Diss. ETH No. 14149

Divide and Conquer in Game Tree Search: Algorithms, Software and Case Studies

Dissertation

submitted to the
Swiss Federal Institute of Technology
Zurich

for the degree of
Doctor of Technical Sciences

presented by

Fabian Mäser

Dipl. Informatik-Ing. ETH
born on April 7th, 1967
citizen of Dägerlen ZH

Accepted on the recommendation of
Prof. Jürg Nievergelt, examiner
Prof. Martin Müller, coexaminer
Prof. Peter Widmayer, coexaminer

2001

Contents

List of Figures

Abstract

The micro world of games provides an ideal testing environment for search techniques and algorithms. Combinatorial Game Theory (CGT) is a promising, relatively new approach to the analysis of games. By decomposing a game into a sum of independent local games, CGT enables the application of divide and conquer to game tree search. This fundamental paradigm of algorithm design promises to lead to efficient search techniques. Whereas CGT has been subject to mathematical research for more than two decades, this thesis aims to progress in the yet little explored field of computational CGT.

In two case studies we apply the powerful methods of CGT in the field of algorithmic game theory. In the first application we investigate a class of games where a local move can lead to an overall win in a sum of games. We present a new computation model that deals with such global threats and use it to analyze Zugzwang positions in king and pawn chess endgames. In the second case study we analyze zero-sum games without Zugzwang. With applications to the game Regio, we demonstrate that the use of heuristic evaluation functions in local game search combined with algorithms for sum game play is a promising divide and conquer approach to heuristic game playing.

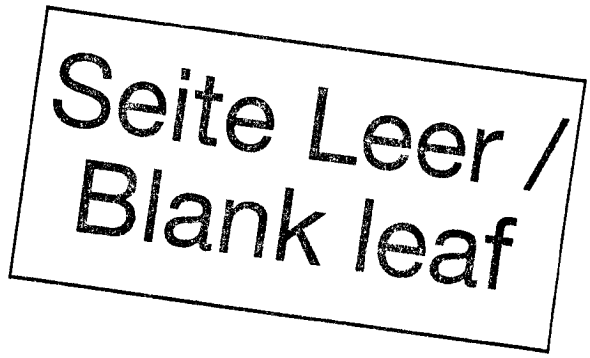
With the Game Bench we present an application framework for combinatorial game programming. Its main focus is an extensible search engine that provides game independent algorithms and data structures for combinatorial game tree search and sum game play. The choice of an object-oriented design and of the programming language Java makes the Game Bench extensible to the programmer's needs and portable to virtually any computer platform. Several applications and student programming projects have demonstrated its value as a tool for rapid prototyping of game playing programs.

Kurzfassung

Die Mini-Welt der Spiele bietet eine ideale Testumgebung für Such-Techniken und -Algorithmen. Die Kombinatorische Spieltheorie (KST) ist ein relativ neuer, vielversprechender Ansatz in der Analyse von Spielen. Indem sie ein Spiel in eine Summe von unabhängigen Teilspielen zerlegt, wendet die Kombinatorische Spieltheorie ein fundamentales Paradigma der Algorithmik in der Spielbaum-Suche an, "Teile und Herrsche". Seit über zwei Jahrzehnten ist die Kombinatorische Spieltheorie Gegenstand der mathematischen Forschung. Ziel dieser Arbeit ist es, Fortschritte zu erzielen in der algorithmischen KST, einem bisher wenig erforschten Gebiet.

In zwei Fallstudien wenden wir die Methoden der KST auf dem Gebiet der algorithmischen Spieltheorie an. Die erste untersucht eine Klasse von Spielen, in denen ein Zug in einem Teilspiel zu einem Sieg in der ganzen Summe führen kann. Wir stellen ein neues Berechnungsmodell vor, welches mit solchen globalen Drohungen umgehen kann und setzen es ein in der Analyse von Zugzwangstellungen in Bauernendspielen im Schachspiel. In der zweiten Fallstudie analysieren wir Zugzwang-freie Nullsummenspiele. Mit Anwendungen im Spiel Regio zeigen wir, dass der Einsatz von heuristischen Bewertungsfunktionen in der kombinatorischen Spielbaum-Suche in Kombination mit einem Algorithmus für das Summenspiel einen vielversprechenden Ansatz, "Teile und Herrsche" in der Spiel-Analyse anzuwenden, liefert.

Die "Game Bench" ist ein Rahmenprogramm für die Implementierung von kombinatorischen Spielen auf dem Computer. Das Kernstück der Game Bench ist eine erweiterbare "Suchmaschine", welche spielunabhängige Algorithmen und Datenstrukturen für die kombinatorische Spielbaum-Suche und das Spielen von Summen zur Verfügung stellt. Die Wahl einer objektorientierten Struktur und der Programmiersprache Java machen die Game Bench erweiterbar für die Bedürfnisse der Spiel-Programmierer und portierbar auf praktisch alle Computer-Plattformen. Den Erfolg der Game Bench als Werkzeug für die Entwicklung von Spielprogrammen dokumentieren verschiedene Anwendungen und Studenten-Projekte.


Seite Leer /
Blank leaf

Acknowledgments

I wish to sincerely thank all the people who assisted me during my work.

First of all, I would like to express my thanks to my advisor, *Jürg Nievergelt*, who made this research possible. I am especially grateful for the time he spent teaching me better ways of presenting my ideas. I am also indebted to my coexaminers *Martin Müller* and *Peter Widmayer* for taking the time to read my thesis and for their valuable suggestions and contributions.

