University Paris-Dauphine
Lamsade

Thesis submitted to the University of Paris-Dauphine
for the degree of Master of Science
in Computer Science

Thesis advisor : Prof. Tristan CAZENAVE

Some Improvements for Monte-Carlo Tree Search, Game Description Language Compilation, Score Bounds and Transpositions

Abdallah SAFFIDINE
<abdallah.saffidine@gmail.com>

Paris, Septembre 2010

Abstract

Game Automatons (GAs) are a model of sequential finite multiplayer games. Monte-Carlo Tree Search (MCTS) is a recent framework for building an Artificial Intelligence (AI) for board game playing requiring potentially no domain specific knowledge. Our work revolves around the application of MCTS to GAs. This thesis contributes three different main parts. We implement a forward chaining compiler for the General Game Playing (GGP) problem; the input is translated from the declarative Game Description Language (GDL) to a GA that can be interfaced with a playing program. We enhance MCTS with an algorithm to keep track of admissible bounds that allows to solve certain positions and improves the playing strength in general. We study how transpositions can be used in MCTS, in particular, we propose a parametric adaptation of the Upper Confidence bound for Trees (UCT) algorithm to the Direct Acyclic Graph (DAG) case.

Résumé

Les automates de jeux (GA) constituent un modèle pour les jeux multijoueurs séquentiels. La recherche arborescente Monte-Carlo (MCTS) est une technique récente permettant de construire des intelligences artificielles (AI) pour des jeux, potentiellement sans faire appel à des connaissances spécifiques au domaine. Ce travail s'intéresse à l'application de MCTS au GA. Il apporte trois contributions distinctes. Nous développons un compilateur pour le problème GGP à base de chaînage avant ; les règles d'un jeu donné sont traduites depuis le langage déclaratif GDL vers un GA qui peut être interfacé avec un programme de jeu. Nous augmentons MCTS d'un algorithme permettant de tenir compte de bornes d'admissibilité ; il permet de résoudre certaines positions et améliore globalement le niveau de jeu. Nous étudions comment tenir compte des transpositions dans MCTS, nous proposons en particulier une adaptation paramétrique de l'algorithm UCT au cas des graphes orientés acycliques DAG.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

AI Artificial Intelligence

AMAF All Moves as First

AST Abstract Syntax Tree

CGT Combinatorial Game Theory

DAG Direct Acyclic Graph

DNF Disjunctive Normal Form

EFG Extensive-form Game

GA Game Automaton

GDL Game Description Language

GGP General Game Playing

IIL Inverted Intermediate Language

KIF Knowledge Interchange Format

LOA Lines of Action

MCTS Monte-Carlo Tree Search

RAVE Rapid Action Value Estimation

UCT Upper Confidence bound for Trees

Acknowledgements

I would like to thank my thesis advisor Tristan Cazenave without whom this work would not have been possible. He provided me with unlimited support, help and guidance.

I am also very grateful to all those who shared insightful thoughts about Games, Machine Learning, Compiling or Game Theory, including but not limited to Jean Méhat, Yann Chevaleyre, Bruno De Fraine and Jérôme Lang.

The financial support of the École Normale Supérieure de Lyon is gratefully acknowledged.

1 Introduction

1.1 Motivation

Game playing is often depicted [Sch01] as a good testbed for AI techniques. The task of building an intelligent player should be a lot easier than building an intelligent general agent. The world of a game is indeed much simpler than the physical world : for instance the goal and the dynamics and the possible interactions of a game are well defined and are known to the players which is not always the case in the real world. Still, interesting games are usually complex. An intelligent player is expected to take a decision without an exhaustive search of the possible outcomes, not to repeat the same mistakes again and again, to be able to play well different games. Moreover, some games might involve chance (backgammon), hidden information (phantom go) or both (most card games). Thus, building an intelligent player is not trivial and has motivated decades of active research over the globe [Sch01].

Researchers believed in the 50s that if a computer could beat the world Chess champion then general AI would be achieved. Sixty years later, the Machine plays consistently better than Humans on several games (chess, backgammon, scrabble[Hsu02, She02]), not so easy games have been solved (Four-in-a-row, Gomoku and Checkers among others [All94]). Some techniques developed in the game playing community spanned to other domains [MRVP09], but general AI is still out of sight. Worse, no good general player has been developed yet.

Chess playing programs[1] are based upon a lot of handcrafted Chess knowledge like an opening book, an endgame database as well as an evaluation function fitted to chess features like being a pawn up, controlling the center etc. Therefore Chess programs have not a clue about, say, Checkers.

To try to address this deficiency, the GGP competition was created in 2005 [GL05]. The competitors are asked to play many different games that are new to them. To put it more precisely, at the beginning of a match, each player receives the rules of the game to be played in a formal language, as well as the role to impersonate. Hence, it is challenging for the programmer to put in any

[1] Chess programs will be used as a running example

game specific knowledge, for the precise game is not known before hand.

MCTS is a new alternative to the combination of the minimax algorithm with an evaluation function. Since it is based on random simulations, it offers the possibility to build a playing program with almost no domains specific knowledge. It now constitutes the state of the art of playing programs in many games such as Go [Cou06, GS08], GGP [FB08] or Hex [CS09]. MCTS algorithms have also been very successfully applied to games with incomplete information such as Phantom Go [Caz06], or to puzzles [Caz07, SWvdH+08].

MCTS has also been used with an evaluation function instead of random playouts, in games such as Amazons [Lor08] and Lines of Action (LOA) [WB09].

1.2 Topics addressed in this thesis

Game Model

We develop a formal model for games in which a game is represented by a so-called GA. This model was already presented in [GL05]. It is quite general, as it can be used to represent a variety of kind of games such as multiplayer games, puzzles, zero-sum games, non-zero sum games, simultaneous and sequential games. Yet, this model can be much more compact than Extensive-form Game (EFG) for instance.

Although several game models already exists. The proposed representation strives to help defining general algorithms in a formal way. It is hoped that eventually game specific algorithms can be expressed on a GA through the use of hypotheses on the GA. Given an algorithm working on a specific game, expressing this algorithm on a restricted class of GAs may enable the algorithm to be used on slightly different games. Trying to identify the hypotheses on a GA for the algorithm to work would shed light on both the algorithm and the first game it was applied to.

To test the different algorithms developed in the course of this work, we devised a compiler transforming a game written in the GDL [LHG06] to a GA that could be interfaced with our algorithm as depicted in figure 1.1 (Figure 3.1 gives more details). Our algorithms could thus be tested on the many different games that were presented in the previous GGP competitions.

Monte-Carlo Tree Search

MCTS Solver was presented in [WBS08] to prove that some position is lost or won. We extend the MCTS algorithm to take score bounds into account. Score bounds are admissibility bounds on the outcome that can be reached in a given node. These bounds are conservative and enable the MCTS to prove the value of some positions.

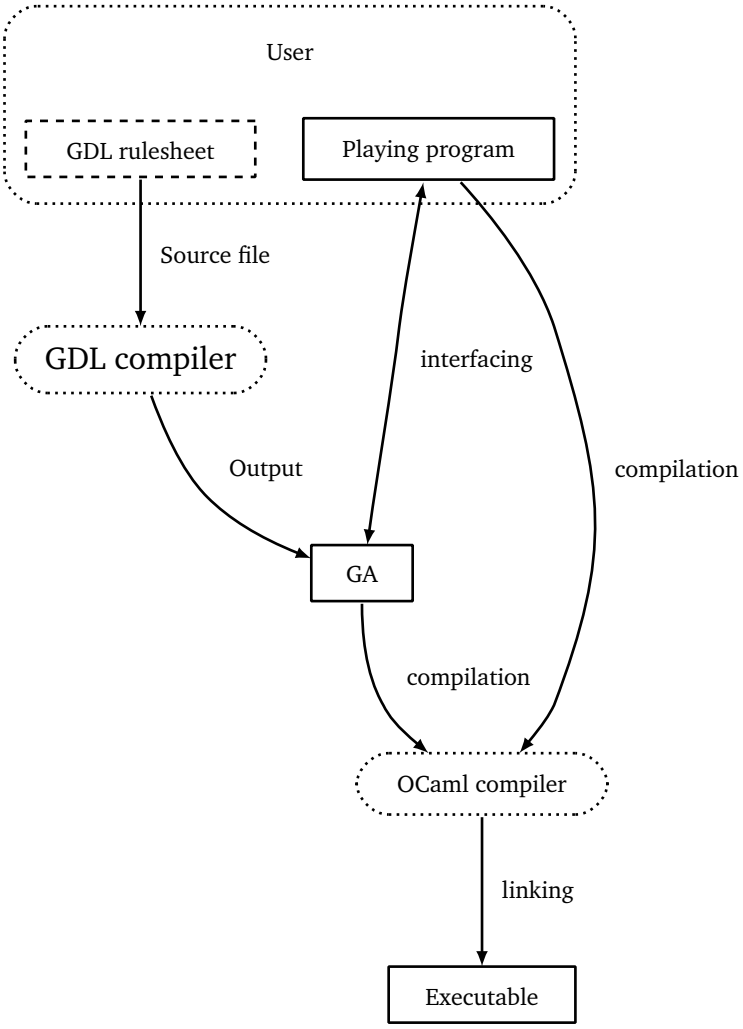


Figure 1.1: Interactions between User and the GDL compiler.

Transpositions occur when the same position can be reached through different move sequences. Taking transpositions into account has drastically improved the playing strength of minimax based AIs but it is not clear how they should be used in MCTS. Introducing transpositions in the MCTS framework has been first described in [CBK08]. We provide a parametric algorithm that generalize previous works on transpositions and MCTS and improves playing strength on the realized tests.

1.3 Reading guide

Section 2 contains general background information about the model used 2.1 as well as the MCTS algorithm 2.3. A reader familiar with game models or

not interested in formal definitions can safely skip section 2.1 and section 2.2. The basics of the MCTS are recalled in section 2.3. The sections 3, 4 and 5 are independent one from another and can be read in any order. The section 3 is self-contained and requires no MCTS knowledge. Little game knowledge is needed for this section but a familiarity with the GGP problem can naturally give further insight. Section 4 and section 5 can also directly be read by someone familiar with the MCTS framework.

2 Premilinaries

Contents

2.1 Game Automaton

Definition

We first present a general and abstract way of defining games to encompass puzzles and multiplayer game, be it turn-taking or simultaneous. Informally, a Game Automaton is a kind of automaton with an initial state and at least one final state. The outgoing transitions in a state are the possible moves in this state. The set of outgoing transitions in a state is the cross-product of the legal moves of each player in the state (non turn players play a *no-operation* move). Final states are labelled with an outcome and players define an preference relation on the possible outcomes.

In the following, for a function f taking several arguments and one argument a, we denote by f_a the partial application of f to a.

Formally, let $P = \{p_1, \ldots, p_k\} \neq \emptyset$ be a non empty finite set of players or agents, let $\mathbb{O}$ be a set of outcomes, for each player p, $\leq_p$ is a total preorder on $\mathbb{O}$. That is, for each possible two outcomes o_1 and o_2 and for each player p, either p prefers o_1 over o_2 or p prefers o_1 over o_2 or p is indifferent between both outcomes. Let Σ be a set of states, let $i \in \Sigma$ be the initial state and $F \subset \Sigma$ be the final states, $F \neq \emptyset$. Each final state $f \in F$ is labelled with a unique outcome $o(f) \in \mathbb{O}$. For each player p, we define the possible moves of p as M_p. For each state s and each player p, we define the legal moves of p in s as $L(s, p) \subset M_p$. The transition function t maps the conjunction of a state and legal moves from all players to another state : $\forall s, t_s : L_s(p_1) \times \ldots \times L_s(p_k) \to \Sigma$. We further ensure the following restrictions. If a state is final then no player has any legal move : $\forall f \in F, \forall p \in P, L(f, p) = \emptyset$, otherwise every player has at least one legal move : $\forall s \notin F, \forall p \in P, L(s, p) \neq \emptyset$. We call Game Automaton the tuple $(\Sigma, i, F, P, M, t, \mathbb{O}, o)$.

We call *underlying graph* of such a game the directed graph $\{\Sigma, T\}$, such that there is an edge between two states if and only if it is possible to go from one state to the other with the transition function : $\forall s_1, s_2 \in \Sigma, (s_1, s_2) \in T \Leftrightarrow \exists m_1 \in M_{p_1}, \ldots, m_k \in M_{p_k}, t_{s_1}(m_1, \ldots, m_k) = s_2$. For each states s_1, s_2, we say that s_2 is a successor of s_1 and s_1 is a predecessor of s_2 if $(s_1, s_2) \in E$. We say that s' is reachable from s if there is a sequence of states $s_0, s_1, \ldots, s_n$ such that $s_0 = s$, $s_n = s'$ and s_{i+1} is a successor of s_i for each $i \in [0, n-1]$. We may also use the word child (resp. parent) for successor (resp. predecessor). If s' is reachable from s, we may say that s is an ancestor of s'.

From now on, we will only consider games with a finite acyclic underlying graph.

It can be useful to consider the unfolding game $\Gamma^\dagger$ of a game Γ. $\Gamma^\dagger$ is defined such that its underlying graph is the unfolding of the underlying graph of Γ. The players, the moves and outcomes are not changed.

A *turn taking* game is a game in which for every state, at most one player has more than one legal move. The player having more than one move is said to be *in turn*. We call *first player* the player in turn in the initial state i. We also use *sequential* as a synonym for turn taking.

A *puzzle* is a game with one player : $\#P = 1$. With those definitions, puzzles are turn taking games with only one player.

A *utility game* is a game in which the outcomes can be expressed as real vectors with one dimension for each player : $\mathbb{O} \subset \mathbb{R}^k$. For an outcome $o = (o_{p_1}, \ldots, o_{p_k}) \in \mathbb{O}$ and a player p_j, we write $o(p_j)$ for the j^{th} component of o that corresponds to p_j. $o(p_j) = o_{p_j}$. A *constant sum* game is a utility game in which the sum of the components of each vector of O is constant, that is, there exist a real number ω such that for all outcome $o \in O$, $o(p_1) + \ldots + o(p_k) = \omega$. A *zero sum* game is a constant sum game in which the constant is equal to zero $\omega = 0$.

