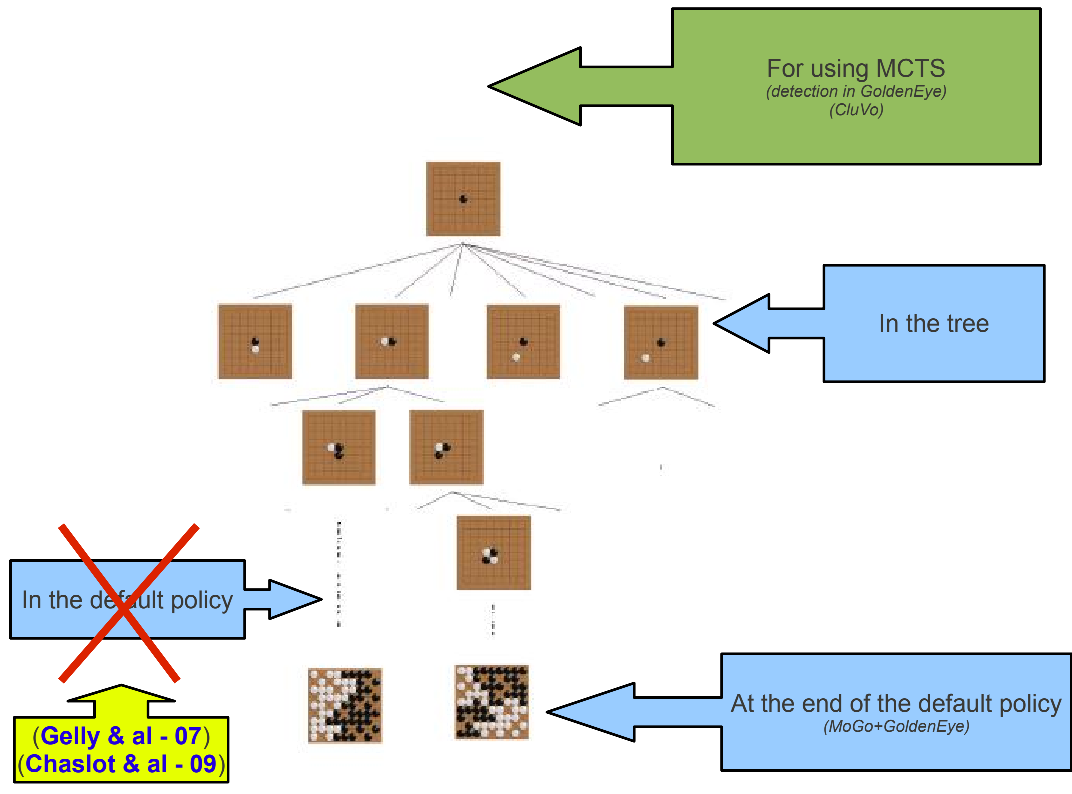
Adding prior knowledge ==> not so easy (Gelly, 07)



GnuGo (strong simulator) > Weak simulator

MCTS + GnuGo < MCTS + Weak simulator

Where will we add/build prior knowledge in MCTS?



Outline

1. Introduction
2. ***Adding static prior knowledge (= patterns)***
3. Adding dynamic prior knowledge (= local solver)
4. Validating prior knowledge (RBGP)
5. Building prior knowledge (CluVo+GMCTS)
6. Conclusion

Using exploration/exploitation formula for biasing the subtree

Formula in MoGo

$$score(d) = \alpha\hat{p}(d) + \beta\widehat{\hat{p}}(d) + (\gamma + \frac{C}{\log(2+n(d))})H(d)$$

3 terms

- $\hat{p}(d)$ = online estimate
- $\widehat{\hat{p}}(d)$ = RAVE values (not detailed here; *knowledge from simulations*)
- $H(d)$ = offline learning + exploration

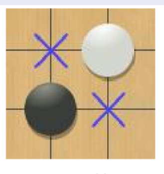
Introduce expert knowledge V_{expert} in the formula $score$

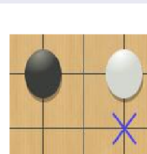
V_{expert} is added in 2 terms:

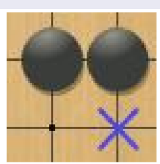
- $H(d) += V_{expert}(d)$
- roughly speaking, $\widehat{\widehat{p}}(d)$ = initialized at V_{expert} (virtual wins)

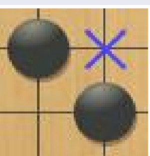
Contributions (Chaslot & al, ACG 2009)

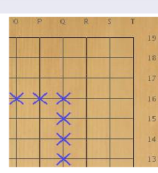
Human prior knowledge


Wall


Bad Kogeima


Empty triangle


Empty triangle


Line of influence

Static prior knowledge

- handcrafting V_{expert}
- tuning expertise weight C

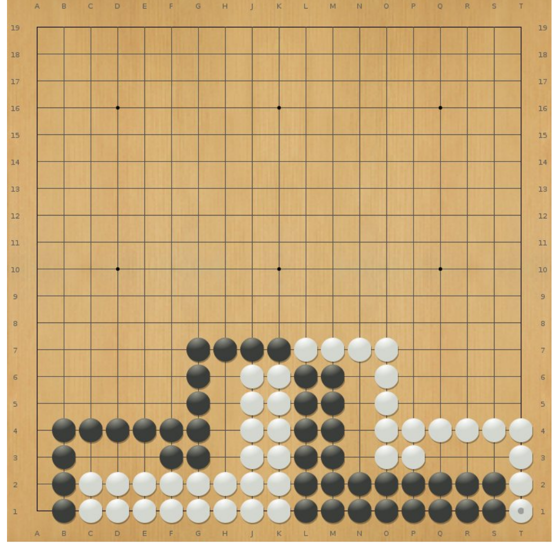
Results

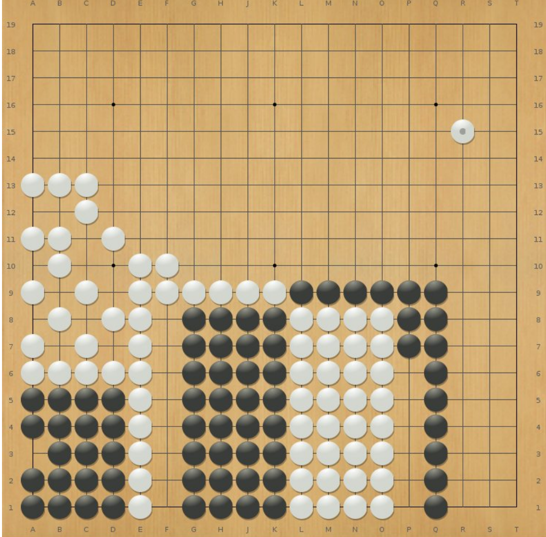
- one pattern, small improvements ($< 1\%$).
- cumulated, $63.5\% \pm 0.5$ against the baseline.

Outline

1. Introduction
2. Adding static prior knowledge (= pattern matching)
3. ***Adding dynamic prior knowledge (= local solver)***
   - Unsolved local situations
   - GoldenEye
     - Detection of this local situation
     - Resolution
     - Improving the resolution : the heuristic inhibition
   - Results
   - GoldenEye in MoGo
4. Validating prior knowledge (RBGP)
5. Building prior knowledge (CluVo+GMCTS)
6. Conclusion

Play or not in the semeai ? Unsolved situation in spite of comput. power





GoldenEye (Unpublished)

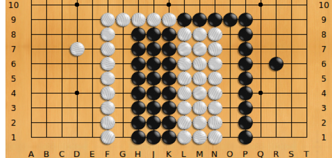
A tactical solver

- detect these situations
- solve them

GoldenEye

Detection by patterns?

- too numerous shapes
- semeai = XOR between groups'lives



A statistical approach

- $AD_{\text{sims}}(s1, s2)$ = frequency of $s1$ alive and $s2$ dead in $sims$ simulations
- $sem_{\text{sims}}(Sb, Sw) = AD_{\text{sims}}(Sb, Sw) + AD_{\text{sims}}(Sw, Sb)$
- 2 stones Sb and Sw are in semeai if:

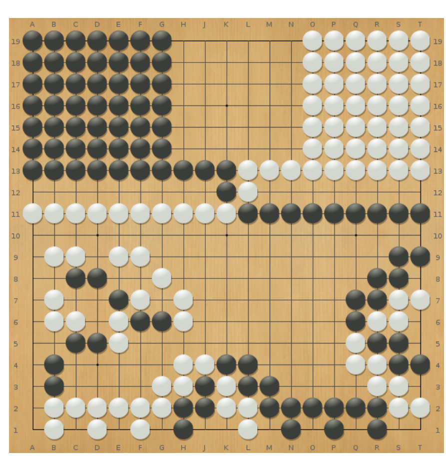
$$sem_{\text{sims}}(Sb, Sw) \geq \rho$$

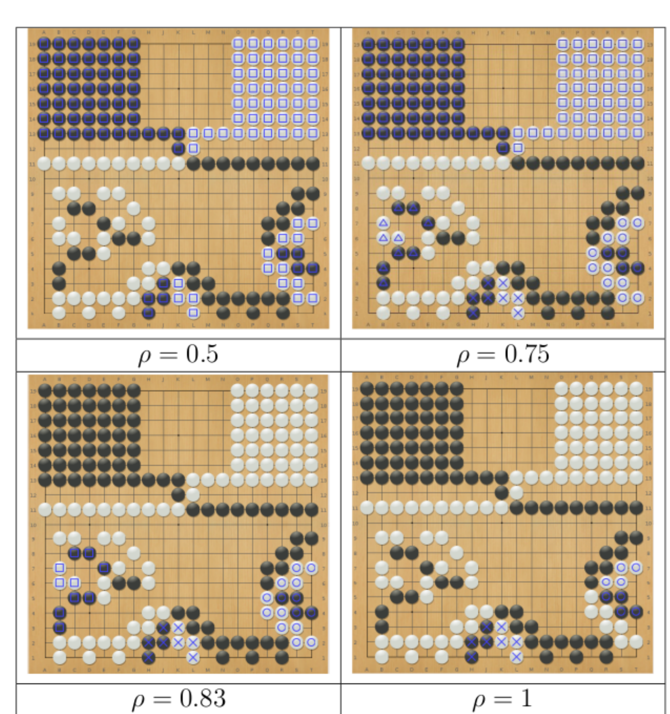
- Semeais built by aggregation of stones.

Other approach

Criticality (Coulom, 2009)

Detection depending on ρ



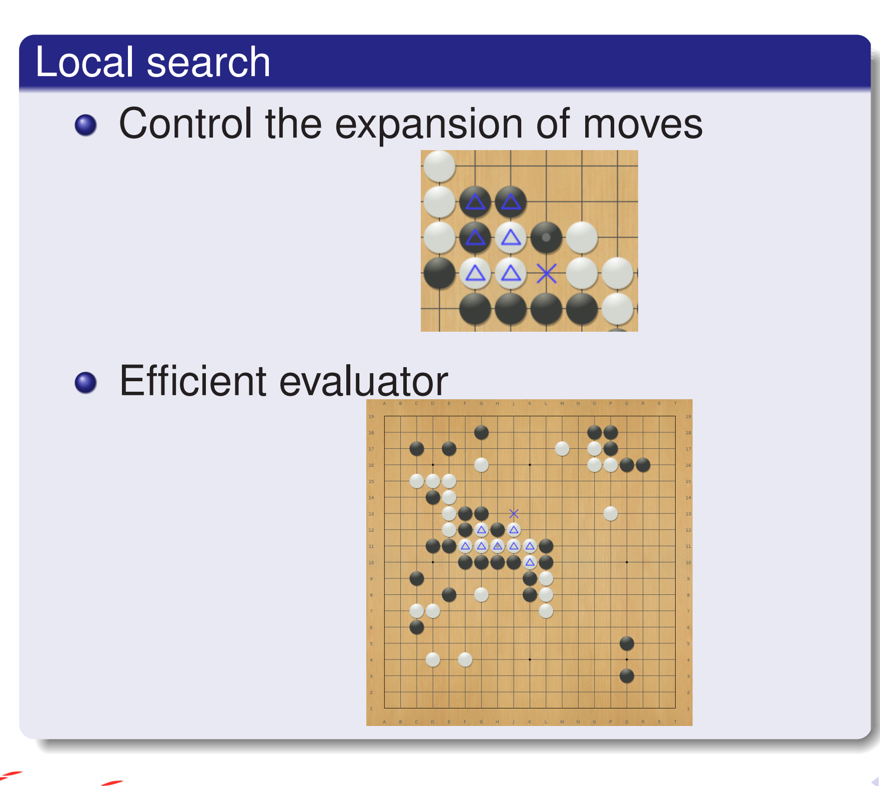


$\rho = 0.5$	$\rho = 0.75$
$\rho = 0.83$	$\rho = 1$

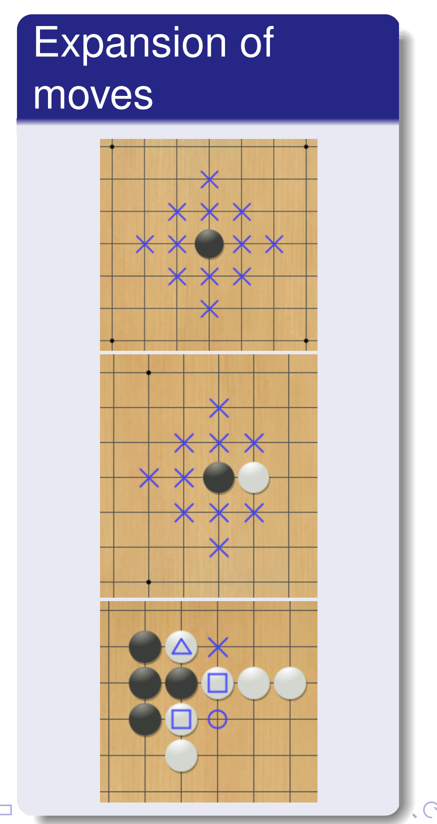
Local search

Local search

- Control the expansion of moves
- Efficient evaluator

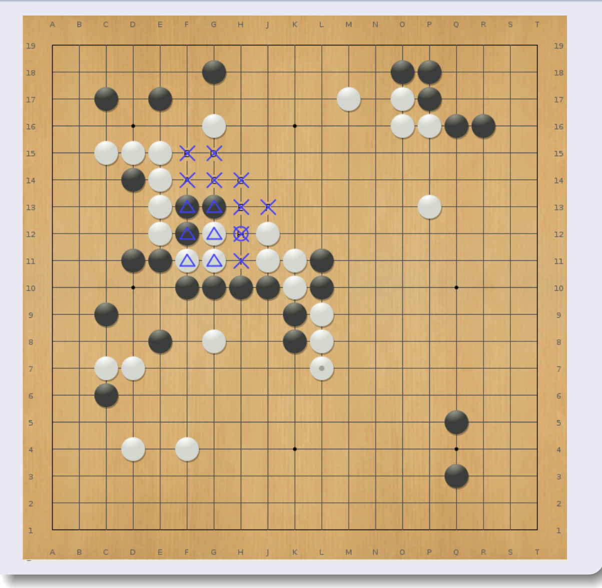


Expansion of moves

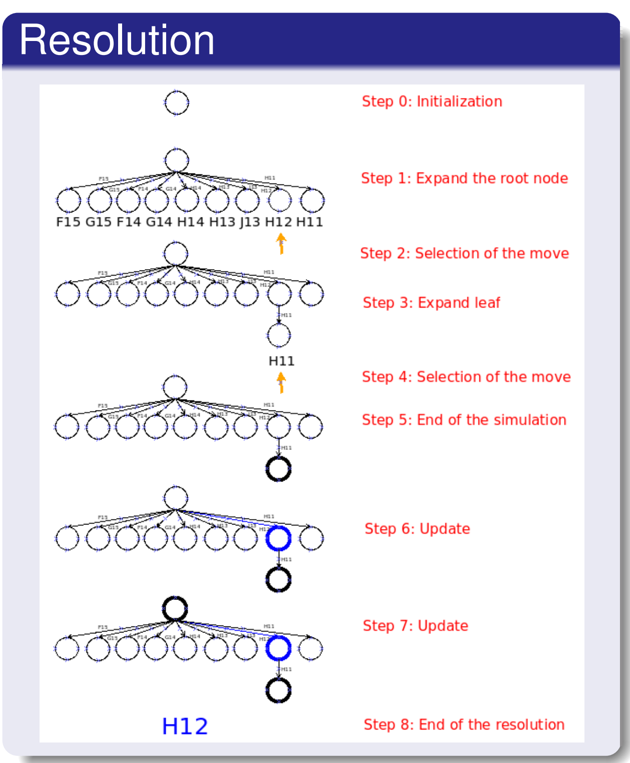


Monte Carlo simulation

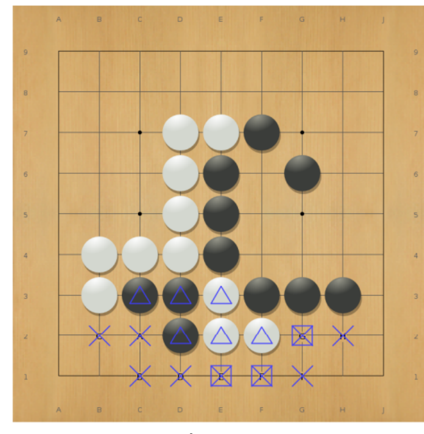
Situation: End game between 2 professional players



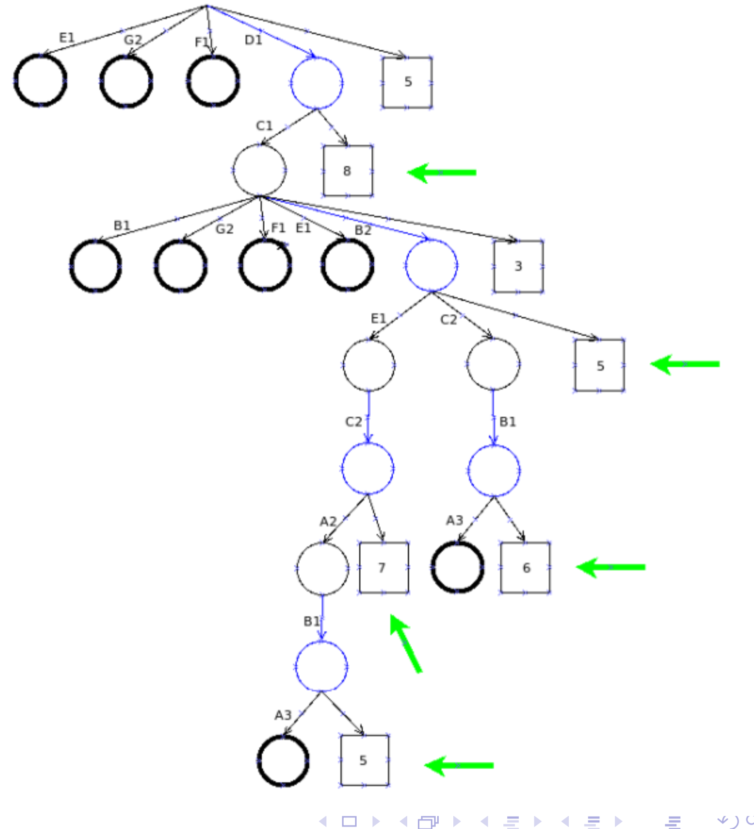
Resolution



After 9 iterations



C2 : 0 sims - 0%
C1 : 0 sims - 0%
B2 : 0 sims - 0%
D1 : 6 sims - 33.33%
E1 : 1 sims - 0%
F1 : 1 sims - 0%
G2 : 1 sims - 0%
H2 : 0 sims - 0%
G1 : 0 sims - 0%



MCTS*

A combination between MCTS and A*

- browse a part of the tree
  - for choosing a leaf (A*)
- launch a MC simulation from this leaf (MCTS)
- update the tree

Other approaches

Proof Number Search (Allis & al, 1994) and variants (e.g. Winands & al, 2002).