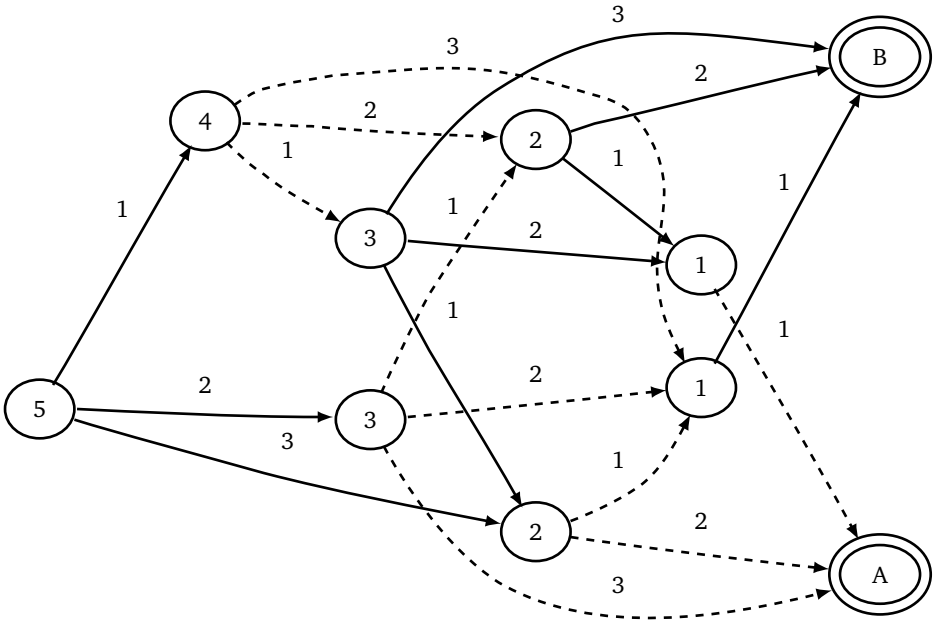


Figure 2.1: Nim Game Automaton

Reduction

Principle

As the Nim GAs 2.1 and 2.2 show, there can be many GA representing the same game. It is possible to specify formally the relation between GAs representing the same game. A GA is constituted of a labelled state transition system. It is therefor natural to extend the concept of *bisimilarity* from labelled state transition systems to GAs.

Let $\Gamma = (\Sigma, i, F, P, M, t, \mathbb{O}, o)$ and $\Gamma' = (\Sigma', i', F', P, M, t', \mathbb{O}', o')$ be two GA. We say that Γ and Γ' are *bisimilar* if the corresponding labelled state transition systems are bisimilar : $(\Sigma, M, t) \sim (\Sigma', M, t')$, the initial states are equivalent $i \sim i'$ and equivalent final states have the same outcomes $\forall p, p', p \sim p' \implies o(p) = o'(p')$. Just as with labelled state transition systems, bisimilarity for GAs is an equivalence relation.

For instance, if Γ is a game and Γ' is its unfolding, then Γ and Γ' are bisimilar. This means that any game theoretic result obtained on Γ' can be carried over to Γ without trouble. This assumption is at the base of tree searches algorithms.

Using the model

When a designing an AI for a given game, using domain specific knowledge is usually a condition to obtain decent results. The way this knowledge is used is

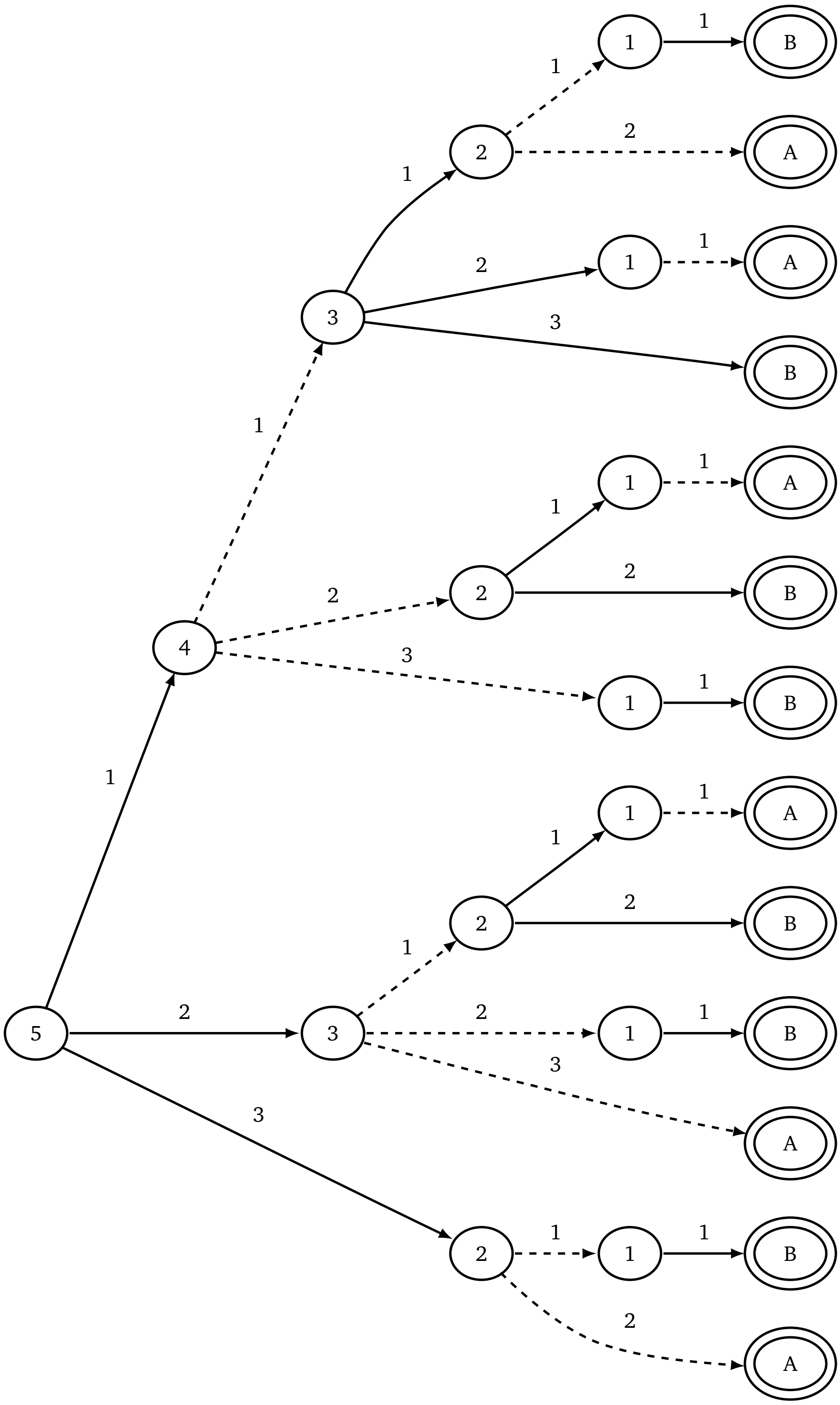


Figure 2.2: Unfolding for Nim

however often transferable to other games. Formalizing the domain specific knowledge required in the proposed model make recognizing on which specific property of the game enable the usage of this knowledge. This can in turn help finding similar domain knowledge in games fulfilling the same hypothesis.

Another goal of this model is to give a framework for proving properties of general algorithms. By definition a general algorithm should be applicable to many different games. When considering a game expressed in this model, game specificities will not clutter the demonstration of the validity of the theorem.

Paranoid reduction

The well known minimax algorithm is a basic game tree search technique [RN02, Chapter 6] for turn taking games. It explore a partial game tree to a fixed depth in a depth first manner. The number of nodes to be explored is usually exponential in the depth of the search and some of the explored nodes will actually not contribute to the final result. The alpha-beta algorithm is a conservative improvement of minimax the avoid exploring some unnecessary nodes. However alpha-beta is based on the game being two-player and zero-sum. Several extensions of alpha-beta try to deal with the multiplayer case [Stu02]. The *paranoid algorithm* is such an extension and we will specify it using GAs.

Given a multiplayer turn taking game $\Gamma = (\Sigma, i, F, P, M, t, \mathbb{O}, o)$, the *paranoid reduction* of Γ for the player $p \in P$ is $(\Sigma, i, F, \{p, -p\}, M', t', \mathbb{O}, o)$ where $-p$ is a new player and M' and t' reflect the changing of players. M' and t' are changed such that the state transitions possible with t' are exactly those possible with t but instead of calling to any player outside p in P, a call is made to $-p$. The preferences of the new player $-p$ are exactly the opposites of the preferences of p : $o_1 \leq_p o_2 \implies o_2 \leq_{-p} o_1$. Thus, the adapted game is a zero sum game and the opponents are merged into a single opponent. The alpha-beta can then be used on this adapted game and the resulting move choice for p will be conveyed back to the original game.

Generality and limitations of the model

Zero sum turn taking games encompass many usual board games such as Chess, Go, Hex, or even Chinese checkers. Using a random player to simulate the dice rolls enables us to represent backgammon and other games that involve chance[QC07].

Puzzles games encompass Same Game, 9-tiles etc. They also encompass problems such as travelling salesman in which a state would represent the history of the cities visited so far, and the outcome would be the distance travelled given a history of visited cities.

Classic games of game theory such as the Prisoner's Dilemma, Rock-paper-scissors can also be directly represented using GA. Indeed, Normal Form Games with complete information can directly be represented through GA.

Games with incomplete information is a very interesting class of games but it lies out of the scope of this thesis. The definition of GA can probably be extended to include such games, drawing inspiration from EFG with incomplete information.

Other game models

Relation to Combinatorial Game Theory (CGT)

Combinatorial Game Theory is another model tool for two players games. One of the main differences between CGT and GA theory is their scope. GA strives for more generality than CGT and can indeed model puzzles, multiplayer games as well as non sequential games or non zero sum games.

Relation to EFG

GAs are very similar to EFGs. They have the same expressive power, that is every problem that can be represented by a GA can be represented in extensive-form and every game that can be represented in extensive can be represented by a GA. Both representations are based on directed graphs. Extensive-form is based on trees while GA is based on acyclic graphs. Therefore the conversion of an extensive-form game to a corresponding GA is straightforward. The reverse conversion can be more involved because one needs to obtain the unfolding graph of the GA.

Despite this equivalence in expressivity, GAs can sometimes be much more compact. In the most extreme cases, the tree in extensive-form is exponentially bigger than the underlying graph of the GA. A concrete albeit artificial example is given in section 5.4 with the game LeftRight. Another example is given with the Nim game[Bou01], compare the number of nodes of figure 2.1 and 2.2. Having a compact representation often allows for more efficient algorithms as will be shown in section 5.

Another reason to be interested in GA beyond the potential efficiency of algorithm is that it allows to reflect naturally real game situations in a more satisfactory way than extensive-form games. A huge number of board game situations depend only on the position set on the board and not on the previous moves[1]. The GA representation of such games can potentially keep the one to one relationship between board states and states of the automaton while the board states would be represented by many nodes in the corresponding extensive-form game depending on the previous moves.

[1] for counter-examples, think about castling or the 50-moves rule in Chess or the ko rule in Go

2.2 Solving a game

Definitions

Let $\Gamma = (\Sigma, i, F, P, M, t, \mathbb{O}, o)$ be a game. A *pure strategy* σ_p for player $p \in P$ is a mapping from each state of $s \in \Sigma$ to a legal move for p in s : $\sigma_p : \Sigma \to M_p, \sigma_p(s) \in L_s(p)$. A *strategy profile* σ is a tuple containing a strategy for every player : $\sigma = (\sigma_{p_1}, \ldots, \sigma_{p_k})$.

Given a strategy profile σ, we define the *game result* of σ to be the outcome in the final state obtained by the following procedure. Start in the initial state i, move from a state s to the state $t_s(\sigma_{p_1}(s), \ldots, \sigma_{p_k}(s)))$ until a final state is reached.

Let σ_1 and σ_2 be two strategies for the player p_j in the game G. We say that σ_1 dominate σ_2 if for every set of strategies σ used by the other players, the game result of the strategy profile consisting in joining σ with σ_1 is better than the game result of joining σ with σ_2 according to p_j. The domination relation is a preorder for the set of strategies of a given player. We call maximal elements of the domination relation *dominant strategies*.

If the preference of every player is a total order and if the game is sequential, then the domination relation is a total preorder for every player. In this case, we call dominant strategies *optimal strategies*. Taking an optimal strategy for every player always lead to the same game result which we call the *value* of the game.

Types of solving

The subgame of Γ starting at $s \in \Sigma$ is the following GA : $(\Sigma', s, F', P, M, t', \mathbb{O}, o)$ where Σ' is the restriction of Σ to the states reachable from s, F' is the restriction of F to the final states reachable from s and t' is the restriction of t to Σ'.

A game Γ is *weakly solved* if we know an optimal strategy for every player of Γ. The game of checkers was weakly solved in 2007 by Jonathan Schaeffer [SBB+07] : the best play is known from the initial state but the value of an arbitrary position is not explicitly determined.

It is *strongly solved* if we know an optimal strategy for every player of Γ in the game Γ and in every subgame of Γ. The game of Nim was completely solved in 1901 by Charles Bouton [Bou01], the perfect play is known for every possible position.

It is *ultra weakly solved* if we know its value. For instance, it is known that the game of Hex is a first player win but no explicit optimal strategy is known for sufficiently big sizes [Maa05, Chapter 4].

Admissible bounds

We call *reachable outcomes* of a given state s the set of outcomes corresponding to the final states final from s. We call *rational outcomes* of s the set of outcomes corresponding to the final states that can be reached by following a dominant strategy for every player.

An *admissible outcome bound* on a state s is a superset of the rational outcomes of s. An admissible outcome bound is *loose* if it is not equal to the rational outcomes, (it is a strict superset), otherwise the bound is said to be *tight*.

2.3 Monte-Carlo Tree Search

As can be guessed from the name, the Monte-Carlo Tree Search algorithm is based on Monte-Carlo simulations and on a tree search procedure. The simulations and the search procedure are interleaved so that four steps can be identified. Namely, the descent, the selection, the simulation and the backpropagation. These four steps are repeated iteratively until a stopping condition is fulfilled[2].

Random playouts

A random playouts from a game state s is a continuation of the game starting at s with each transition randomly selected until a final state is reached. The basic idea behind these Monte-Carlo simulations is that the expected outcome of random playouts played from a state s can serve as an evaluation of s.

Of course the expected outcome of a continuation from s between perfect players played is by definition the best evaluation of s but for most game states, perfect players are not computationally feasible. On the other hand, the expectation of a random playout can be efficiently estimated by averaging the results of several successive random playouts.