Inhibition

Principle: a temporary pruning

- A node is inhibited if its k last simulations have been lost.
- A node is reactivated if all brothers are inhibited or solved.
- When a node is reactivated, all brothers are also reactivated.
- When inhibited, the node and the resulting subtree are not visited anymore

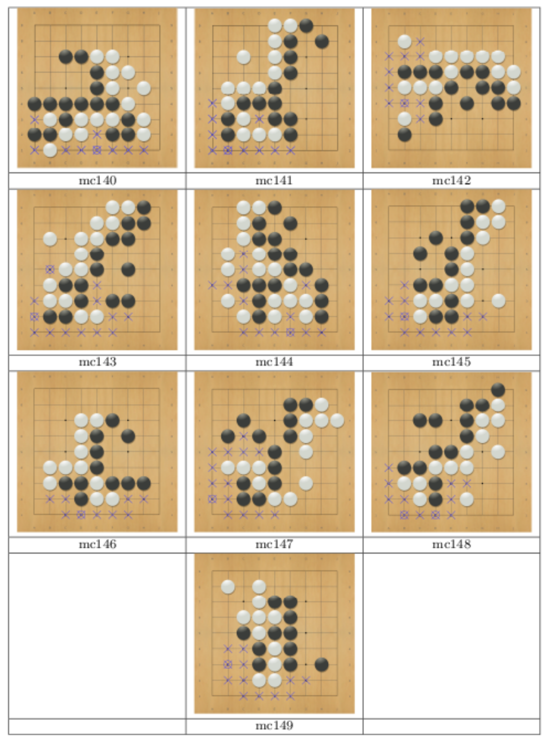
Advantages

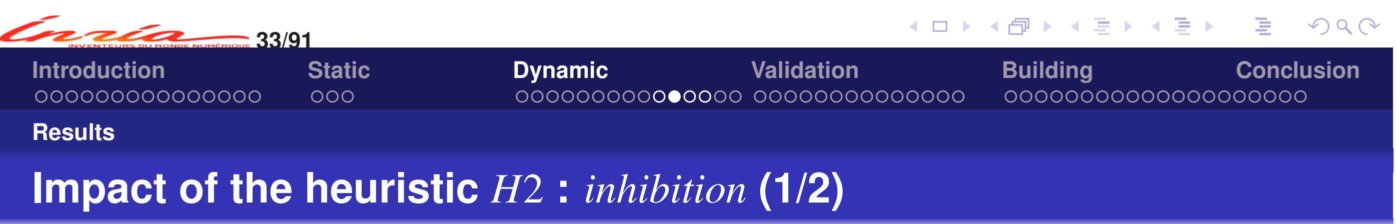
- Reduce considerably the part of the tree to browse.
- Avoid to spend time in recently refuted variants.

Controlling the branching factor (i.e. nb children at each node)

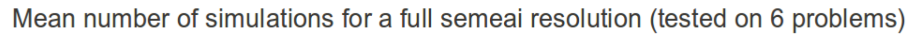
Progressive Widening (Coulom, 07), Progressive Unpruning (Chaslot, 07) ...

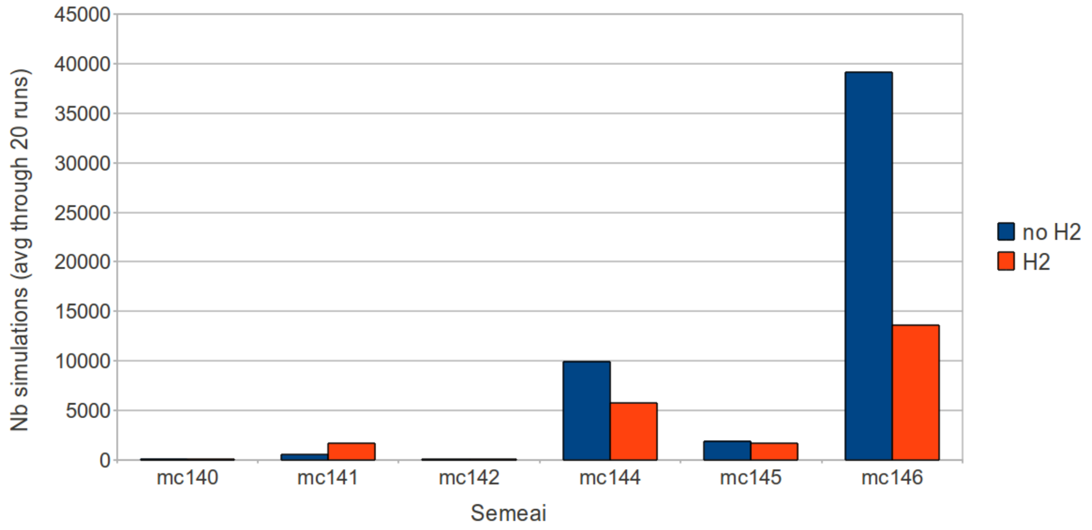
10 semeais from Yoji Ojima, the author of Zen. Black to play.





Impact of the heuristic $H2$: *inhibition* (1/2)

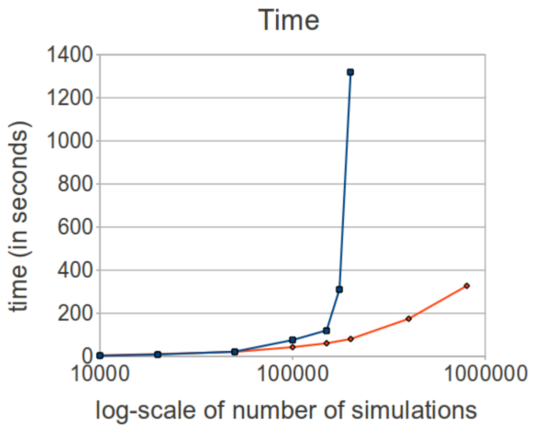


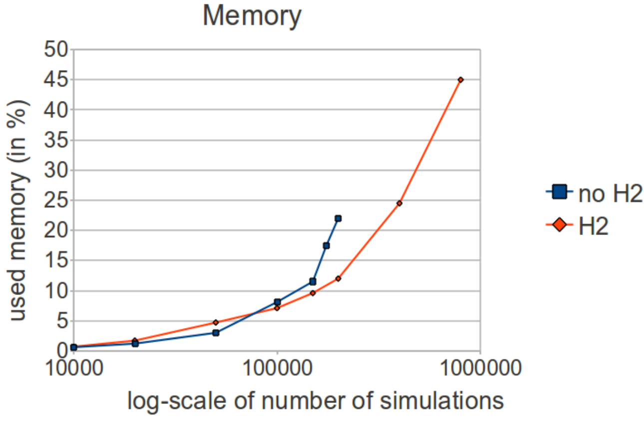


In difficult semeais,

- Fully solved more rapidly.

Impact of the heuristic $H2$: $inhibition$ (2/2)





With a same number of simulations

- Exponentially less spent time
- Less used memory

Comparison GoldenEye and MoGo

GoldenEye	10	20	50	100	10^3	10^4	10^5
mc140	0	0	20	20	20	20	20
mc141	0	0	0	5	20	20	20
mc142	0	20	20	20	20	20	20
mc143	0	0	7	6	8	18	20
mc144	0	0	0	19	19	20	20
mc145	0	13	11	20	20	20	20
mc146	0	0	20	20	20	20	20
mc147	0	0	0	0	5	10	20
mc148	0	0	0	0	20	20	9
mc149	0	0	0	1	8	10	8
Failed	200	167	122	89	40	22	23

MoGo	100	200	500	10^3	10^4	10^5
mc140	12	14	20	20	20	20
mc141	4	7	10	14	18	20
mc142	0	0	1	3	16	20
mc143	6	7	9	14	19	17
mc144	0	0	0	1	13	16
mc145	0	2	2	13	19	20
mc146	5	9	19	20	20	20
mc147	0	0	0	0	0	1
mc148	4	1	0	3	20	19
mc149	0	0	11	17	19	20
Failed	169	140	128	95	36	27

Comparison

- In terms of simulations, GoldenEye is better
- In terms of time, GoldenEye and MoGo are similar

How to introduce dynamic prior knowledge in MoGo? (Bourki & al, CG 2010)

Two solutions for using dynamic knowledge:

- by Expertise
- by Conditioning

Expertise

We introduce a bias in the score.

Conditioning

All simulations not consistent with the solver are discarded and replayed.

Play or not in the semeai ?