The expectation in a state is not the true value (see section 2.2) of that state. The estimation can be improved, or at least the convergence can be accelerated using various methods. A promising technique seems to be *simulation balancing* [ST09] but it does not constitute the subject of this thesis.

Descent and update

As opposed to the description in section 5, the classic MCTS algorithm actually constructs an unfolding graph of the game (see section 2.1). In the constructed tree, the root node corresponds to the submitted position s and each node

[2] common stopping conditions include threshold on the number of simulation or on the time elapsed

correspond to a position reachable from s. An edge is labelled with the move needed from the father node to the child node.

An aggregation of the results of the playouts related to a node n is stored in n[3]. Other data used for heuristics can also be stored in the nodes, for instance in section 4 we will need to store admissible bounds.

The expansion of the tree in MCTS is similar to one in a best first search algorithm. Therefor the tree needs to be stored in memory. For each playout conducted, a node is added to the tree[4].

The random simulations are always started from leaf nodes of the tree. Deciding which leaf should give rise to the next simulation is done through the descent of the tree. Starting from the root node, a move is selected among the outgoing edges. The corresponding child is reached and the process continues until a leaf is reached. The process can also stop in a internal node if not every child has been created yet. How the next child is selected is detailed in section 2.3. Once the process has stopped and the tree has been expanded, a random simulation is run and the tree is updated accordingly.

Updating the tree given the outcome of a random playout is simple enough. One needs either the list of the traversed nodes, or only the leaf node from which the simulation was conducted if father nodes are accessible from their children. The basic information stocked in each node n is the total number of playouts that traversed n and the mean outcome of these playouts.

It is necessary to specify what is meant by the mean outcome in a node n. We first need a mapping from $\mathbb{O}$ to $\mathbb{R}$, for instance in Chess $\mathbb{O} = \{\text{Black}, \text{White}, \text{Draw}\}$ where Black indicates that Black has won the game, we can have Black $\rightarrow 0$, Draw $\rightarrow 0.5$, White $\rightarrow 1$. If the game is a puzzle or a two-players constant sum game, the concept is not ambiguous. Otherwise we need to store the mean outcome of the player who is in turn in the father of n.

Selection

Deciding which edge should be explored can be viewed as a multi-armed bandit problem [ACBF02, KS06]. Priority is naturally given to promising edges, that is edges leading to a high mean outcome[5]. However the mean outcome might not be accurate; the confidence in the mean outcome is a function of the number of playouts. Hence we might also want to emphasize nodes with a low number of playouts in order to have a more reliable mean outcome. This is called the *exploration — exploitation* dilemma.

A solution to this dilemma in the case of a tree is presented in [KS06] through the use of an *Upper Confidence bound*. The UCT value is defined for each node x to be $u(x) = \mu(x) + c \times \sqrt{\frac{\log p(x)}{n(x)}}$ where $\mu(x)$ is the mean outcome,

[3] we will show in section 5 that it is better to actually store results related to edges

[4] many implementations put a threshold on the tree size

[5] from the view point of the player whose turn it is

$n(x)$ is the total number of playouts that went through x and $p(x)$ is the total number of playouts that went through the father of x. The edge selected is the one maximizing the UCT value. Following the UCT policy ensures that the mean outcome will converge towards the game value while the regret is minimized.

Although the UCT theoretically converges to the minimax outcome, in practical settings it might be interesting to have a quick idea on which move is to be selected at the root node without performing a huge number of random simulations. Heuristics can often improve the playing strength of the algorithm by providing early advice on which area of the game tree is best explored. These heuristics cannot be successfully applied to every GA as they are based on domain specific knowledge. We will quickly present the All Moves as First (AMAF) heuristic [HPW09] and its integration through Rapid Action Value Estimation (RAVE) [GS07], for it can actually be applied to a non negligible number of games.

For a GA $(\Sigma, i, F, P, M, t, \mathbb{O}, o)$, the number of move labels is usually much smaller than the number of state transitions $\mathrm{card}(M) \ll \mathrm{card}(\{(s, m, s') \in \Sigma \times M \times \Sigma | t(s, m) = s'\})$. For instance in the variant for the game Nim used in figure 2.1 and in figure 2.2, there are 6 move labels ($\{1, 2, 3\}$ for one player and for its opponent) while the number of edges is 16 in the minimal GA and 27 for the unfolded GA. The AMAF heuristic can be applied when the preceding is true and the move labels actually denote some game concepts. For instance, AMAF has been successfully applied to Go where move labels corresponds to positions where stones are played.

Consider a node n and an edge e going out of n and labelled m. After a certain number of playouts in the whole MCTS tree, the mean outcome for e is based only on the number of playouts that went through e which is likely to be small, therefore the mean outcome is not very reliable. However the number of playouts that went through edges labelled m is much higher. The principle of the AMAF algorithm is to use the data for the move label m instead of the edge e when calculating the UCT value of e. AMAF improves the playing level in Go because the final board state is not affected by when stones were played[6] but only by their positions. That is, the contribution from a move to a final board state lies in the move label as well as in the corresponding edges.

The RAVE algorithm can bridge the gap between AMAF and the normal behavior of UCT. When only a few simulations are available for an edge e, the edge value has a high variance and is not reliable so the AMAF should be used to evaluate the edge; conversely when many simulations were realized, the edge value is reliable and is more specific than the label value. Using RAVE consists in a smooth transition between the label value and the edge value. We denote the edge value of an edge e by $\mu_{\text{edge}}(e)$ and its label value by $\mu_{\text{label}}(e)$. The RAVE value of e is thus defined as $\mu_{\text{RAVE}}(e) = (1 - \beta(e))\mu_{\text{edge}}(e) + \beta(e)\mu_{\text{label}}(e)$,

[6] we omit capture rules for the sake of simplicity of explanation

with $\beta(e)$ decreasing from 1 to 0 as the number of simulations through e increases.

2.4 Restrictions for this work

In the rest of this work, we will assume some restrictions over the material presented in this section. These restrictions can be motivated by difficulties to generalize our results, or simply to ease the exposition. The restrictions described here do not affect section 3.

Kind of games considered

We will not consider general GAs but rather make a certain number of restrictive assumptions. First, the GAs considered are sequential games. Second, we assume no chance is involved. Finally, we are not interested in multiplayer games, that is we are only dealing with puzzles and two-players games. These hypotheses are geared toward the MCTS algorithm. They are consistent with most of the papers published on MCTS. Indeed, most of the publications related to MCTS deal with the game of Go which is a two-players sequential, zero sum game.

MCTS

We will also limit ourselves to the most basic MCTS algorithm. Indeed, we will not use the RAVE method, nor the AMAF heuristic in the following sections. Similarly, we will not perform any simulation balancing. Section 4 is perfectly compatible with these improvement of the MCTS framework but this limitation makes presentation easier. We leave extension of the methods in section 5 to the AMAF heuristic and integration with the RAVE algorithm as future works.

Admissible bounds

In section 4, the introduction of admissible bounds to MCTS is presented. We defined in section 2.2 admissible bounds to be superset of the rational outcomes. In this work, though, we only consider admissible bounds that form an interval. For instance if the possible outcomes are $\{\text{Win}, \text{Draw}, \text{Loss}\}$, we will not consider $\{\text{Win}, \text{Loss}\}$. Using general admissibility bounds instead of interval admissibility bounds is beyond the scope of section 4 and is left as future work[7].

[7] it not yet clear whether it could be useful in practice

3 A compiler for the Game Description Language

Contents

3.1 Introduction

GGP has been described as a *Grand AI Challenge* [GL05, Thi09] and it has spanned research in several directions. Some works aim at extracting knowledge from the rules [Clu07], while GGP can also be used to study the behavior of a general algorithm on several different games such as in [MC10]. Another possibility is studying the possible interpretation and compilation of the GDL [Wau09] in order to process games events faster.

While the third direction mentioned does not directly contribute to AI, it is important for several reasons. It can enable easier integration with playing programs and let other researchers work on GGP without bothering with interface writing, GDL interpreting and such non AI details. Having a faster state machine may move the speed bottleneck of the program from the GGP module to the AI module and can help performance distinction between

different AI algorithms. Finally, as GDL is high level language, compiling the rules to a fast state machine and extracting knowledge from the rules are sometimes similar things. For instance, factoring a game as in [GST09] could greatly improve some compilation schemes.

This direction is also that of our work. We focus on the compilation of rulesheets in the GDL into GAs. More precisely, the compiler described in this work takes a game rulesheet written in GDL as an input and outputs an OCaml module that can be interfaced with our playing program also written in OCaml. The module and the playing program are then compiled to native code by the standard OCaml compiler [LDG+96] so that the resulting program runs in reasonable time. The generated module exhibits a GA-like interface. Figure 1.1 sums the normal usage of our compiler up.

The remaining of this section is organized as follows: we first describe the GDL, then the various passes used by our compiler to generate OCamlcode. To conclude, we briefly present some experimental considerations and a list of extension to this compiler that seem suitable.

3.2 Game Description Language

The Game Description Language [LHG06] is based on Datalog and allows to define a large class of GAs (see section 2.1 for a formal definition of a GA). It is a rule based language that features function constants, negation-as-a-failure and variables. Some predefined predicates confer the dynamics of a GA to the language.

Syntax

A limited number of syntactic constructs appear in GDL[1]. Predefined *predicates* are presented in table 3.1. Function constants may appear and have a fixed arity determined by context of the first appearance. Logic operators are simply `or`, `and` and `not`; they appear only in the body of rules. Existentially quantified variables may also be used bearing some restrictions defined in section 3.2. Rules are compose of a head term and a body made of logic terms.

A GDL source file is composed of a set of grounded terms that we will call B for *base facts* and a set of rules. The Knowledge Interchange Format is used for the concrete syntax of GDL.

The definition of GDL [LHG06] makes sure that each variable of a negative literal also appears in a positive literal. The goal of this restriction is probably to make efficient implementations of GDL easier. Indeed, it is possible to wait until every variable in a negative literal are bound before checking if the corresponding fact is in the knowledge base. Put another way, it enables to

[1] we depart a bit from the presentation in [LHG06] to ease the sketch of our compiler

Name	Arity	Appearance
`does`	2	body
`goal`	2	base, body, head
`init`	1	base, head
`legal`	2	base, body, head
`next`	1	head
`role`	1	base
`terminal`	0	base, body, head
`true`	1	body

Table 3.1: Predefined predicates in GDL with their arity and restriction on their appearance. *Base* means th

deal with the negation by only checking for ground terms in the knowledge base. This property is called *safety*.

Semantics

The base facts B defined in the source file are always considered to hold. The semantics also make use of the logical closure over the rules defined in the files, that is at a time τ, the rules allow to deduce more facts that are true at τ based on facts that are known to hold at τ.

The semantics of a program in the GDL can be described through the GA formalism as follows

- The set of players participating to the game is the set of arguments to the predicate `role`.
- A state of the GA is defined by a set of facts that is closed under application of the rules in the source files.
- The initial state is the closure over the facts that are arguments to the predicate `init`.
- Final states are those in which the fact `terminal` holds.
- For each player p and each final state s, exactly one fact of the form `goal`(p, o_p) holds. We say that $0 \leq o_p \leq 100$ is the reward for player p in s. The outcome o in the final state is the tuple $(o_{p_1}, \ldots, o_{p_k})$.
- The preference relation of the players is the natural ordering on their reward. That is $(o_{p_1}, \ldots, o_{p_k}) \leq_p (o'_{p_1}, \ldots, o'_{p_k}) \iff o_p \leq o'_p$.
- For each player p and each state s, the legal moves for p in a state s are $L_s(p) = \{m_p | \texttt{legal}(p, m_p)$ holds in $s\}$

- The transition relation is defined by using the predicates `does` and `next`. For a move $m = (m_{p_1}, \ldots, m_{p_k})$ in a state s, let q be the closure of the following set of facts : $s \cup \{\texttt{does}(p_1, m_{p_1}), \ldots, \texttt{does}(p_k, m_{p_k})\}$. Let n be the set of fact f such that $\texttt{next}(f)$ holds in n. The resulting state of applying m to s is the closure of the set $\{\texttt{true}(f) | f \in n\} \cup B$.

3.3 Intermediate languages

Translating GDL programs to programs in the target language can be decomposed into several steps. Each of this step corresponds to the translation from one language to another. We used three intermediate languages in this work. The first one *mini-GDL* is a desugared version of GDL. In the second intermediate language, *normal-GDL*, the rules are decomposed until a normal form is reached. The transition between a declarative language and an imperative one takes place when the program is transformed into the *Inverted Intermediate Language (IIL)*. Finally the program in the IIL is transformed in an abstract syntax tree of the target language.

Desugaring

Mini-GDL is a subset of GDL that has the same expressivity. Disjunctions in rules are no longer possible and the equal predicate is not used.

The right hand side of a rule in GDL contains a logical formula made of an arbitrary nesting of conjunctions, disjunctions and negations[2]. The first step in transforming a rule from GDL to mini-GDL is to put in Disjunctive Normal Form (DNF).

A rule in DNF can now be split over several as many subrules as the number of disjunctions it is made of. Indeed a rule with a conclusion c and a right hand side made of the disjunction of two hypotheses h_1 and h_2 is logically equivalent to two rules with h_1 and h_2 as hypotheses and the same conclusion c : $\{c \leftarrow h_1 \vee h_2\} \equiv \{c \leftarrow h_1, c \leftarrow h_2\}$.

A rule involving equalities can be turned into an equivalent rule without any equality. The transformation is made of two recursive processes, a substitution and a decomposition. When we are faced with an equality between t_1 and t_2 in a rule r, either at least one of the two terms is a variable (we'll assume it is t_1) or both are made of a function constant and a list of subterms. In the former case the substitution takes place : we obtain an equivalent rule by replacing every instance of t_1 in r by t_2 and dropping the equality. In the latter case, if the function constants are different then the equality is unsatisfiable and r cannot fire else we can replace the equality between t_1 and t_2 by equalities between

[2] although there are some restriction on the negation possibilities

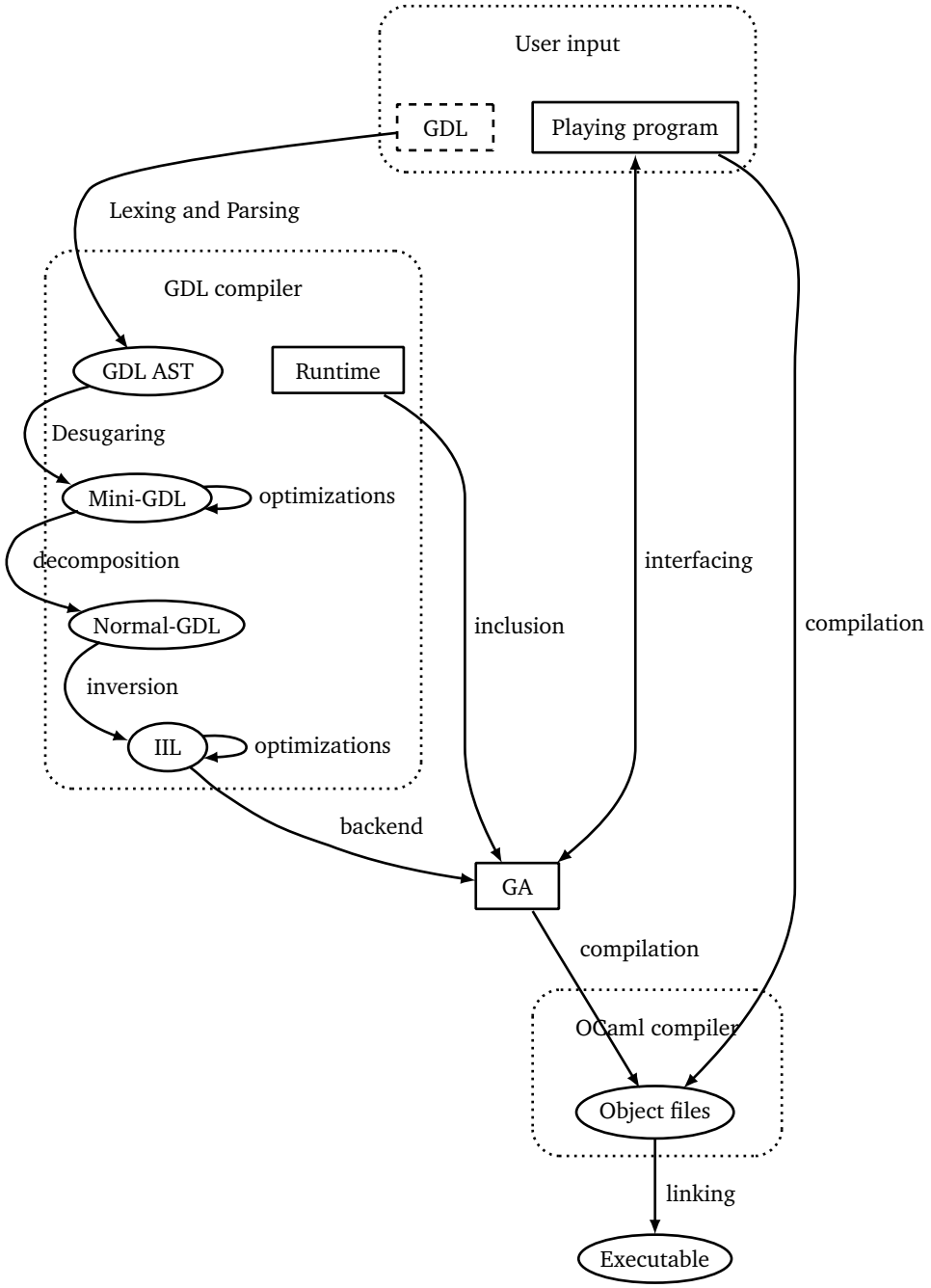


Figure 3.1: Steps and transformations between a GGP program written in GDL and the executable.

the subterms of t_1 and the subterms of t_2[3]. We can carry this operation until the rule obtained does not have any equality left.

Decomposition

GDL is built upon Datalog, therefore techniques applied to Datalog are often worth consideration in GDL. One such technique consists in decomposing the rules until a normal form is obtained. [LS09] presented a decomposition such that each rule in normal form is made of at most two literals in the right hand side. This decomposition is briefly recalled, then the adaptations needed to use it with GDL are presented.

Let $r = c \leftarrow t_1 \wedge t_2 \wedge t_3 \wedge \cdots \wedge t_n$ be a rule with $n > 2$ hypotheses. We create a new term t_{new} and replace r by the following two rules. $r_1 = t_{\text{new}} \leftarrow t_1 \wedge t_2$ and $r_2 = c \leftarrow t_{\text{new}} \wedge t_3 \wedge \cdots \wedge t_n$. Since variables can occur in the different terms and in c, t_{new} needs to carry the right variables so that c is instantiated with the same value when r is fired and when r_1 then r_2 are fired. This is achieved by embedding in t_{new} exactly the variables that appear on the one hand in t_1 or t_2 and on the second hand in either c or any of $t_3, \ldots, t_n$. The fact that variables that appear in t_1 or t_2 but not in $t_3, \ldots, t_n$ or c do not appear in t_{new} ensures that the number of intermediate facts is kept relatively low.

The right hand side of rules in mini-GDL are not terms but literals so some care has to be taken to adapt negative literals properly. GDL involves stratified negation which is not extensively covered by the presentation in [LS09] of the decomposition, but as Liu and Stoller acknowledge, the extension is straightforward.

The decomposition of rules calls for an order of the literals, the simplest such order is the one inherited from the mini-GDL rule. However it is naturally interesting that the safety property (see section 3.2) holds after the rules are decomposed. Consequently, literals might need to be reordered so that every variable appearing in a negative literal m appears in a positive literal before m. The programmer who wrote the game in Knowledge Interchange Format (KIF) might have ordered the literals to strive for efficiency or the literals might have been reordered by optimizations at the mini-GDL stage[4]. In order to minimize interferences with the original ordering, only negative literals are moved. The following fixpoint algorithm is used to reorder the literals and decompose the rules.

Inversion

After the decomposition is performed, the *inversion* transformation takes place. Each function constant and each predicate will generate a function in the target

[3] function constants with different arities are always considered to be different.

[4] no such heuristic is implemented yet however

Input: set of rules Γ
Result: Γ is decomposed
while *There exists a rule r with more than 3 literals in Γ* **do**
  let $\gamma =$ `decompose-step(`r`);`
  $\Gamma := \Gamma \setminus \{r\}$;
  $\Gamma := \Gamma \cup \gamma$;
end

Algorithm 3.1: Fixpoint to transform the set of mini-GDL rules into a set of normal-GDL rules

Input: rule $r = c \leftarrow l_1 \wedge l_2 \wedge \cdots \wedge l_n$
Output: set of rules equivalent to r
if l_1 *is a negative literal and does not correspond to a ground term* **then**
  let $i =$ index of the first positive literal;
  let $r' = c \leftarrow l_i \wedge l_1 \wedge \cdots \wedge l_{i-1} \wedge l_{i+1} \wedge \cdots \wedge l_n$;
  return $\{r'\}$
else
  if l_2 *is a negative literal and the variable in l_2 do not appear in l_1* **then**
    let $i =$ index of the first positive literal after l_2;
    let $r' = c \leftarrow l_1 \wedge l_i \wedge l_2 \wedge \cdots \wedge l_{i-1} \wedge l_{i+1} \wedge \cdots \wedge l_n$;
    return $\{r'\}$
  else
    compute ρ (resp. ρ') the set of variables appearing in l_1 or in l_2 (resp. in l_3 or ... or in l_n);
    let $t_{\text{new}} =$ new term made of the variables in $\rho \cap \rho'$;
    let $l_{\text{new}} =$ the positive literal based on t_{new};
    let $r' = t_{\text{new}} \leftarrow l_1 \wedge l_2$ and **let** $r'' = c \leftarrow l_{\text{new}} \wedge l_3 \wedge \cdots \wedge l_n$;
    return $\{r', r''\}$
  end
end

Algorithm 3.2: One step in the decomposition of a rule

language. This function would in turn trigger the functions corresponding to head of rules in the body of which the original function constant appeared.

For instance the following Tic-tac-toe rules express in GDL the fact that if a player has a column or a row then that player has a line, and that a line is a terminal condition.

```
(<= (line ?player) (column ?player)
                   (row ?player))
(<= terminal (line ?player))
```

These rules are roughly translated in the following pseudo-code.

```
let fun_column (var_player) =
```

```
  add_to_DB ("column", var_player);
  if present_in_DB ("row", var_player)
  then fun_line (var_player)

let fun_row (var_player) =
  add_to_DB ("row", var_player);
  if present_in_DB ("column", var_player)
  then fun_line (var_player)

let fun_line (var_player) =
  add_to_DB ("line", var_player);
  fun_terminal ()
```

The inversion transformation must also take into account the fact that a given function constant can naturally appear in several rule bodies. Such a function constant need still to be translated into a single function of the target language. Therefore, an important step of the inversion transformation is to associate to each function constant f the couples (rule head, remaining rule body) of the rules that can be triggered by f.

Target language

One the game has been translated to the IIL, it can be processed by the backend to have a legitimate OCaml program. OCaml was chosen as a backend for several reasons. Efficiency is an important criterion and OCaml is compiled to native code. Stratification was handled with a continuation style: when in a strate n a function f of strate $m > n$ was called, the computation of f was postponed until the strate m was reached. First class functions make this manipulation very easy[5]. Finally, even though Haskell also satisfies these requirements, we already had a playing program written in OCaml and generating a module in the same language makes interfacing easier.

3.4 Discussion and future works

Performance

The usual interpretation of GDL is done through an off the shelf prolog interpreter such as YAP. A simple benchmark for a GGP engine is to play a large number of games using a basic Monte-Carlo AI[6]. We rapidly compared the speed of programs for several single player and two players games generated by our compiler named `dlc`, to data for a YAP based engine which we denote by `ary`[7].

[5] or rather, the absence of first class functions make it clumsy!

[6] details can be found in [Wau09]

[7] this data was kindly provided by Jean Méhat

The comparison cannot not be followed too seriously as the tests were realized on vastly different computers. `ary` was tested on a 8 core desktop computer while `dlc` was tested on a laptop. The laptop was a year more recent than the desktop computer, so the results can still give a vague idea on the performance.

On some games, such as Tic-tac-toe, `dlc` was between 5% and 20% faster than `ary`. That is, the number of random playouts realized by our engine was in average 5% to 20% superior to the number of random playouts realized by `ary` during the same time. The worst result noted was on Connect4 where `dlc` was 80% slower than `ary`.

Although these results are far less impressive than those obtained by Kevin Waugh in [Wau09], they are still encouraging for several reasons. The generated code is not optimized by our compiler and some runtime data structures are clearly suboptimal. To the best of our knowledge it is the first forward chaining approach to GGP and rulesheets are tested and optimized with resolution based engines.

Future works

Several optimizations would enhance this proof-of-concept compiler. Magic sets [KSS91] or a more recent method called *demand transformation*[TL10, Section 4] can be used to direct the bottom-up evaluation since the needed queries are known beforehand. Better performance could also be obtained by using the intermediate language described in [LS09] rather than IIL. Finally, it would be interesting to develop other back-ends than OCaml, Haskell and C++ are appealing.

4 Bounded MCTS

Contents

The following section draws heavily from [CS10].

MCTS is a successful algorithm used in many state of the art game engines. We propose to improve a MCTS solver when a game has more than two outcomes. It is for example the case in games that can end in draw positions. In this case it improves significantly a MCTS solver to take into account bounds on the possible scores of a node in order to select the nodes to explore. We apply our algorithm to solving Seki in the game of Go and to Connect Four.

4.1 Introduction

In LOA, MCTS has been successfully combined with exact results in a MCTS solver [WBS08]. We propose to further extend this combination to games that have more than two outcomes. Example of such a game is playing a Seki in the game of Go: the game can be either lost, won or draw (i.e. Seki). Improving MCTS for Seki and Semeai is important for Monte-Carlo Go since this is one of the main weaknesses of current Monte-Carlo Go programs. We also address the application of our algorithm to Connect Four that can also end in a draw.

The second section deals with the state of the art in MCTS solver, the third section details our algorithm that takes bounds into account in a MCTS solver, the fourth section explains why Seki and Semeai are difficult for Monte-Carlo Go programs, the fifth section gives experimental results.

4.2 Monte-Carlo tree search solver

MCTS is able to converge to the optimal play given infinite time, however it is not able to prove the value of a position if it is not associated to a solver. MCTS is not good at finding narrow lines of tactical play. The association to a solver enables MCTS to alleviate this weakness and to find some of them.

Combining exact values with MCTS has been addressed by Winands et al. in their MCTS solver [WBS08]. Two special values can be assigned to nodes : $+\infty$ and $-\infty$. When a node is associated to a solved position (for example a terminal position) it is associated to $+\infty$ for a won position and to $-\infty$ for a lost position. When a max node has a won child, the node is solved and the node value is set to $+\infty$. When a max node has all its children equal to $-\infty$ it is lost and set to $-\infty$. The descent of the tree is stopped as soon as a solved node is reached, in this case no simulation takes place and 1.0 is backpropagated for won positions, whereas -1.0 is backpropagated for lost ones.

Combining such a solver to MCTS improved a LOA program, winning 65% of the time against the MCTS version without a solver. Winands et al. did not try to prove draws since draws are exceptional in LOA.

4.3 Integration of score bounds in MCTS

We assume the outcomes of the game belong to an interval $[minscore, maxscore]$ of $\mathbb{R}$, the player *Max* is trying to maximize the outcome while the player *Min* is trying to minimize the outcome.

In the following we are supposing that the tree is a minimax tree. It can be a partial tree of a sequential perfect information deterministic zero-sum game in which each node is either a *max-node* when the player *Max* is to play in the associated position or a *min-node* otherwise. Note that we do not require the child of a *max-node* to be a *min-node*, so a step-based approach to MCTS (for instance in Arimaa [Koz09]) is possible. It can also be a partial tree of a perfect information deterministic one player puzzle. In this latter case, each node is a max-node and *Max* is the only player considered.

We assume that there are legal moves in a game position if and only if the game position is non terminal. Nodes corresponding to terminal game positions are called *terminal nodes*. Other nodes are called *internal nodes*.

Our algorithm adds score bounds to nodes in the MCTS tree. It needs slight modifications of the backpropagation and descent steps. We first define the

bounds that we consider and express a few desired properties. Then we show how bounds can be initially set and then incrementally adapted as the available information grows. We then show how such knowledge can be used to safely prune nodes and subtrees and how the bounds can be used to heuristically bias the descent of the tree.