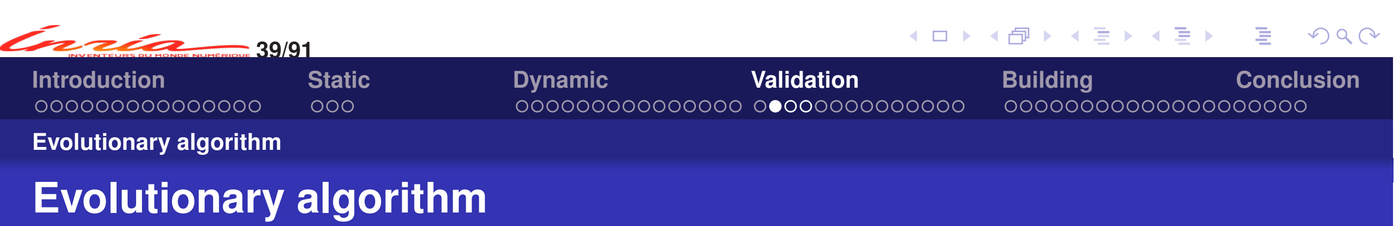
Results

Version of the algorithm	Percentage of "good" moves
Situation in which the semeai should be played 1K sims per move	
MoGo	32 %
MoGo with expertise	**79 %**
MoGo with conditioning	24 %
MoGo with exp.+condit.	**84 %**
Situation in which the semeai should not be played 1K sims per move / 30K sims per move	
MoGo	**100%** / 58 %
MoGo with expertise	**95 %** / 51 %
MoGo with conditioning	**93 %** / 0 %
MoGo with exp.+condit.	**93 %** / 54 %

- We get an improvement when the semeai should be played (MoGo+exp.+cond.)
- We get no improvement when the semeai should not be played

Outline

1. Introduction
2. Adding static prior knowledge (= patterns)
3. Adding dynamic prior knowledge (= local solver)
4. ***Validating prior knowledge***
   1. Evolutionary algorithm
   2. Application to MCTS
   3. Statistics
   4. RBGP Algorithms
   5. Experiments
5. Building prior knowledge (CluVo+GMCTS)
6. Conclusion



Evolutionary algorithm

Evolutionary algorithm

- automatic building of a program by trial (mutation) and error (test)
- solving a task
- 3 main troubles
  1. cost of the evaluation of a mutation
  2. size of the huge set of possible mutations
  3. stochastic fitness function

Noisy evolutionary algorithm

Noisy EA with binary fitness values

- I have a program P;
- I have a set S of possible mutations on P;
- One Monte-Carlo evaluation of $P + m$ for $m \in S$ provides:
  - Either I get a win versus P ($fitness = 1$);
  - or I get a loss ($fitness = 0$).
- We want:
  - to find a good mutation in S (expectation > 0.5, if any);
  - not to validate a bad mutation.

Evolutionary algorithm

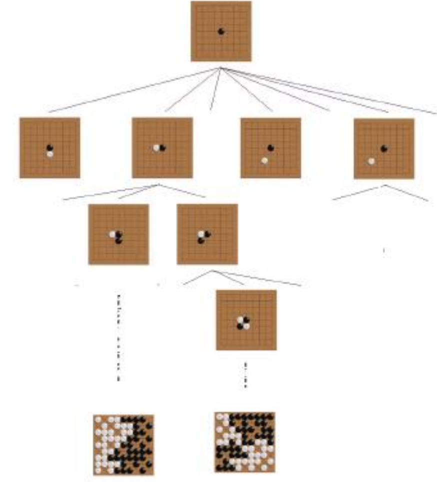
Evolutionary algorithm

- automatic building of a program
- 2 main issues
  1. load balancing
  2. statistical validation
- bandit-based approach
  $\Rightarrow$ load balancing
- racing-based approach $\Rightarrow$ both!

Application to MCTS

- 1st wins vs pros in the game of Go by our program MoGo
- better if good “bias”
- handcrafting this bias ?
  - boring
  - tedious
  - uneasy (error prone)
  - biased by human ideas

$\Rightarrow$ EA!



Goals

- automatize random generation of patterns
- automatize validation

Validation/rejection of multiple mutations

MSHT effect

- difficult
- so many trials $\Rightarrow$ cumulated risk:

  5% of risk per test + 100 tests = 99.4% of risk
  $(= 1 - (1 - 0.05)^{100})$.

- whenever we have 95% confidence, if 100 trials, with probability 99.4%, we accept a bad modification.
  $\Rightarrow$ Multiple Simultaneous Hypothesis Testing

Validation by Hoeffding bounds or Bernstein bounds (V. Mnih & al, 2008)

- Hoeffding's bound

$$deviation_{\text{Hoeffding}}(\delta, n) = \sqrt{\log(2/\delta)/(2n)}. \quad (1)$$

- Bernstein's bound

$$deviation_{\text{Bernstein}} = \hat{\sigma}\sqrt{2\log(3/\delta)/n} + 3\log(3/\delta)/n \quad (2)$$

Bernstein bounds versus Hoeffding bounds

- Bernstein better if small variance
- here variance roughly 1/4
- so, we keep Hoeffding

Racing algorithms

Stopping algorithms

- running tests until validation or rejection

Bandit algorithms

- distributing the computational effort among individuals

Racing algorithms = bandit + stopping

- distributing the computational effort *plus* validating

RBGP Algorithm (Hoock & O. Teytaud, EuroGP 2010 and EA 2011)

RBGP algorithm.
$S = S_0 =$ some set of mutations.
while $S \neq \emptyset$ **do**
  Select $s \in S$ // the selection rule is not specified here
  // (the result is independent of it)
  Let n be the number of simulations of mutation s.
  Simulate s n more times (*i.e.* now $nbSim(s) = 2n$).
  //this ensures $nbTest(s) = O(\log(nbSim(s)))$
  $computeBounds(s)$
  if $lb(s) > 0.501$ **then**
    Accept s; exit the program.
  else if $ub(s) < 0.504$ **then**
    $S = S \setminus \{s\}$ s is discarded.
  end if
end while

Compute Bounds

Function $computeBounds(s)$ (for countable S)
Static internal variable: $nbTest(s)$, initialized at 0.
Let n be the number of times s has been simulated.
Let r be the reward over those n simulations.
$nbTest(s) = nbTest(s) + 1$
Let i be such that $s_i = s$ and $\delta_i = 6\delta/(\pi^2 i^2)$.
Let $lb(s) = r/n - deviation_{\text{Hoeffding}}\left(\delta_i/(\left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.
Let $ub(s) = r/n + deviation_{\text{Hoeffding}}\left(\delta_i/(\left(\frac{\pi^2 nbTest(s)^2}{6}\right)), n\right)$.

Parameters

- $n =$ number of simulations of mutation *s*
- $nbTest(s) =$ number of times mutation *s* has been selected

Some maths

- Risk for t^{th} test of mutation i:

$$\delta_i / \left(\frac{\pi^2 t^2}{6}\right).$$

- $\sum_{t>0} \delta_i/(\pi^2 t^2/6) = \delta_i \Rightarrow$ the risk for mutation i (over all t) is $\leq \delta_i$;
- $\sum_{i>0} \delta_i = \delta \Rightarrow$ risk (over all i, t) $\leq \delta$.

$\Rightarrow$ MSHT ok!

Properties of RBGP (with proba $\geq 1 - \delta$)

- **Termination:** Halts after finite time.
- **Efficiency:** If $\exists$ mutation $\geq 0.504 \Rightarrow$ one pattern found.
- **Consistency:** No bad mutation accepted.

Protocol

Algorithm

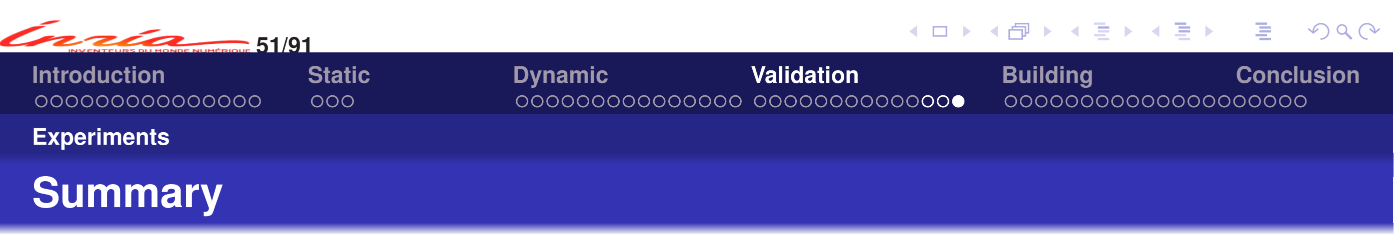
- Random pattern = mutation
- While(1) S=random set of patterns ; RBGP(S)

3 testbeds

- 9x9 Go (tested with the program MoGo)
- 19x19 Go
- 7x7 NoGo (a variant of Go)

Results

- good results in 9x9 Go (53.5% - 4 mutations)
- 19x19 (with the big database)
  - probably too strong coefficients
  - towards adaptive strength of mutation ?
- good results in 19x19 Go light, *i.e.* without the big database (61% - 6 mutations)
- very good results in Nogo (almost 70% - 43 mutations)
- Importantly: very difficult for programmers!



Summary

Properties of RBGP

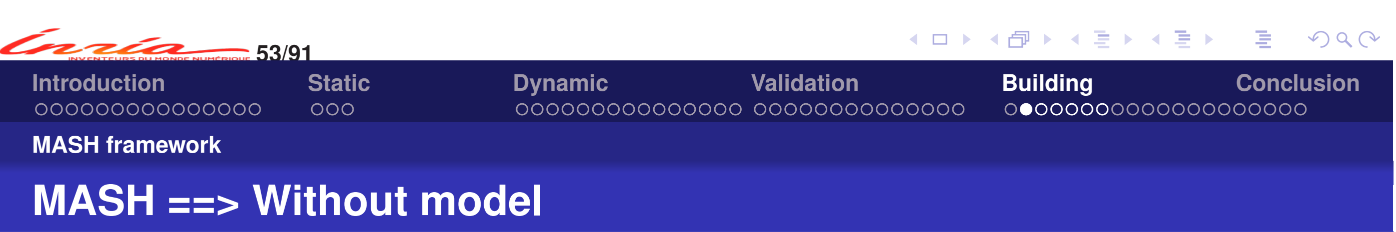
- terminates
- efficient
- consistent
- takes into account MSHT effect.