Pessimistic and optimistic bounds

For each node n, we attach a pessimistic (noted $\mathrm{pess}(n)$) and an optimistic (noted $\mathrm{opti}(n)$) bound to n. Note that optimistic and pessimistic bounds in the context of game tree search were first introduced by Hans Berliner in his B* algorithm [Ber79]. The names of the bounds are defined after *Max*'s point of view, for instance in both max- and min-nodes, the pessimistic bound is a lower bound of the best achievable outcome for *Max* (assuming rational play from *Min*). For a fixed node n, the bound $\mathrm{pess}(n)$ is increasing (resp. $\mathrm{opti}(n)$ is decreasing) as more and more information is available. This evolution is such that no false assumption is made on the expectation of n : the outcome of optimal play from node n on, noted $\mathrm{real}(n)$, is always between $\mathrm{pess}(n)$ and $\mathrm{opti}(n)$. That is $\mathrm{pess}(n) \leq \mathrm{real}(n) \leq \mathrm{opti}(n)$. If there is enough time allocated to information discovering in n, $\mathrm{pess}(n)$ and $\mathrm{opti}(n)$ will converge towards $\mathrm{real}(n)$. A position corresponding to a node n is solved if and only if $\mathrm{pess}(n) = \mathrm{real}(n) = \mathrm{opti}(n)$.

If the node n is terminal then the pessimistic and the optimistic values correspond to the score of the terminal position $\mathrm{pess}(n) = \mathrm{opti}(n) = \mathrm{score}(n)$. Initial bounds for internal nodes can either be set to the lowest and highest scores $\mathrm{pess}(n) = minscore$ and $\mathrm{opti}(n) = maxscore$, or to some values given by an appropriate admissible heuristic [HNR68]. At a given time, the optimistic value of an internal node is the best possible outcome that *Max* can hope for, taking into account the information present in the tree and assuming rational play for both player. Conversely the pessimistic value of an internal node is the worst possible outcome that *Max* can fear, with the same hypothesis. Therefore it is sensible to update bounds of internal nodes in the following way.

If n is an internal max-node then	If n is an internal min-node then
$\mathrm{pess}(n) := \max_{s \in \mathrm{children}(n)} \mathrm{pess}(s)$	$\mathrm{pess}(n) := \min_{s \in \mathrm{children}(n)} \mathrm{pess}(s)$
$\mathrm{opti}(n) := \max_{s \in \mathrm{children}(n)} \mathrm{opti}(s)$	$\mathrm{opti}(n) := \min_{s \in \mathrm{children}(n)} \mathrm{opti}(s)$

Updating the tree

Knowledge about bounds appears at terminal nodes, for the pessimistic and optimistic values of a terminal node match its real value. This knowledge is then recursively upwards propagated as long as it adds information to some node. Using a fast incremental algorithm enables not to slow down the MCTS procedure.

Let s be a recently updated node whose parent is a max-node n. If $\mathrm{pess}(s)$ has just been increased, then we might want to increase $\mathrm{pess}(n)$ as well. It happens when the new pessimistic bound for s is greater than the pessimistic bound for n : $\mathrm{pess}(n) := \max(\mathrm{pess}(n), \mathrm{pess}(s))$. If $\mathrm{opti}(s)$ has just been decreased, then we might want to decrease $\mathrm{opti}(n)$ as well. It happens when the old optimistic bound for s was the greatest among the optimistic bounds of all children of n. $\mathrm{opti}(n) := \max_{s \in \mathrm{children}(n)} \mathrm{opti}(s)$. The converse update process takes place when s is the child of a min-node.

When n is not fully expanded, that is when some children of n have not been created yet, a dummy child d such that $\mathrm{pess}(d) = minscore$ and $\mathrm{opti}(d) = maxscore$ can be added to n to be able to compute conservative bounds for n despite bounds for some children being unavailable.

Input: node s
Result: Update the pessimistic bounds of the ancestors of s
if *s is not the root node* **then**
  let $n =$ the parent of s;
  let $old\_pess = \mathrm{pess}(n)$;
  if $old\_pess < \mathrm{pess}(s)$ **then**
    if *n is a* Max *node* **then**
      $\mathrm{pess}(n) := \mathrm{pess}(s)$;
      `prop-pess` (n);
    else
      $\mathrm{pess}(n) := \min_{s' \in \mathrm{children}(n)} \mathrm{pess}(s')$;
      if $old\_pess > \mathrm{pess}(n)$ **then**
        `prop-pess` (n);
      end
    end
  end
end

Algorithm 4.1: `prop-pess` : Propagating pessimistic bounds

Pruning nodes with alpha-beta style cuts

Once pessimistic and optimistic bounds are available, it is possible to prune subtrees using simple rules. Given a max-node (resp. min-node) n and a child s of n, the subtree starting at s can safely be pruned if $\mathrm{opti}(s) \leq \mathrm{pess}(n)$ (resp. $\mathrm{pess}(s) \geq opti(n)$).

To prove that the rules are safe, let's suppose n is an unsolved max-node and s is a child of n such that $\mathrm{opti}(s) \leq \mathrm{pess}(n)$. We want to prove it is not useful to explore the child s. On the one hand, n has at least one child left unpruned. That is, there is at least a child of n, s^+, such that $\mathrm{opti}(s') > \mathrm{pess}(n)$. This comes directly from the fact that as n is unsolved, $\mathrm{opti}(n) > \mathrm{pess}(n)$, or

Input: node s
Result: Update the optimistic bounds of the ancestors of s
if *s is not the root node* **then**
  let $n =$ the parent of s;
  let $old\_opti = \text{opti}(n)$;
  if $old\_opti > \text{opti}(s)$ **then**
    if *n is a* Max *node* **then**
      $\text{opti}(n) := \max_{s' \in \text{children}(n)} \text{opti}(s')$;
      if $old\_opti > \text{opti}(n)$ **then**
        `prop-opti(`n`);`
      end
    else
      $\text{opti}(n) := \text{opti}(s)$;
      `prop-opti(`n`);`
    end
  end
end

Algorithm 4.2: `prop-opti` : Propagating optimistic bounds

equivalently $\max_{s^+ \in \text{children}(n)} \text{opti}(s^+) > \text{pess}(n)$. s^+ is not solved. On the other hand, let us show that there exists at least one other child of n better worth choosing than s. By definition of the pessimistic bound of n, there is at least a child of n, s', such that $\text{pess}(s') = \text{pess}(n)$. The optimistic outcome in s is smaller than the pessimistic outcome in s' : $\text{real}(s) \leq \text{opti}(s) \leq \text{pess}(s') \leq \text{real}(s')$. Now either $s \neq s'$ and s' can be explored instead of s with no loss, or $s = s'$ and s is solved and does not need to be explored any further, in the latter case s^+ could be explored instead of s.

An example of a cut node is given in Figure 4.1. In this figure, the *min-node* d has a solved child (f) with a 0.5 score, therefore the best *Max* can hope for this node is 0.5. Node a has also a solved child (c) that scores 0.5. This makes node d useless to explore since it cannot improve upon c.

Bounds based node value bias

The pessimistic and optimistic bounds of nodes can also be used to influence the choice among uncut children in a complementary heuristic manner. In a max-node n, the chosen node is the one maximizing a value function Q_{max}.

In the following example, we assume the outcomes to be reals from $[0, 1]$ and for sake of simplicity the Q function is assumed to be the mean of random playouts. Figure 4.2 shows an artificial tree with given bounds and given results of Monte-Carlo evaluations. The node a has two children b and c. Random simulations seem to indicate that the position corresponding to node c is less favorable to *Max* than the position corresponding to b. However the lower and

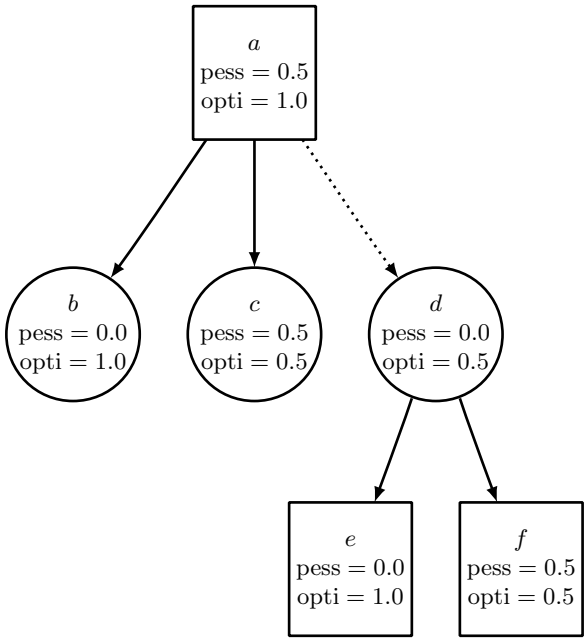


Figure 4.1: Example of a cut. The d node is cut because its optimistic value is smaller or equal to the pessimistic value of its father.

upper bounds of the outcome in c and b seem to mitigate this estimation.

This example intuitively shows that taking bounds into account could improve the node selection process. It is possible to add bound induced bias to the node values of a son s of n by setting two bias terms γ and δ, and rather using adapted Q' node values defined as $Q'_{max}(s) = Q_{max}(s) + \gamma\,\mathrm{pess}(s) + \delta\,\mathrm{opti}(s)$ and $Q'_{min}(s) = Q_{min}(s) - \gamma\,\mathrm{opti}(s) - \delta\,\mathrm{pess}(s)$.

4.4 Why Seki and Semeai are hard for MCTS

The figure 4.3 shows two Semeai. The first one is unsettled, the first player wins. In this position, random playouts give a probability of 0.5 for Black to win the Semeai if he plays the first move of the playout. However if Black plays perfectly he always wins the Semeai.

The second Semeai of figure 4.3 is won for Black even if White plays first. The probability for White to win the Semeai in a random game starting with a White move is 0.45. The true value with perfect play should be 0.0.

We have written a dynamic programming program to compute the exact probabilities of winning the Semeai for Black if he plays first. A probability p of playing in the Semeai is used to model what would happen on a 19×19 board where the Semeai is only a part of the board. In this case playing moves outside of the Semeai during the playout has to be modeled.

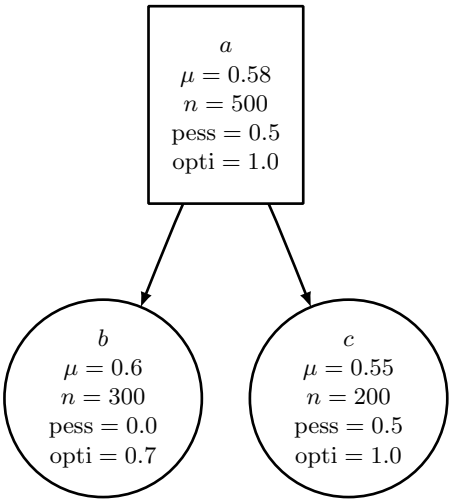


Figure 4.2: Artificial tree in which the bounds could be useful to guide the selection.

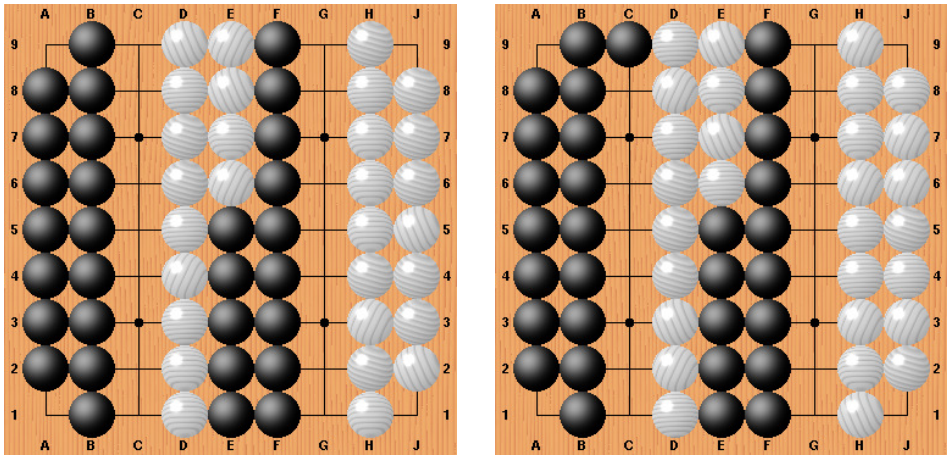


Figure 4.3: An unsettled Semeai and Semeai lost for White.

The table 4.1 gives the probabilities of winning the Semeai for Black if he plays first according to the number of liberties of Black (the rows) and the number of liberties of White (the column). The table was computed with the dynamic programming algorithm and with a probability $p = 0.0$ of playing outside the Semeai. We can now confirm, looking at row 9, column 9 that the probability for Black to win the first Semeai of figure 4.3 is 0.50.

We have computed the tables for a probability $p = 0.80$ of playing outside the Semeai. We choose this probability because it is likely to happen in a real 19×19 game. The dynamic programming was initialized with a probability

Own liberties	Opponent liberties								
	1	2	3	4	5	6	7	8	9
1	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	1.00	0.50	0.30	0.20	0.14	0.11	0.08	0.07	0.05
3	1.00	0.70	0.50	0.37	0.29	0.23	0.18	0.15	0.13
4	1.00	0.80	0.63	0.50	0.40	0.33	0.28	0.24	0.20
5	1.00	0.86	0.71	0.60	0.50	0.42	0.36	0.31	0.27
6	1.00	0.89	0.77	0.67	0.58	0.50	0.44	0.38	0.34
7	1.00	0.92	0.82	0.72	0.64	0.56	0.50	0.45	0.40
8	1.00	0.93	0.85	0.76	0.69	0.62	0.55	0.50	0.45
9	1.00	0.95	0.87	0.80	0.73	0.66	0.60	0.55	0.50

Table 4.1: Proportion of wins for random play on the liberties when always playing in the Semeai

Own liberties	Opponent liberties								
	1	2	3	4	5	6	7	8	9
1	1.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	1.00	0.50	0.30	0.20	0.14	0.11	0.08	0.07	0.05
3	1.00	0.70	0.50	0.37	0.29	0.23	0.18	0.15	0.13
4	1.00	0.80	0.63	0.50	0.40	0.33	0.28	0.24	0.20
5	1.00	0.86	0.71	0.60	0.50	0.42	0.36	0.31	0.27
6	1.00	0.89	0.77	0.67	0.58	0.50	0.44	0.38	0.34
7	1.00	0.92	0.82	0.72	0.64	0.56	0.50	0.45	0.40
8	1.00	0.93	0.85	0.76	0.69	0.62	0.55	0.50	0.45
9	1.00	0.95	0.87	0.80	0.73	0.66	0.60	0.55	0.50

Table 4.2: Proportion of wins for random play on the liberties when playing outside the Semeai 80% of the time

of 1 of winning when the opponent has only one liberty in order to model the rule of capturing a string in atari as usually used in Monte-Carlo Go programs. The results for random play on the liberties are given in table 4.2.

In these two tables, when the strings have six liberties or more, the values for lost positions are close to the values for won positions, so MCTS is not well guided by the mean of the playouts.

4.5 Experimental Results

In order to apply the score bounded MCTS algorithm, we have chosen games that can often finish as draws. Such two games are playing a Seki in the game of Go and Connect Four. The first subsection details the application to Seki, the second subsection is about Connect Four.

Seki problems

We have tested Monte-Carlo with bounds on Seki problems since there are three possible exact values for a Seki: Won, Lost or Draw. Monte-Carlo with bounds can only cut nodes when there are exact values, and if the values are only Won and Lost the nodes are directly cut without any need for bounds.

Solving Seki problems has been addressed in [NKM06]. We use more simple and easy to define problems than in [NKM06]. Our aim is to show that Monte-Carlo with bounds can improve on Monte-Carlo without bounds as used in [WBS08].

We used Seki problems with liberties for the players ranging from one to six liberties. The number of shared liberties is always two. The *Max* player (usually Black) plays first. The figure 4.4 shows the problem that has three liberties for *Max* (Black), four liberties for *Min* (White) and two shared liberties. The other problems of the test suite are very similar except for the number of liberties of Black and White. The results of these Seki problems are given in table 4.3. We can see that when *Max* has the same number of liberties than *Min* or one liberty less, the result is Draw.

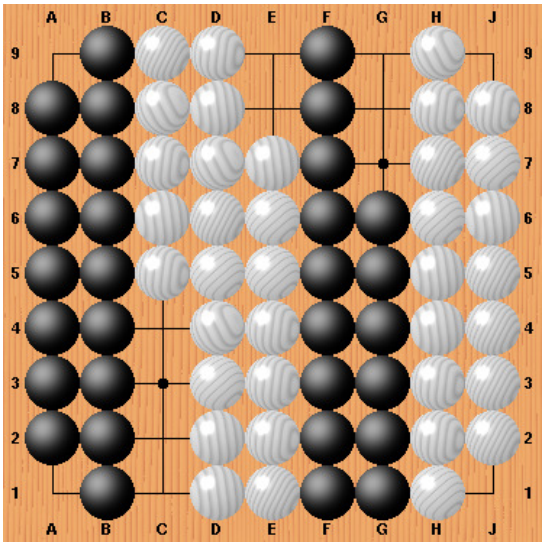


Figure 4.4: A test seki with two shared liberties, three liberties for the *Max* player (Black) and four liberties for the *Min* player (White).

Min liberties	*Max* liberties					
	1	2	3	4	5	6
1	Draw	Won	Won	Won	Won	Won
2	Draw	Draw	Won	Won	Won	Won
3	Lost	Draw	Draw	Won	Won	Won
4	Lost	Lost	Draw	Draw	Won	Won
5	Lost	Lost	Lost	Draw	Draw	Won
6	Lost	Lost	Lost	Lost	Draw	Draw

Table 4.3: Results for Sekis with two shared liberties

Min liberties	*Max* liberties					
	1	2	3	4	5	6
1	359	479	1 535	2 059	10 566	25 670
2	1 389	11 047	12 627	68 718	98 155	289 324
3	7 219	60 755	541 065	283 782	516 514	791 945
4	41 385	422 975	$>10^6$	$>10^6$	>989 407	>999 395
5	275 670	$>10^6$	$>10^6$	$>10^6$	$>10^6$	$>10^6$
6	$>10^6$	$>10^6$	$>10^6$	$>10^6$	$>10^6$	$>10^6$

Table 4.4: Number of playouts for solving Sekis with two shared liberties

The first algorithm we have tested is simply to use a solver that cuts nodes when a child is won for the color to play as in [WBS08]. The search was limited to 1 000 000 playouts. Each problem is solved thirty times and the results in the tables are the average number of playouts required to solve a problem. An optimized Monte-Carlo tree search algorithm using the Rave heuristic is used. The results are given in table 4.4. The result corresponding to the problem of figure 4.4 is at row labeled *4 min lib* and at column labeled *3 max lib*, it is not solved in 1 000 000 playouts.

The next algorithm uses bounds on score, node pruning and no bias on move selection (i.e. $\gamma = 0$ and $\delta = 0$). Its results are given in table 4.5. Table 4.5 shows that Monte-Carlo with bounds and node pruning works better than a Monte-Carlo solver without bounds.

Comparing table 4.5 to table 4.4 we can also observe that Monte-Carlo with bounds and node pruning is up to five time faster than a simple Monte-Carlo solver. The problem with three *Min* liberties and three *Max* liberties is solved in 107,353 playouts when it is solved in 541,065 playouts by a plain Monte-Carlo solver.

The third algorithm uses bounds on score, node pruning and biases move selection with $\delta = 10000$. The results are given in table 4.6. We can see in

Min liberties	*Max* liberties					
	1	2	3	4	5	6
1	192	421	864	2 000	4 605	14 521
2	786	3 665	3 427	17 902	40 364	116 749
3	4 232	22 021	107 353	94 844	263 485	588 912
4	21 581	177 693	>964 871	$>10^6$	878 072	$>10^6$
5	125 793	$>10^6$	$>10^6$	$>10^6$	$>10^6$	$>10^6$
6	825 760	$>10^6$	$>10^6$	$>10^6$	$>10^6$	$>10^6$

Table 4.5: Number of playouts for solving Sekis with two shared liberties, bounds on score, node pruning, no bias

Min liberties	*Max* liberties					
	1	2	3	4	5	6
1	137	259	391	1 135	2 808	7 164
2	501	1 098	1 525	3 284	13 034	29 182
3	1 026	5 118	9 208	19 523	31 584	141 440
4	2 269	10 094	58 397	102 314	224 109	412 043
5	6 907	27 947	127 588	737 774	>999 587	$>10^6$
6	16 461	85 542	372 366	$>10^6$	$>10^6$	$>10^6$

Table 4.6: Number of playouts for solving Sekis with two shared liberties, bounds on score, node pruning, biasing with $\gamma = 0$ and $\delta = 10000$

this table that the number of playouts is divided by up to ten. For example the problem with three *Max* lib and three *Min* lib is now solved in 9,208 playouts (it was 107,353 playouts without biasing move selection and 541,065 playouts without bounds). We can see that eight more problems can be solved within the 1,000,000 playouts limit.

Connect Four

Connect Four was solved for the standard size 7×6 by L. V. Allis in 1988 [All88]. We tested a plain MCTS Solver as described in [WBS08] (plain), a score bounded MCTS with alpha-beta style cuts but no selection guidance that is with $\gamma = 0$ and $\delta = 0$ (cuts) and a score bounded MCTS with cuts and selection guidance with $\gamma = 0$ and $\delta = -0.1$ (guided cuts). We tried multiple values for γ and δ and we observed that the value of γ does not matter much and that the best value for δ was consistently $\delta = -0.1$. We solved various small sizes of Connect Four. We recorded the average over thirty runs of the number of playouts needed to solve each size. The results are given in table

	Size			
	3×3	3×4	4×3	4×4
plain MCTS Solver	2 700.9	26 042.7	227 617.6	$> 5 \times 10^6$
MCTS Solver with cuts	2 529.2	12 496.7	31 772.9	386 324.3
MCTS Solver with guided cuts	1 607.1	9 792.7	24 340.2	351 320.3

Table 4.7: Comparison of solvers for various sizes of Connect Four

4.7.

Concerning 7×6 Connect Four we did a 200 games match between a Monte-Carlo with alpha-beta style cuts on bounds and a Monte-Carlo without it. Each program played 10 000 playouts before choosing each move. The result was that the program with cuts scored 114.5 out of 200 against the program without cuts (a win scores 1, a draw scores 0.5 and a loss scores 0).

4.6 Conclusion and Future Works

We have presented an algorithm that takes into account bounds on the possible values of a node to select nodes to explore in a MCTS solver. For games that have more than two outcomes, the algorithm improves significantly on a MCTS solver that does not use bounds.

In our solver we avoided solved nodes during the descent of the MCTS tree. As [WBS08] points out, it may be problematic for a heuristic program to avoid solved nodes as it can lead MCTS to overestimate a node.

It could be interesting to make γ and δ vary with the number of playout of a node as in RAVE. We may also investigate alternative ways to let score bounds influence the child selection process, possibly taking into account the bounds of the father.

We currently backpropagate the real score of a playout, it could be interesting to adjust the propagated score to keep it consistent with the bounds of each node during the backpropagation.

5 Transpositions in MCTS

Contents

The following section draws heavily from [SCM10].

5.1 Introduction

MCTS is a very successful algorithm for multiple complete information games such as Go [Cou06, Cou07, GS08, CCF+09] or Hex [CS09]. Monte-Carlo programs usually deal with transpositions the *simple way*: they do not modify the UCT formula and develop a DAG instead of a tree.

Transpositions are widely used in combination with the Alpha-Beta algorithm [Bre98] and they are a crucial optimization for games such as Chess. Transpositions are also used in combination with the MCTS algorithm but little work has been done to improve their use or even to show they are useful. The only works we are aware of are the paper by Childs and Kocsis [CBK08] and the paper by Méhat and Cazenave [MC10].

We will use the following notations for a given object x. If x is a node, then $c(x)$ is the set of the edges going out of x, similarly if x is an edge and y is its destination, then $c(x) = c(y)$ is the set of the edges going out y. We indulge in saying that $c(x)$ is the set of children of x even when x is an edge. If x is an edge and y is its origin, then $b(x) = c(y)$ is the set of edges going out of

y. $b(x)$ is the set of the "siblings" of x plus x. During the backpropagation step, payoffs are cumulatively attached to nodes or edges. We denote by $\mu(x)$ the mean of payoffs attached to x (be it an edge or a node), and by $n(x)$ the number of payoffs attached to x. If x is an edge and y is its origin, we denote by $p(x)$ the total number of payoffs the children of y have received: $p(x) = \sum_{e \in c(y)} n(e) = \sum_{e \in b(x)} n(e)$. Let x be a node or an edge, between the apparition of x in the tree and the first apparition of a child of x, some payoffs (usually one) are attached to x, we denote the mean (resp. the number) of such payoffs by $\mu'(x)$ (resp. $n'(x)$). We denote by $\pi(x)$ the best move in x according to a context dependant policy.

Before having a look at transpositions in the MCTS framework, we first use the notation to express a few remarks on the plain UCT algorithm (when there is no transpositions). The following equalities are either part of the definition of the UCT algorithm or can easily be deduced. The payoffs available at a node or an edge x are exactly those available at the children of x and those that were obtained before the creation of the first child: $n(x) = n'(x) + \sum_{e \in c(x)} n(e)$. The mean of a move is equal to the weighted mean of the means of the children moves and the payoffs carried before creation of the first child:

$$\mu(x) = \frac{\mu'(x) \times n'(x) + \sum_{e \in c(x)} \mu(e) \times n(e)}{n' + \sum_{e \in c(x)} n(e)} \tag{5.1}$$

The plain UCT value [KS06] with an exploration constant c giving the score of a node x is written

$$u(x) = \mu(x) + c \times \sqrt{\frac{\log p(x)}{n(x)}} \tag{5.2}$$

The plain UCT policy consists in selecting the move with the highest UCT formula: $\pi(x) = \max_{e \in c(x)} u(e)$. When enough simulations are run at x, the mean of x and the mean of the best child of x are converging towards the same value [KS06]:

$$\lim_{n(x) \to \infty} \mu(x) = \lim_{n(x) \to \infty} \mu(\pi(x)) \tag{5.3}$$

Our main contribution consists in providing a parametric formula adapted from the UCT formula 5.2 so that some transpositions are taken into account. Our framework encompasses the the work presented in [CBK08]. We show that the simple way is often surpassed by other parameter settings on an artificial one player game as well as on the two player game Hex. We do not have a definitive explanation on how parameters influence the playing strength yet. We show that storing aggregations of the payoffs on the edge rather than on the nodes is preferable from a conceptual point of view and our experiment show that it also often lead to better results.

The rest of this article is organized as follows. We first recall the most common way of handling transpositions in the MCTS context. We study the possible adaptation of the backpropagation mechanism to DAG game

trees. We present a parametric framework to define an adapted score and an adapted exploration factor of a move in the game tree. We then show that our framework is general enough to encompass the existing tools for transpositions in MCTS. Finally, experimental results on an artificial single player game and on the two players game Hex are presented.

5.2 Motivation

Introducing transpositions in MCTS is challenging for several reasons. First, equation 5.1 may not hold anymore since the children moves might be simulated through other paths. Second, UCT is based on the principle that the best moves will be chosen more than the other moves and consequently the mean of a node will converge towards the mean of its best child ; having equation 5.1 holding is not sufficient as demonstrated by figure 5.2 where equation 5.3 is not satisfied.

The most common way to deal with transpositions in the MCTS framework, beside ignoring them completely, is what will be referred to in this article as the *simple way*. Each position encountered during the descent corresponds to a unique node. The nodes are stored in hash-table with the key being the hash value of the corresponding position. Mean payoff and number of simulations that traversed a node during the descent are stored in that node. The plain UCT policy is used to select nodes.

The simple way shares more information than ignoring transpositions. Indeed, the score of every playout generated after a given position a is cumulated in the node representing a. To the contrary, playouts generated after a when transpositions not detected are divided among all represents of a in the tree depending on the moves that preceded them.

It is desirable to maximize the usage of a given amount of information because it allows to make better informed decisions. In the MCTS context, information is in the form of playouts. If a playout is to be maximally used, it may be necessary to have its payoff available outside of the path it took in the game tree. For instance in figure 5.3 the information provided by the playouts were only propagated on the edges of the path they took. There is not enough information directly available at a even though a sufficient number of playouts has been run to assert that b is a better position than c.

Nevertheless, it is not trivial to share the maximum amount of information. A simple idea is to keep the DAG structure of the underlying graph and to directly propagate the outcome of a playout on every possible ancestor path. It is not always a good idea to do so in a UCT setting, as demonstrated by the counter-example 5.2. We will further study this idea under the name *update-all* in section 5.3.