Runtime Analysis

- Let λ the size of the population S.
- Theoretically, the choice of λ should be $\log(\delta)/\log(1-f)$ (Hoock & Teytaud, EA 2011)
- Experimentally, results are nearly the same ==> $\lambda = 1$ is sufficient

Outline

1. Introduction
2. Adding static prior knowledge (= patterns)
3. Adding dynamic prior knowledge (= local solver)
4. Validating prior knowledge (RBGP)
5. ***Building prior knowledge (CluVo+GMCTS)***
   1. MASH framework
   2. Algorithm
6. Conclusion



MASH ==> Without model

MCTS

- Requires a model
- MCTS unusable
- Goal : MCTS usable in PO case without model?
- Build a model

MASH framework

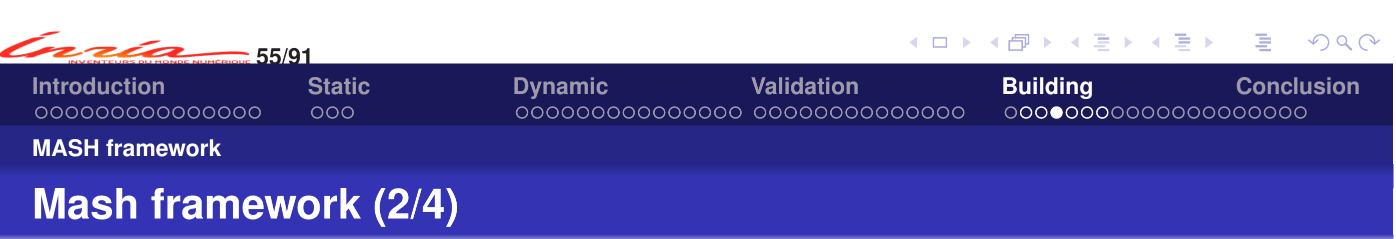
- no prior knowledge
- no model

Mash framework (1/4)

Environment

- Square room
  - Grey textures
  - Enclosed by 4 walls
- One avatar
- 2 flags
  - Blue
  - Red

View of the avatar



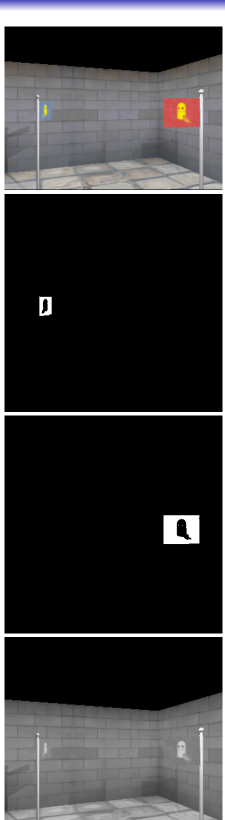
Mash framework (2/4)

Observation

- 3 heuristics
  - *Blue*
  - *Red*
  - *Identity*
- Each heuristic gives 100,000 features
  - One feature per pixel
- 300,000 features ==> 10,000 features
  - Random sampling
- More than $2^{10,000}$ observations

For the solver

- Sequence of numbers (e.g. 0 235 1 0 1 ...)
- No notion of heuristics



Mash framework (3/4)

Action

- 4 actions
  - Go forward
  - Go backward
  - Turn right
  - Turn left

For the solver

- Actions are 0, 1, 2, 3
- ==> Agnostic

Mash framework (4/4): one of the tasks (“blue then red”)

Rewards

- Hit a wall: -1
- Touch the first time the blue flag: +5
- Touch another time the blue flag: 0
- Touch the red flag:
  - After having touched the blue flag: +10 **==> Final state**
  - Without having touched the blue flag: -5
- By default: 0
- Optimal cumulative reward: **+15**

For the solver

- Moving without touching anything ==> 0
- Very rare reward information

Solving “blue then red“ : a challenge

Easy

- Very small action space
- No bad final state

Difficult

- Huge state space
- No prior knowledge
- Reward carries little information
- Small probability to reach quickly a good final state
  - The length of a correct sequence of actions could be greater than 100
- Partially observable (PO)
- Simulations are expensive
- No model

Motivation

”Philosophy“

- Optimize or improve a policy ==> Impossible:
  - Reward carries little information
  - No prior knowledge ==> direction toward a good final state is unknown
  - Small probability to reach quickly a good final state
  - Simulations are expensive
  - ==> Random is not an option
- First of all, reach a good final state
  1. Acquire know-how
     Be able to accomplish difficult subtasks
     - almost impossible randomly
     - maybe have nothing to do with the real task
  2. Combine know-how for accomplishing the task

Algorithm (Hoock & Bibai, TAAI 2012)

1. Building Macro-Actions (MAs)
   - categorize actions
   - build meaningful MAs using the categories
2. Clustering features
   - Simulate (with macro-actions)
   - Clustering of features (by merge of clusters)
3. GMCTS
   - subgoal = a cluster of features
   - policy with memory = tree of subgoals:
     + start at root
     + in a node, vote: use actions which tend to activate this node's features
   - this tree of subgoals is built using MCTS

1. ***Building Macro-Actions (MAs)***
   - categorize actions
   - build meaningful MAs using the categories
2. Clustering features
   - Simulate (with macro-actions)
   - Clustering of features (by merge of clusters)
3. GMCTS
   - subgoal = a cluster of features
   - policy with memory = tree of subgoals:
     + start at root
     + in a node, vote: use actions which tend to activate this node's features
   - this tree of subgoals is built using MCTS

Learning a categorization of actions

Learning

- Repetitive scenario
  - Study of impact

Categorization

- stationary (action applied many times ==> state becomes fixed)
  - e.g. go forward
- periodic (action applied many times ==> loop)
  - e.g. turn left
- inverse (action a + action b = action b + action a = no action)
  - e.g. turn left and turn right

Macro-actions

Remark

- Simulations very expensive
  - 15 actions / second
- Random search
  - Inefficient

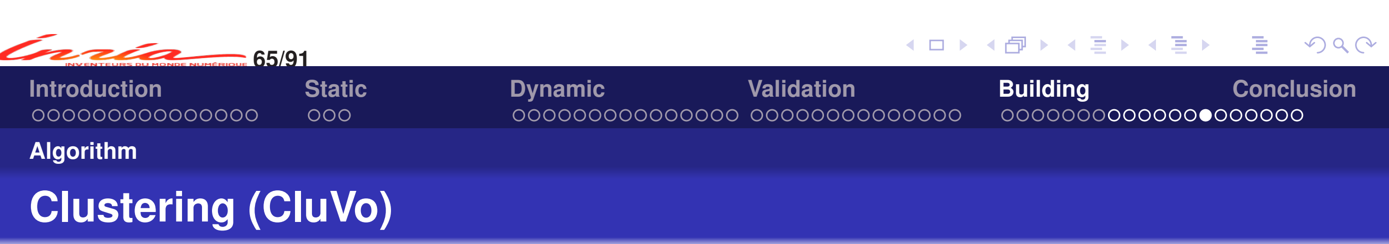
Proposed tool 1 : we need Macro-Actions (MAs)

- actions repeated several times
- ==> Better exploration of the environment
  - Macro Q-Learning (Sutton & al, 1997)

Proposed tool 2 : we use categories of actions for defining MAs

- More efficient random simulations
  - e.g. If last action == turn left then **no** turn right

1. Building Macro-Actions (MAs)
   - categorize actions
   - build meaningful MAs using the categories
2. ***Clustering features***
   - Simulate (with macro-actions)
   - Clustering of features (by merge of clusters)
3. GMCTS
   - subgoal = a cluster of features
   - policy with memory = tree of subgoals:
     + start at root
     + in a node, vote: use actions which tend to activate this node's features
   - this tree of subgoals is built using MCTS



Clustering (CluVo)

Algorithm: An agglomerative approach

- Initialization: one feature = one cluster
- Repeat
  - Generate a collection of lists of features
    - ==> Simulations
  - Merge some clusters
- Until the nb of clusters does not change anymore

Expected behaviour on ”blue then red“:

- Finding 2 clusters
  - Features provided by the *red* heuristic
  - Features provided by the *blue* heuristic

1. Building Macro-Actions (MAs)
   - categorize actions
   - build meaningful MAs using the categories
2. Clustering features
   - Simulate (with macro-actions)
   - Clustering of features (by merge of clusters)
3. ***GMCTS***
   - subgoal = a cluster of features
   - policy with memory = tree of subgoals:
     + start at root
     + in a node, vote: use actions which tend to activate this node's features
   - this tree of subgoals is built using MCTS

Policy with memory

Sometimes,

- Accomplish several small tasks (subgoals)
  - Divide and Conquer
  - Divide & Evolve (Bibai & al, 10), Qcut (Menache & al, 02)
- Touching the blue flag **then** touching the red flag
- (1) (2)
- policy[BlueCluster] policy[RedCluster]

Solution

- Memory = tree of subgoals

Memory

Subgoal

- Activate the largest possible number of features of a cluster
- e.g. MASH application, touch a flag
- ==> Voting scheme

Voting scheme

Learning statistically with simulations

Representation of the memory

- Memory represented by a tree
- Tree of subgoals
  - Node: a subgoal
  - Edge: decision of subgoal to accomplish

Learning a sequence of goals

GMCTS

Goal Monte Carlo Tree Search

Algorithm

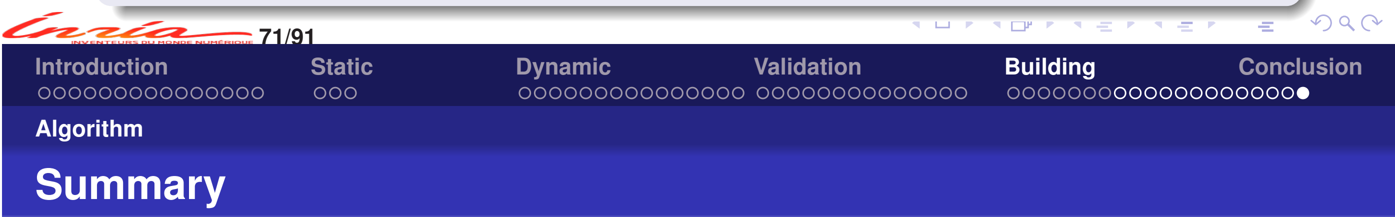
- Loop: 1..maxSimulations
  - Restart the application
  - Repeat
    - Choose the most promising subgoal or create a new subgoal
    - while subgoal not reached
      - Apply voting scheme
      - Keep rewards in memory
  - Until a final state is reached
  - Update tree (e.g. with kept rewards)
- Return the most simulated sequence of subgoals

Results of combination MA + CluVo + GMCTS

CluVo (2 hours per run including 10 minutes for MA)

- 12 restarts
- 3 successes

GMCTS (8 hours per run)



Run	Iteration	AvgCR	Success
1	1	-13.65	
	10	4.82	
	100	9.67	
2	1	8.87	
	10	10.62	
	100	10.2	
3	1	12.09 ± 1.06	81%
	10	12.64 ± 1.03	86%
	100	11.38 ± 1.21	77%

Computational time

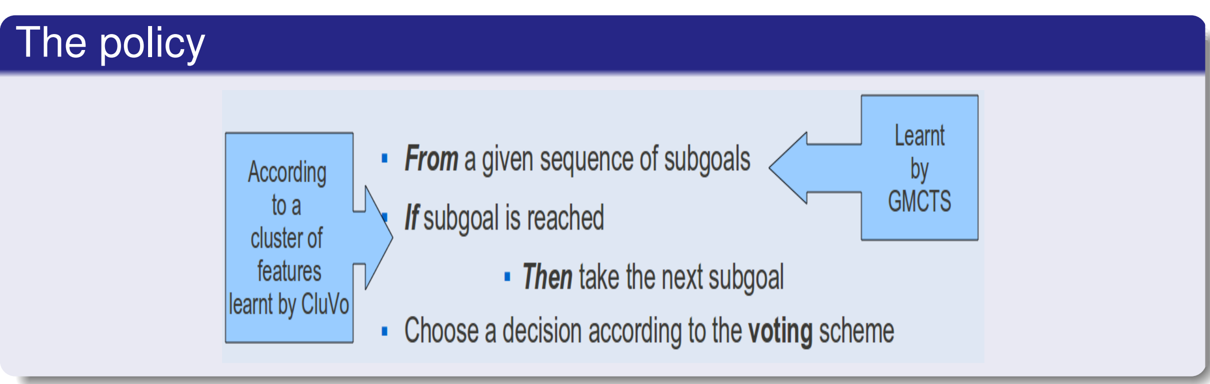
- 48 hours on a sequential machine
- 10 hours on a parallel machine

Summary

Efficient simulations

- Macro-actions for exploration
- Categorization of action for avoiding useless actions

The policy

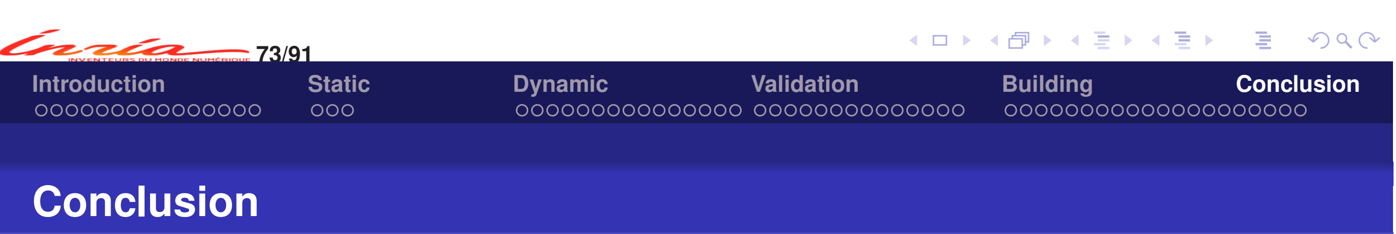


Generic?

- Applied successfully on 2 different applications
- e.g. Optimization of a letters draw

Outline

1. Introduction
2. Adding static prior knowledge (= patterns)
3. Adding dynamic prior knowledge (= local solver)
4. Validating prior knowledge (RBGP)
5. Building prior knowledge (CluVo+GMCTS)
6. ***Conclusion***



Conclusion

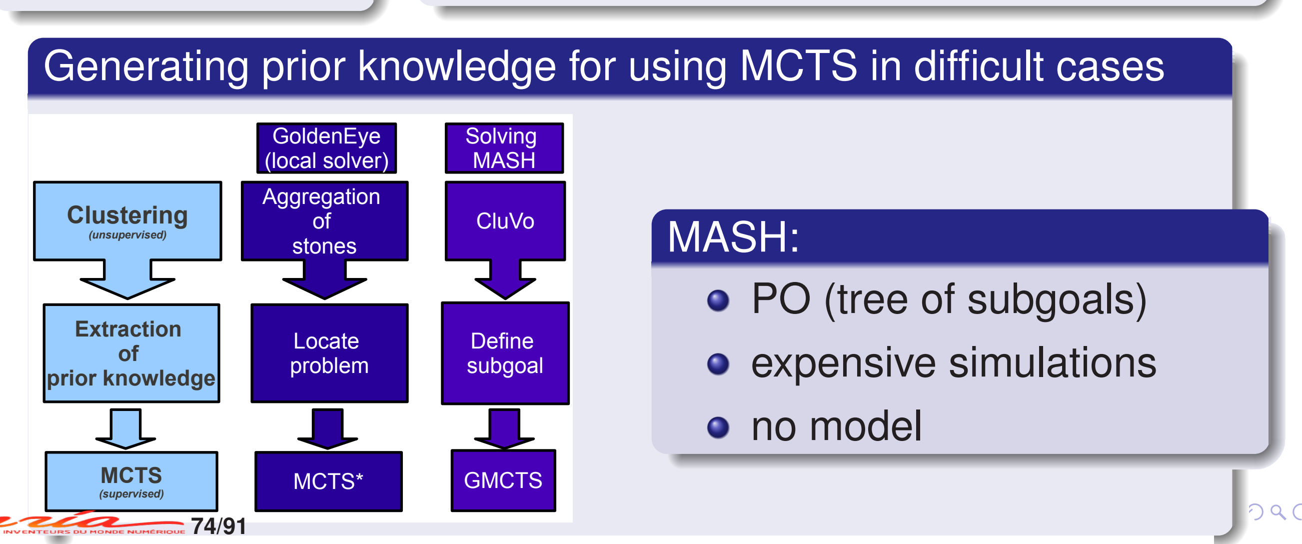
Prior knowledge

- static (patterns)
- dynamic (local solver)

Adding prior knowledge for improving MCTS

- By hand
- Automatically
  - Racing-based Genetic Programming
  - Rigorous validation

Generating prior knowledge for using MCTS in difficult cases



MASH:

- PO (tree of subgoals)
- expensive simulations
- no model

Perspective

GoldenEye

- Compare MCTS* with the state ot the art (e.g. PNS)
- Find new solutions for adding dynamic knowledge in MoGo
- Find other applications for MCTS*

GMCTS

- GMCTS with another definition of subgoal and another scheme (instead of voting)
- Check the genericity of the algorithm

RBGP

- Choose randomly or adapt the coefficient of the random pattern, instead of fixed values
- Non pure random patterns?
- "Improve" the bound formula
  - getting rid of the union bound
  - better parametrization of the δ_i (we want $\sum = \delta$, many possible cases)

Thank you!

References (1/2)

1. A. Bourki, G. Chaslot, M. Coulm, V. Danjean, H. Doghmen, T. Hérault, J.-B. Hoock, A. Rimmel, F. Teytaud, O. Teytaud, P. Vayssière, and Z. Yu. Scalability and Parallelization of Monte-Carlo Tree Search. In *The International Conference on Computers and Games 2010*, Kanazawa, Japon, 2010.
2. G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud. Adding expert knowledge and exploration in Monte-Carlo Tree Search. In *Advances in Computer Games*, Pamplona, Espagne, 2009. Springer.
3. S. Gelly, J. B. Hoock, A. Rimmel, O. Teytaud, and Y. Kalemkarian. The parallelization of monte-carlo planning. In *Proceedings of the International Conference on Informatics in Control, Automation and Robotics (ICINCO 2008)*, pages 198–203, 2008.
4. J.-B. Hoock and J. Bibai. Solving a Goal-planning task in the MASH project. In *The 2012 Conference on Technologies and Applications of Artificial Intelligence (TAAI 2012)*, Tainan, Taïwan, Province De Chine, 2012.
5. J.-B. Hoock and O. Teytaud. Bandit-Based Genetic Programming. In *13th European Conference on Genetic Programming*, Istanbul, Turkey, 2010. Springer.
6. J.-B. Hoock and O. Teytaud. Progress Rate in Noisy Genetic Programming for Choosing λ. In *Artificial Evolution*, Angers, France, 2011.