5.3 Possible Adaptations of UCT to Transpositions

The first requirement of using transpositions is to keep the DAG structure of the partial game tree. The partial game tree is composed of nodes and edges, since we are not concerned with memory issues in this first approach, it is safe to assume that it is easy to access the outgoing edges as well as the in edges of given nodes. When a transposition occurs, the subtree of the involved node is not duplicated. Since we keep the game structure, each possible position corresponds to at most one node in the DAG and each node in the DAG corresponds to exactly one possible position in the game. We will indulge ourselves to identify a node and the corresponding position. We will also continue to call the graph made by the nodes and the moves *game tree* even though it is now a DAG.

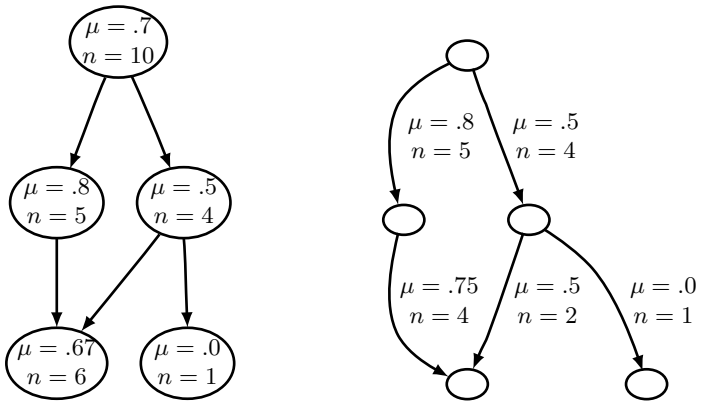
Storing results in the edges rather than in the nodes

In order to descend the game tree, one has to select moves from the root position until reaching an end of the game-tree. The selection uses the results of the previous playouts which need to be attached to moves. A move corresponds exactly to an edge of the game tree, however it is also possible to attach the results to nodes of the game tree. When the game tree is a tree, there is a one to one correspondence between edges and nodes, save for the root node. To each node but the root, correspond a unique parent edge and each edge has of course a unique destination. It is therefore equivalent to attach information to an edge (a, b) or to the destination b of that edge. MCTS implementations seem to prefer attaching information to nodes rather than to edges for implementation simplicity reasons. When the game tree is a DAG, we do not have this one to one correspondence so there may be a difference between attaching information to nodes or to edges.

In the following we will assume that aggregations of the payoffs are attached to the edges of the DAG rather than to the nodes (5.1 shows the two possibilities for a toy tree). The payoffs of a node a can still be accessed by aggregating[1] the payoffs of the edges arriving in a. No edge arrives at the root node but the results at the root node are usually not needed. On the other hand, the payoffs of an edge cannot be easily obtained from the payoffs of its starting node and its ending node, therefore storing the results in the edges is more general than storing the results only in the nodes[2].

[1] The particular aggregation depends on the backpropagation method used (see section 5.3): in the update-all case, the data of a node is equivalent to the data of the edge with the biggest number of playouts.

[2] As an implementation note, it is possible to store the aggregations of the edges in the start node provided one associates the relevant move.



(a) Storing the results in the nodes

(b) Storing the results in the edges

Figure 5.1: Example of the update-descent backpropagation results stored on nodes and on edges for a toy tree.

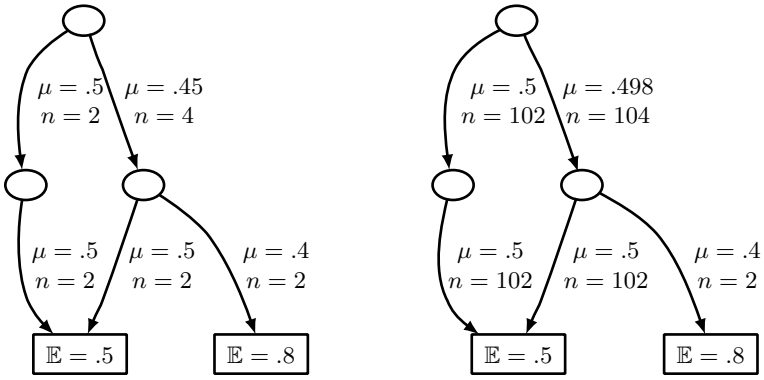
Backpropagation

After the tree was descended and a simulation lead to a payoff, information has to be propagated upwards. When the game tree is a plain tree, the propagation is straightforward. The traversed nodes are exactly the ancestors of the leaf node from which the simulation was performed. The edges to be updated are thus easily accessed and for each edge, one simulation is added to the counter and the total score is updated. Similarly, in the hash-table solution, the traversed edges are stored on a stack and they are updated the same way.

In the general DAG problem however, many distinct algorithms are possible. The ancestor edges are a superset of the traversed edges and it is not clear which need to be updated and if and how the aggregation should be adapted. We will be concerned with three possible ways to deal with the update step: updating every ancestor edge, updating the descent path, updating the ancestor edges but modifying the aggregation of the edge not belonging to the descent path.

Updating every ancestor edge without modifying the aggregation is simple enough, provided one takes care that each edge is not updated more than once after each playout. We call this method *update-all*. Update-all might suffer from deficiencies in schemata like the counter-example presented in figure 5.2. The problem in update-all made obvious by this counter-example is that the distribution of playouts in the different available branches does not correspond to a distribution as given by UCT: assumption 5.3 is not satisfied.

The other straightforward method is to update only the traversed edges, we call it *update-descent*. This method is very similar to the standard UCT algorithm implemented on a regular tree and it is used in the simple way. When such a backpropagation is selected, the selection mechanism can be



(a) Initial settings (b) 100 playouts later

Figure 5.2: Counter-example for the update-all backpropagation procedure. If the initial estimation of the edges is imperfect, the UCT policy combined with the update-all backpropagation procedure is likely to lead to errors

adjusted so that transpositions are taken into account when evaluating a move. The possibilities for the selection mechanism are presented in the following section.

The backpropagation procedure advocated in [CBK08] for their selection procedure UCT3 is also noteworthy. We did not implement it because the same behaviour could be obtained directly with update-descent backpropagation (see section 5.3).

Selection

The descent of the game tree can be described as follows. Start from the root node. When in a node a, select a move m available in a using a selection procedure. If m corresponds to an edge in the game tree, move along that edge to another node of the tree and repeat. If m does not correspond to an edge in the tree, consider the position b resulting from playing m in a. It is possible that b was already encountered and there is a node representing b in the tree, in this case, we have just discovered a transposition, build an edge from a to b, move along that edge and repeat the procedure from b. Otherwise construct a new node corresponding to b and create an edge between a and b, the descent is finished.

The selection process consists in selecting a move that maximizes a given formula. State of the art implementations usually rely on complex formulae that embed heuristics or domain specific knowledge, but the baseline remains the UCT formula[3] defined in equation 5.2.

[3]Although these heuristics tend to make the exploration term unnecessary.

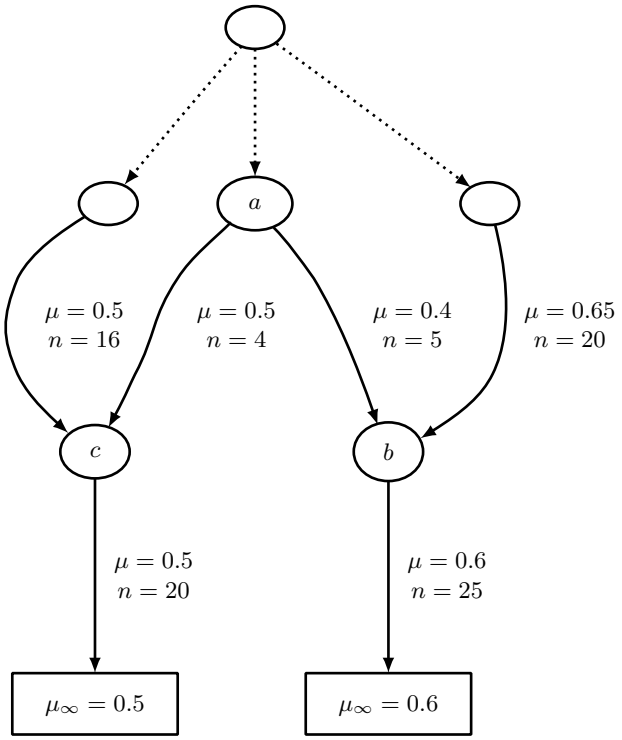


Figure 5.3: There is enough information in the game tree to know that position b is better than position c, but there is not enough local information at node a to make the right decision.

When the game tree is a DAG and we use the update-descent backpropagation method, the equation 5.1 does not hold anymore, so it is not absurd to look for another way of estimating the value of a move than the UCT value. Simply put, equation 5.1 says that all the needed information is available locally, however deep transpositions can provide useful information that would not be accessible locally.

For instance in the partial game tree in figure 5.3, it is desirable to use the information provided by the transpositions in node b and c in order to make the right choice at node a. The local information in a is not enough to decide confidently between b and c, but if we have a look at the outgoing edges of b and c then we will have more information. This example could be adapted so that we would need to look arbitrarily deep to get enough information.

We define a parametric *adapted score* to try to take advantage of the transpositions to gain further insight in the intrinsic value of the move. The adapted score is parameterized by a depth d and is written for an edge e $\mu_d(e)$. $\mu_d(e)$ uses the number of playouts, the mean payoff and the adapted score of the descendants up to depth d. The adapted score is given by the following recursive

formula.

$$\begin{aligned}\mu_0(e) &= \mu(e)\\ \mu_d(e) &= \frac{\sum_{f\in c(e)} \mu_{d-1}(f) \times n(f)}{\sum_{f\in c(e)} n(f)}\end{aligned}$$

The UCT algorithm uses an exploration factor to balance concentration on promising moves and exploration of less known paths. The exploration factor of an edge tries to quantify the information directly available at it. It does not allow to acknowledge that transpositions occurring after the edge offer additional information to evaluate the quality of a move. So just as we did above with the adapted score, we define a parametric *adapted exploration factor* to replace the exploration factor. Specifically, for an edge e, we define a parametric *move exploration* that accounts for the adaptation of the number of payoffs available at edge e and is written $n_d(e)$ and a parametric *origin eploration* that accounts for the adaptation of the total number of payoffs at the origin of e and is written $p_d(e)$. The parameter d also refers to a depth. $n_d(e)$ and $p_d(e)$ are defined by the following formulae.

$$\begin{aligned}n_0(e) &= n(e)\\ n_d(e) &= \sum_{f\in c(e)} n_{d-1}(f)\\ p_d(e) &= \sum_{f\in b(e)} n_d(f)\end{aligned}$$

In the MCTS algorithm, the tree is built progressively as the simulations are run. So any aggregation of edges built after edge e will lack the information available in $\mu'(e)$ and $n'(e)$. This can lead to a leak of information that becomes more serious as the depth d grows. If we attach $\mu'(e)$ and $n'(e)$ along $\mu(e)$ and $n(e)$ to an edge it is possible to avoid the leak of information and to slightly adapt the above formulae to also take advantage of this information. Another advantage of the following formulation is that is avoids to treat separately edges without any child.

$$\begin{aligned}\mu_0(e) &= \mu(e)\\ \mu_d(e) &= \frac{\mu'(e) \times n'(e) + \sum_{f\in c(e)} \mu_{d-1}(f) \times n(f)}{n'(e) + \sum_{f\in c(e)} n(f)}\\ n_0(e) &= n(e)\\ n_d(e) &= n'(e) + \sum_{f\in c(e)} n_{d-1}(f)\\ p_d(e) &= \sum_{f\in b(e)} n_d(f)\end{aligned}$$

If the height of the partial game tree is bounded by h[4], then there is no difference between $d_i = h$ and $d_i = h + x$ for $i \in \{1, 2, 3\}$ and $x \in \mathbb{N}$. When d_i is chosen sufficiently big, we write $d_i = \infty$ to avoid the need to specify any bound. Since the underlying graph of the game tree is acyclic, if h is a bound on the height of an edge e then $h - 1$ is a bound on the height of any child of e, therefore we can write the following equality which recalls equation 5.1.

$$\mu_\infty(e) = \frac{\mu'(e) \times n'(e) + \sum_{f \in c(e)} \mu_\infty(f) \times n(f)}{n'(e) + \sum_{f \in c(e)} n(f)}$$

The formulae proposed do not ensure that any playout will not account for more than once in the values of $n_d(e)$ and $p_d(e)$. However a playout can only be counted multiple times if there are transpositions in the subtree starting after e. It is not clear to the authors how a transposition in the subtree of e should affect the confidence in the adapted score of e. Thus, it is not clear whether such playouts need to be accounted several times or just once. Admitting several accounts gives rise to a simpler formula and was chosen for this reason.

We can now adapt formula 5.2 to use the adapted score and the adapted exploration to give a value to a move. We define the adapted value of an edge e with parameters $(d_1, d_2, d_3) \in \mathbb{N}^3$ and exploration constant c to be $u_{d_1,d_2,d_3}(e) = \mu_{d_1}(e) + c \times \sqrt{\frac{\log p_{d_2}(e)}{n_{d_3}(e)}}$. The notation (d_1, d_2, d_3) makes it easy to express a few remarks about the framework.

- When no transposition occur in the game, such as when the board state includes the move list, every parameterization gives rise to exactly the same selection behavior which is also that of the plain UCT algorithm.
- The parameterization $(0, 0, 0)$ is not the same as completely ignoring transpositions since each position in the game appears only once in the game tree when we use parameterization $(0, 0, 0)$.
- The simple way (see section 5.2) can be obtained through the $(1, 1, 1)$ parameterization.
- The selection rules in [CBK08] can be obtained through our formalism: UCT1 corresponds to parameterization $(0, 0, 0)$, UCT2 is $(1, 0, 0)$ and UCT3 is $(\infty, 0, 0)$.
- It is possible to adapt the UCT value in almost the same way when the results are stored in the nodes rather than in the edges but it would not be possible to have a parameterization similar to any of d_1, d_2 or d_3 equaling to zero.

[4]for instance if the game cannot last more than h move or if one node is created after each playout and there will not be more than h playouts

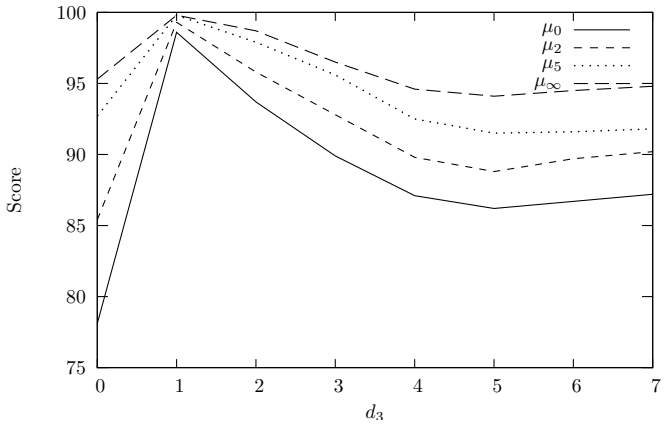


Figure 5.4: LeftRight results.