References (2/2)

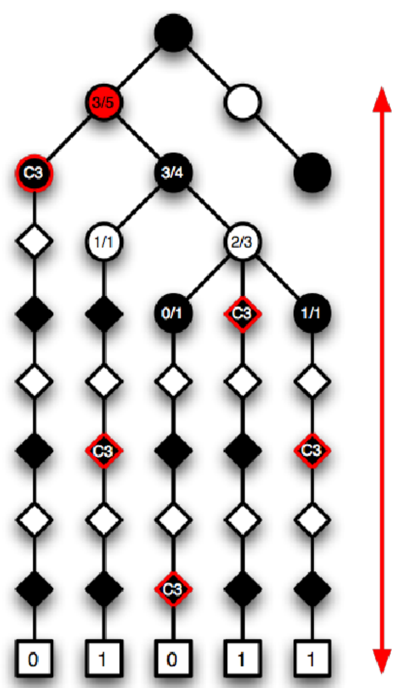
1. L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. *Artif. Intell.*, 66(1):91–124, 1994.
2. J. Bibai, P. Savéant, M. Schoenauer, and V. Vincent. An Evolutionary Metaheuristic for Domain-Independent Satisficing Planning. In R. Brafman, H. Geffner, J. Hoffmann, and H. Kautz, editors, *20th International Conference on Automated Planning and Scheduling-ICAPS2010*, pages 15–25, Toronto, Canada, May 2010. AAAI Press.
3. T. Cazenave and N. Jouandeau. On the parallelization of UCT. In *Proceedings of CGW07*, pages 93–101, 2007.
4. G. Chaslot, M. Winands, J. Uiterwijk, H. van den Herik, and B. Bouzy. Progressive strategies for monte-carlo tree search. In P. Wang et al., editors, *Proceedings of the 10th Joint Conference on Information Sciences (JCIS 2007)*, pages 655–661. World Scientific Publishing Co. Pte. Ltd., 2007.
5. R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. *In P. Ciancarini and H. J. van den Herik, editors, Proceedings of the 5th International Conference on Computers and Games, Turin, Italy*, 2006.
6. R. Coulom. Computing elo ratings of move patterns in the game of go. In *Computer Games Workshop, Amsterdam, The Netherlands*, 2007.
7. R. Coulom. Criticality: a monte-carlo heuristic for go programs, 2009. Invited talk at the University of Electro-Communications, Tokyo, Japan.
8. S. Gelly. Thesis : Une contribution a l'apprentissage par renforcement ; application au computer go. 2007.
9. A. Mcgovern, R. S. Sutton, and A. H. Fagg. Roles of macro-actions in accelerating reinforcement learning. In *In Grace Hopper Celebration of Women in Computing*, pages 13–18, 1997.
10. I. Menache, S. Mannor, and N. Shimkin. Q-cut - dynamic discovery of sub-goals in reinforcement learning. In *Proceedings of the 13th European Conference on Machine Learning*, ECML '02, pages 295–306, London, UK, UK, 2002. Springer-Verlag.
11. V. Mnih, C. Szepesvári, and J.-Y. Audibert. Empirical Bernstein stopping. In *ICML '08: Proceedings of the 25th international conference on Machine learning*, pages 672–679, New York, NY, USA, 2008. ACM.

Appendix

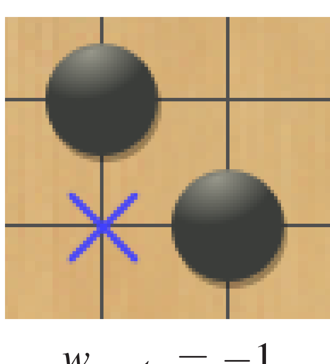
AMAF (Gelly & Silver, ICML 2007)

AMAF (=RAVE)

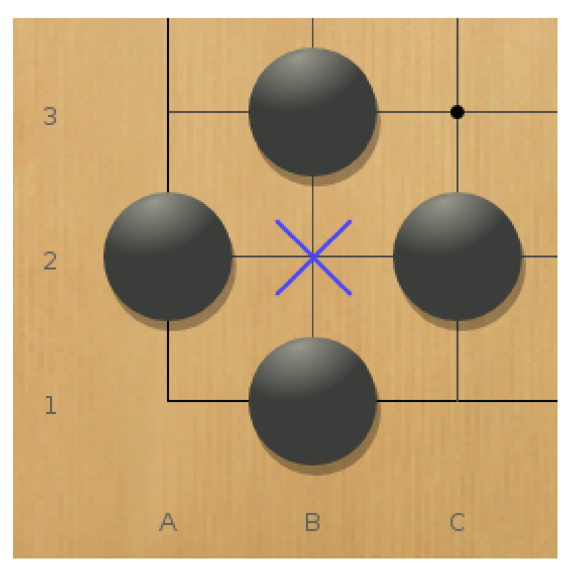
- Keep for each node n and each move d
  the number of wins $amafWins$ and losses $amafLosses$ where d has been played after n.
- Compute a score
  $\widehat{p}(d) = \frac{amafWins(d)}{amafWins(d)+amafLosses(d)}$
  empirical score when d has been played after n



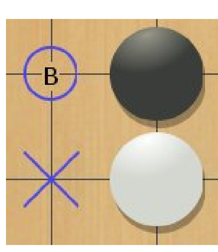
Static : Pattern matching



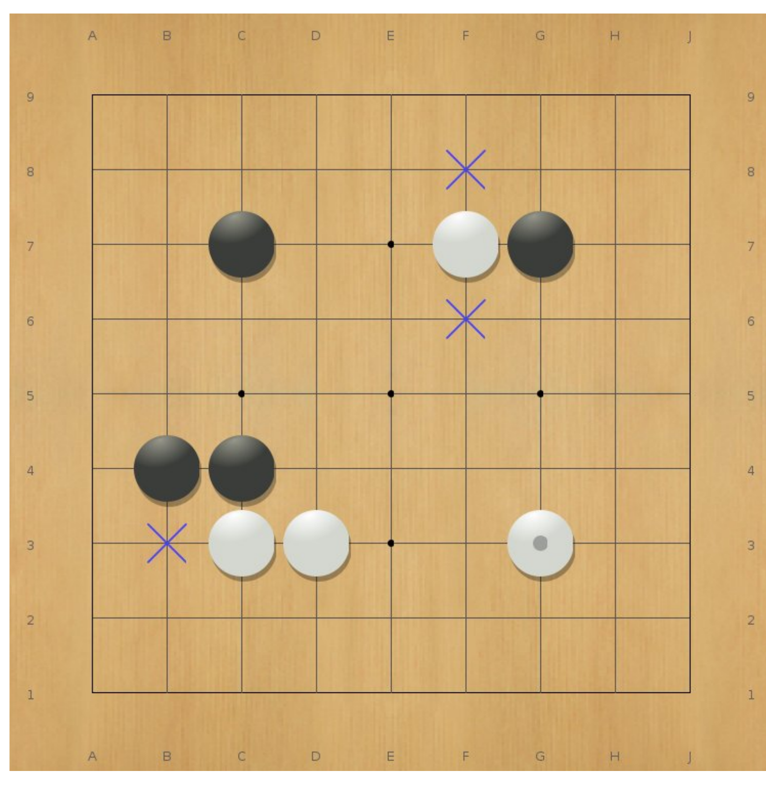
$w_{empty} = -1$



RBGP : Random pattern = Mutation



0.75



RBGP : The population size

Let λ the size of the population S.

Runtime Analysis

How to choose the parameter λ ?

- Bigger λ is, ...
- ... Better the quality is ...
- ... But more rapidly the cost increase

Theoretically,

$$\lambda = \log(\delta)/\log(1-f) \quad (3)$$

- with f the frequency of good mutations
- and δ the risk level chosen by the user
- proof in Hoock & Teytaud, EA 2011

RBGP : Results

9x9 Go

- Far less human expertise
  - positive patterns and negative patterns
- Positive patterns

Tested code	Opponent	Success rate
MoGoCVS + P1	MoGoCVS	$50.78\% \pm 0.10\%$
MoGoCVS + P1 + P2	MoGoCVS +P1	$51.2\% \pm 0.20\%$
MoGoCVS + P1 + P2	MoGoCVS	$51.9\% \pm 0.16\%$

- Negative patterns

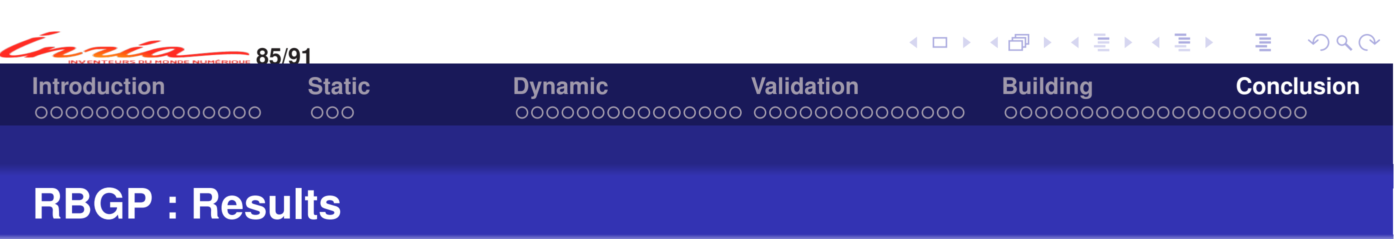
Tested code	Opponent	Success rate
MoGoCVS + P1 + P2 + P3	MoGoCVS + P1 + P2	$50.9\% \pm 0.2\%$
MoGoCVS + P1 + P2 + P3	MoGoCVS	$52.6\% \pm 0.16\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS + P1 + P2 + P3	$50.6\% \pm 0.13\%$
MoGoCVS + P1 + P2 + P3 + P4	MoGoCVS	$53.5\% \pm 0.16\%$

- **53.5%** +/-0.16 against the baseline

RBGP : Results

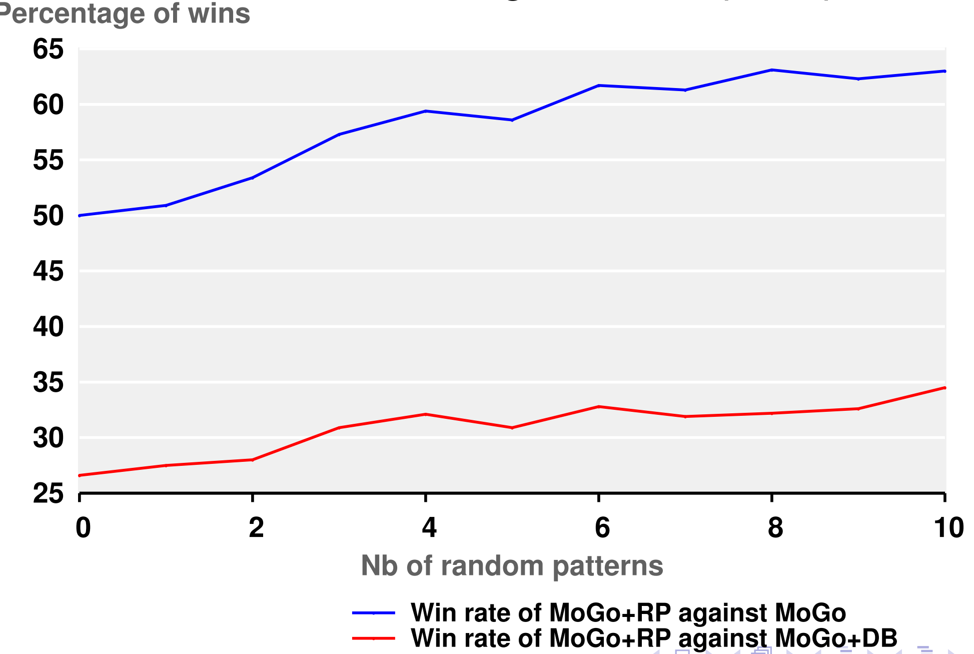
19x19 Go

- a lot of human expertise already exists in the full MoGo
  - no mutation validated by RBGP
  - not so bad, humans often validate bad mutations :-)
  - maybe better with smaller mutation strength ?
- 19x19 with no database - light MoGo
  - better for saving up memory
  - faster
  - easier for RBGP



Introduction Static Dynamic Validation Building **Conclusion**

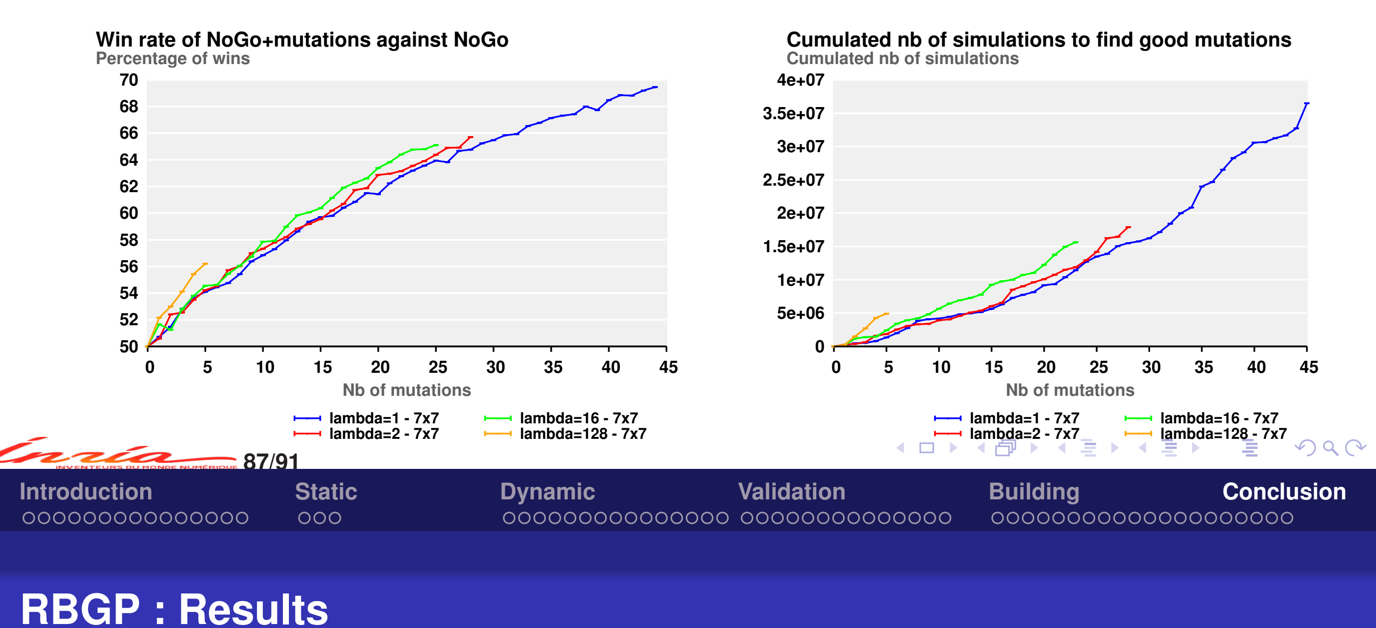
RBGP : Results



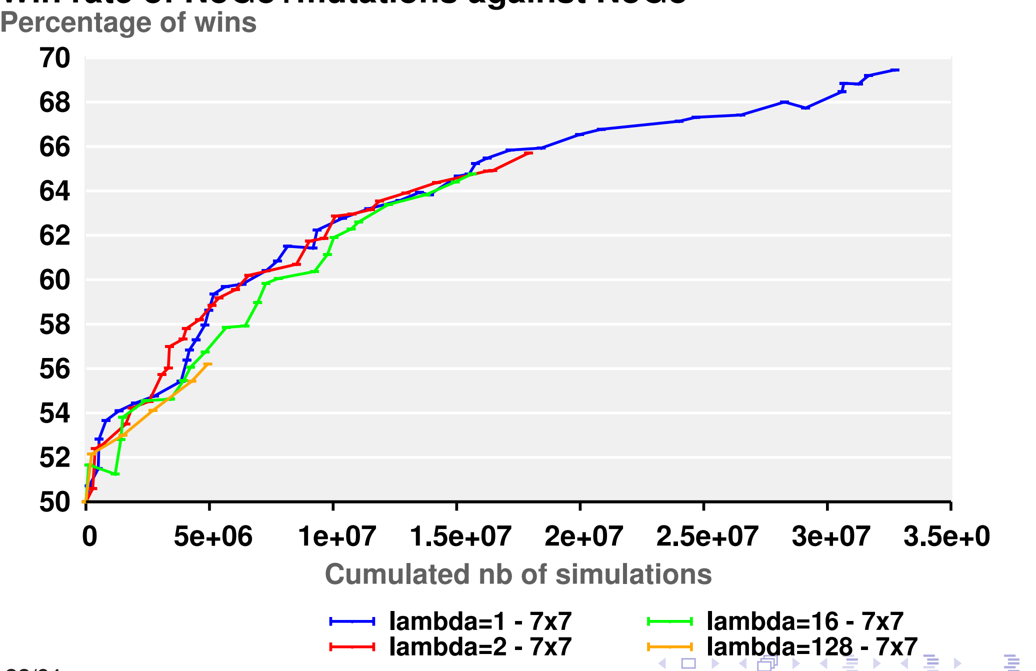
RBGP : Results

testbed NoGo

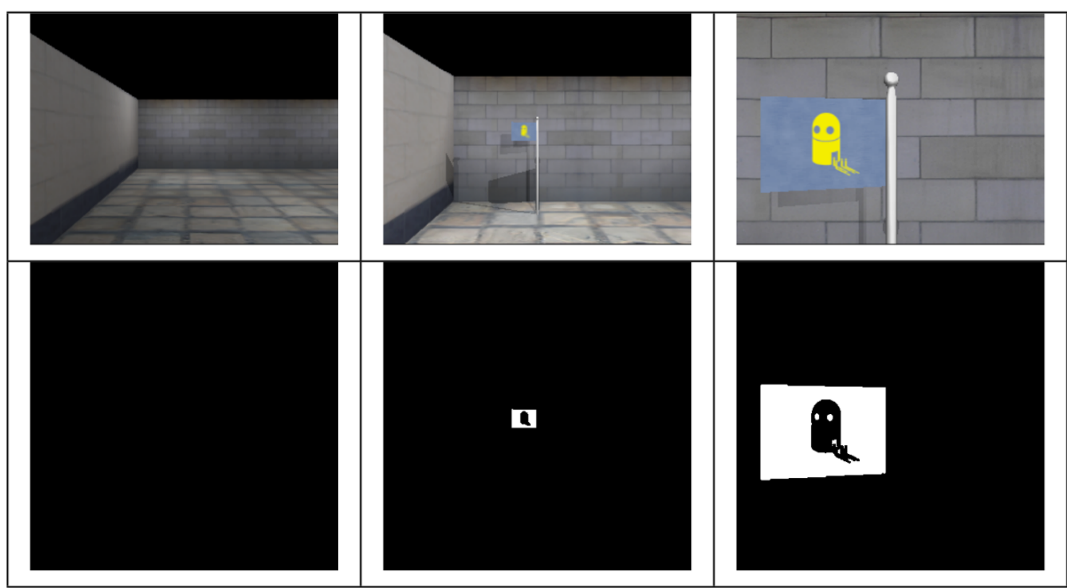
- a variant of game of Go
- a nice challenge for game developpers according to the Birs seminar on games
- no human expertise



RBGP : Results



CluVo+GMCTS : Activate/deactivate features



CluVo+GMCTS - Voting scheme (1/2) : A simple policy

Voting scheme

- Goal ==> Activate the largest possible number of features of a cluster.
- For a given cluster,

Vote

- some features are activated
- use these features for choosing an action

Exploration

- No feature is activated
- choose an action for activating features

e.g. applied on MASH

- Goal ==> Touch a flag.
- For a given flag,

- the flag is seen
- move toward the flag

- No flag is not seen
- find as soon as possible the flag

CluVo+GMCTS - Voting scheme (1/2) : Main components of the policy

Vote

- Each activated feature votes for an action
- ==> Increase the number of activated features

Exploration

- Some actions activate more frequently features
- ==> Activate as soon as possible some features