5.4 Experimental results

Tests on LeftRight

LeftRight is an artificial one player game already used in [Caz09] under the name "left move", at each step the player is asked to chose to move Left or to move Right ; after a given number of steps the score of the player is the number of steps walked towards Left. A position is uniquely determined by the number of steps made towards Left and the total number of moves played so far, transitions are therefore very frequent[5].

We used 300 moves long games for our tests. Each test was run 200 times and the standard error is never over 0.3% on the following scores.

The UCT algorithm performs well at LeftRight so the number of simulations had to be low enough to get any differentiating result. We decided to run 100 playouts per move. The plain UCT algorithm without detection of transpositions with an exploration constant of 0.3 performs 81.5 %, that is in average 243.5 moves out of 300 were Left. We also tested the update-all backpropagation algorithm which scored 77.7 %. We tested different values for all three parameters but the scores almost did not evolve with d_2 so for the sake of clarity we present results with d_2 set to 0 in figure 5.4.

The best score was 99.8% with the parameterization $(\infty, 0, 1)$ which basically means that in average less than one move was played to the Right in each game. Setting d_3 to 1 generally constituted a huge improvement. Raising d_1 was consistently improving the score obtained, eventually culminating with $d_1 = \infty$.

[5] if there are h steps the full game tree has only $\frac{h \times (h-1)}{2}$ nodes if transpositions are recognized but 2^h nodes otherwise

Tests on Hex

Hex is two-player zero sum game that cannot end in a draw. Every game will end after at most a certain number of moves and can be labeled as a win for Black or as a win for White. Rules and details about Hex can be found in [Bro00]. Various board sizes are possible, sizes from 1 to 8 have been computer solved [HAH09]. Transpositions happen frequently in Hex because a position is completely defined by the sets of moves each player played, the particular order that occurred before has no influence on the position. MCTS is quite successful in Hex [CS09], hence Hex can serve as a good experimentation ground to test our parametric algorithms.

Hex offers a strong advantage to the first player and it is common practice to balance a game with a compulsory mediocre first move[6]. We used a size 5 board with an initial stone on *b2*. Each test was a 400 games match between the parameterization to be tested and a standard AI In each test, the standard AI played Black on 200 games and White on the remaining 200 games. The reported score designates the average number of games won by a parameterization. The standard error was never over 2.5%.

The standard AI used the plain UCT algorithm with an exploration constant of 0.3, it did not detect transpositions and it could perform 1 000 playouts at each move. We also ran a similar 400 games match between the standard AI and an implementation of the update-all backpropagation algorithm with an exploration constant of 0.3 and 1 000 playouts per move. The update-all algorithm scored 51.5% which means that it won 206 games out of 400. The parameterization to be tested also used a 0.3 exploration constant and 1 000 playouts at each move. The results are presented in figure 5.5 for d_2 set to 0 and in figure 5.6 for d_2 set to 1.

The best score was 63.5% with the parameterization $(0, 1, 2)$. It seems that setting d_1 as low as possible might improve the results, indeed with $d_1 = 0$ the scores were consistently over 53% while having $d_1 = 1$ led to having scores between 48% and 62%. Setting $d_1 = 0$ is only possible when the payoffs are stored per edge instead of per node as discussed in section 5.3.

5.5 Conclusion and Future Work

We have presented a parametric algorithm to deal with transpositions in MCTS. Different parameters did improve on usual MCTS algorithms for two games: LeftRight and Hex.

In this paper we did not deal with the graph history interaction problem [KM04]. In some games the problem occurs and we might adapt the MCTS algorithm to deal with it.

[6] Even more common is the swap rule or pie-rule.

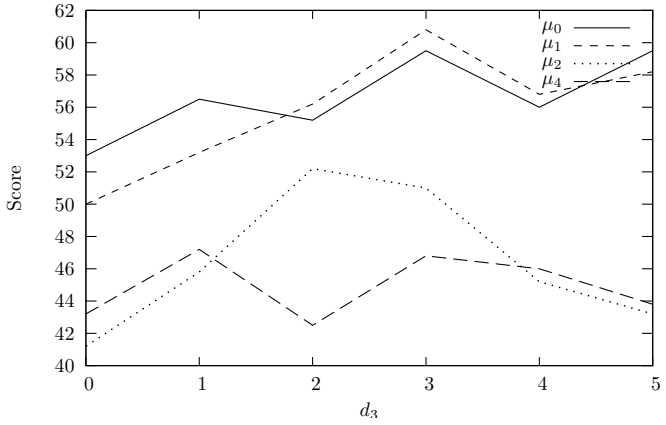


Figure 5.5: Hex results with d_2 set to 0

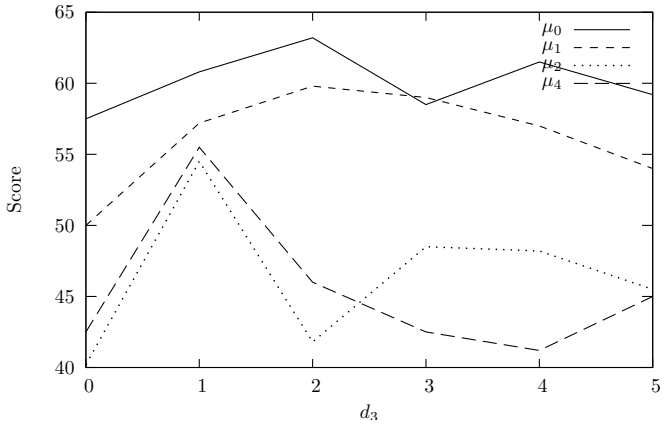


Figure 5.6: Hex results with d_2 set to 1

We have defined a parameterized value for moves that integrates the information provided by some relevant transpositions. The distributions of the values for the available moves at some nodes do not necessarily correspond to a UCT distribution. An interesting continuation of our work would be to define an alternative parametric adapted score so that the arising distributions would still correspond to UCT distributions.

Another possibility to take into account the information provided by the transpositions is to treat them as contextual side information. This information can be integrated in the value using the RAVE formula [GS07], or to use the episode context framework described in [Ros10].

Bibliography

[ACBF02] Peter Auer, Nicoló Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine learning*, 47(2):235–256, 2002.

[All88] Louis Victor Allis. A knowledge-based approach of connect-four the game is solved: White wins. Masters thesis, Vrije Universitat Amsterdam, Amsterdam, The Netherlands, October 1988.

[All94] Louis Victor Allis. *Searching for Solutions in Games an Artificial Intelligence*. Phd thesis, Vrije Universitat Amsterdam, Maastricht, 1994.

[Ber79] Hans J. Berliner. The B^* tree search algorithm: A best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[Bou01] Charles L. Bouton. Nim, a game with a complete mathematical theory. *Annals of Mathematics*, 3(1):35–39, 1901.

[Bre98] Dennis Michel Breuker. *Memory versus Search in Games*. Phd thesis, Universiteit Maastricht, 1998.

[Bro00] Cameron Browne. *Hex Strategy: Making the Right Connections*. Natick, MA, 2000.

[Caz06] Tristan Cazenave. A Phantom-Go program. In *Advances in Computer Games 2005*, volume 4250 of *Lecture Notes in Computer Science*, pages 120–125. Springer, 2006.

[Caz07] Tristan Cazenave. Reflexive monte-carlo search. In *Computer Games Workshop*, pages 165–173, Amsterdam, The Netherlands, 2007.

[Caz09] Tristan Cazenave. Nested monte-carlo search. In *IJCAI*, pages 456–461, 2009.

[CBK08] Benjamen E. Childs, James H. Brodeur, and Levente Kocsis. Transpositions and move groups in Monte Carlo Tree Search. In *CIG-08*, pages 389–395, 2008.

[CCF+09] Guillaume Chaslot, Louis Chatriot, C. Fiter, Sylvain Gelly, Jean-Baptiste Hoock, Julien Perez, Arpad Rimmel, and Olivier Teytaud. Combiner connaissances expertes, hors-ligne, transientes et en ligne pour l'exploration Monte-Carlo. Apprentissage et MC. *Revue d'Intelligence Artificielle*, 23(2-3):203–220, 2009.

[Clu07] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139. AAAI Press, 2007.

[Cou06] Rémi Coulom. Efficient selectivity and back-up operators in monte-carlo tree search. In *Computers and Games 2006*, Volume 4630 of LNCS, pages 72–83, Torino, Italy, 2006. Springer.

[Cou07] Rémi Coulom. Computing Elo ratings of move patterns in the game of Go. *ICGA Journal*, 30(4):198–208, December 2007.

[CS09] Tristan Cazenave and Abdallah Saffidine. Utilisation de la recherche arborescente Monte-Carlo au Hex. *Revue d'Intelligence Artificielle*, 23(2-3):183–202, 2009.

[CS10] Tristan Cazenave and Abdallah Saffidine. Score bounded Monte-Carlo tree search. In *Computer and Games*, 2010.

[FB08] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.

[GL05] Michael Genesereth and Nathaniel Love. General game playing: Overview of the aaai competition. *AI Magazine*, 26:62–72, 2005.

[GS07] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *ICML*, pages 273–280, 2007.

[GS08] Sylvain Gelly and David Silver. Achieving master level play in 9 x 9 computer go. In *AAAI*, pages 1537–1540, 2008.

[GST09] Martin Günther, Stephan Schiffel, and Michael Thielscher. Factoring general games. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 27–34, 2009.

[HAH09] Philip Henderson, Broderick Arneson, and Ryan B. Hayward. Solving 8x8 Hex. In Craig Boutilier, editor, *IJCAI*, pages 505–510, 2009.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. Syst. Sci. Cybernet.*, 4(2):100–107, 1968.

[HPW09] David P. Helmbold and Aleatha Parker-Wood. All-moves-as-first heuristics in monte-carlo go. In Olivas Arabnia, de la Fuente, editor, *Proceedings of the 2009 International Conference on Artificial Intelligence*, pages 605–610, 2009.

[Hsu02] Feng-hsiung Hsu. *Behind Deep Blue: Building the computer that defeated the world chess champion*. Princeton Univ Pr, 2002.

[KM04] Akihiro Kishimoto and Martin Müller. A general solution to the graph history interaction problem. In *AAAI*, pages 644–649, 2004.

[Koz09] Tomáš Kozelek. Methods of MCTS and the game Arimaa. Master's thesis, Charles University in Prague, 2009.

[KS06] Levente Kocsis and Csaba Szepesvàri. Bandit based monte-carlo planning. In *ECML*, volume 4212 of *Lecture Notes in Computer Science*, pages 282–293. Springer, 2006.

[KSS91] D.B. Kemp, P.J. Stuckey, and D. Srivastava. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the 1991 Int. Symposium on Logic Programming*, pages 337–351. Citeseer, 1991.

[LDG+96] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The Objective Caml system. *Software and documentation available from http://pauillac. inria. fr/ocaml*, 1996.

[LHG06] Nathaniel C. Love, Timothy L. Hinrichs, and Michael R. Genesereth. General Game Playing: Game Description Language specification. Technical report, Stanford University, 2006.

[Lor08] Richard J. Lorentz. Amazons discover monte-carlo. In *Computers and Games*, pages 13–24, 2008.

[LS09] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6):1–38, 2009.

[Maa05] Thomas Maarup. Hex: Everything you always wanted to know about Hex but were afraid to ask. Master's thesis, Department of Mathematics and Computer Science, University of Southern Denmark, Odense, Denmark, 2005.

[MC10] Jean Méhat and Tristan Cazenave. Combining UCT and nested Monte-Carlo search for single-player general game playing. *to appear*, 2010.

[MRVP09] Frédéric De Mesmay, Arpad Rimmel, Yevgen Voronenko, and Markus Püschel. Bandit-based optimization on graphs with application to library performance tuning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 729–736. ACM, 2009.

[NKM06] Xiaozhen Niu, Akihiro Kishimoto, and Martin Müller. Recognizing seki in computer go. In *ACG*, pages 88–103, 2006.

[QC07] Michel Quenault and Tristan Cazenave. Extended general gaming model. In *Computer Games Workshop 2007*, pages 195–204, June 2007.

[RN02] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, 2 edition, December 2002.

[Ros10] Christopher D. Rosin. Multi-armed bandits with episode context. In *Proceedings ISAIM*, 2010.

[SBB+07] Jonathan Schaeffer, Neil Burch, Yngvi Björnsson, Akihiro Kishimoto, Martin Müller, Robert Lake, Paul Lu, and Steve Sutphen. Checkers is solved. *Science*, 317(5844):1518, 2007.

[Sch01] Jonathan Schaeffer. A gamut of games. *AI Magazine*, 22(3):29–46, 2001.

[SCM10] Abdallah Saffidine, Tristan Cazenave, and Jean Méhat. UCD : Upper Confidence bound for rooted Directed acyclic graphs. In *International Workshop on Computer Games*, 2010.

[She02] Brian Sheppard. World-championship-caliber scrabble. *Artificial Intelligence*, 134(1-2):241 – 275, 2002.

[ST09] David Silver and Gerald Tesauro. Monte-Carlo simulation balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 945–952. ACM, 2009.

[Stu02] Nathan Sturtevant. A comparison of algorithms for multi-player games. In *Computer and Games*, 2002.

[SWvdH+08] Maarten P. D. Schadd, Mark H. M. Winands, H. Jaap van den Herik, Guillaume Chaslot, and Jos W. H. M. Uiterwijk. Single-player monte-carlo tree search. In *Computers and Games*, pages 1–12, 2008.

[Thi09] Michael Thielscher. Answer set programming for single-player games in general game playing. In *ICLP*, pages 327–341, 2009.

[TL10] K. Tuncay Tekle and Yanhong A. Liu. Precise complexity analysis for efficient Datalog queries. *PPDP, Hagenberg, Austria*, 2010.

[Wau09] Kevin Waugh. Faster state manipulation in general games using generated code. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 2009.

[WB09] Mark H. M. Winands and Yngvi Björnsson. Evaluation function based Monte-Carlo LOA. In *Advances in Computer Games*, 2009.

[WBS08] Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In *Computers and Games*, pages 25–36, 2008.