Danilo Biella and *Marcel Schneider* relied on the Game Bench as a program basis for their diploma projects. *Werner Hartmann* allowed the Game Bench to be part of the teaching materials provided by the EducETH web server. *Mathias Schulze* used the Game Bench to analyze and solve his game "Crash!". I also wish to thank the students who used the Game Bench for their semester projects in Professor Nievergelt's game theory courses at ETH Zürich.

I have always enjoyed working in the pleasant atmosphere of our research group for which I would like to thank the former and present members: *Silvania Avelar*, *Adrian Brüngger*, *Michele De Lorenzi*, *Ralph Gasser*, *Reto Lamprecht*, *Thomas Lincke*, *Ambros Marzetta*, *Matthias Müller*, *Nora Sleumer*, *Vincent Tscherter* and *Christoph Wirth*.

Finally, I thank my friends and family for their invaluable support and motivation.

Chapter 1

Introduction

This chapter gives an overview of the contents of this thesis. After a brief motivation for studying combinatorial games, we give an introduction to *combinatorial game theory* in Section 1.2. We discuss related work and contributions in Sections 1.3 and 1.5. In Section 1.4 we introduce the main software project, the *Game Bench*. Finally, we give an overview of structure and content of this document in Section 1.6.

1.1 Why Study Combinatorial Games?

"Amusing oneself with games may sound like a frivolous occupation. But the fact is that the bulk of interesting and natural mathematical problems that are hardest in complexity classes beyond *NP* ... are two player games." (A. Fraenkel [16])

The micro world of games provides an ideal testing environment for AI techniques such as search, pattern recognition and knowledge representation. Rules of games are usually simple to describe, improvements (for instance playing strength) can be measured objectively, and human expert knowledge is available. The combination of search and heuristic evaluation as described by Shannon as early as in 1950 [48] leads to extremely strong game playing programs. Most notable are the results achieved by the programs *Deep Blue* [26] in chess and *Chinook* [46] in Checkers.

Furthermore, games provide state spaces of arbitrary sizes for developing and testing techniques of *exhaustive search*. *Nine Men's Morris* with a state space of approximately size 10^{10} is an example of a non-trivial game solved by a combination of alpha-beta search and retrograde analysis [21]. In fact, games are natural mathematical problems that lie in complexity classes beyond *NP* such as *P-Space* or even *Exptime*. Problems that do not admit

polynomial time algorithms have long been considered *intractable* from an algorithmic point of view. "But efficient algorithms can only be found for selected, relatively simple problems, whereas the world of applications is not simple. ... Whereas computer scientists feel they know everything worth knowing about sorting and binary search, we know little about how to organize large state spaces for effective exhaustive search. ... And for sharpening our tools, the well defined "micro worlds" of games and puzzles ... are well suited." [43]

1.2 Combinatorial Game Theory

Combinatorial game theory (CGT) deals with *sums* of games: a player to move in a sum chooses one of its component games and plays a move in it. One of the first results in analyzing sums of games is Bouton's solution of the game *Nim* [9] published in 1902. 74 years later, Conway presents a general theory of combinatorial games [11]. In 1982, three of the main exponents of the theory, Berlekamp, Conway and Guy publish more results and many examples in their classic *Winning Ways* [5].

As a research area of computer science, combinatorial game theory is still in its beginnings. The reason why this elegant mathematical theory enables promising new approaches to game tree search is that it suggests one of the most fundamental algorithmic paradigms, *divide and conquer.*

An example of a divide and conquer algorithm that computes minimax solutions of games that can be partitioned into independent subgames is *decomposition search* [41]. Decomposition search analyzes a game's independent components separately and combines the computed *local results* using techniques of combinatorial game theory. As a result, it is often possible to solve much larger problems than with standard full-width search.

1.3 State of the Art

Combinatorial game theory has mainly been used for pencil-and-paper analysis of the games described in *Winning Ways* [5] such as *Domineering* or *Hackenbush.* The first computer tool to support such analysis is Wolfe's *Gamesman's Toolkit* [52] which implements virtually all the material in finite combinatorial game theory. As a games calculator, it allows the user to perform algebraic manipulations on games. As a library of CGT functions, it supports the game programmer in analyzing a particular game using the computer. In Section 3.3 we discuss work in the field of combinatorial game

programming.

While many of the games described and analyzed in [5] were invented by the authors to demonstrate the analytical power of combinatorial game theory, there have been a few applications to traditional games of which we mention:

- *Go* endgames decompose into independent components. With the definition and analysis of *Mathematical Go* [6], Berlekamp and Wolfe map this classical game to a combinatorial game as defined by Conway [11]. In another application, Müller [39] introduces a heuristic sum game model for Go and solves late endgame positions exactly by means of combinatorial game theory.

- Elkies [13] shows that combinatorial game theory also applies to certain *chess endgames*, mainly positions with only king and pawns on each side. In these positions, each piece is restricted to some small area of the board in such a way that a decomposition of the game becomes possible.

1.4 The Game Bench: a Framework for Combinatorial Game Programming

The *Game Bench* is an application framework for programs that implement combinatorial games. Its main goal is to provide the game programmer with a game independent *search engine* that implements algorithms and data structures of *decomposition search*. Further, it provides game independent support for the implementation of common types of games such as games played on rectangular boards and graphs.

The Game Bench is written in Java which makes it portable to all of today's most popular computer platforms. Its variety of game independent support makes the Game Bench well suited for fast prototyping. On the other hand, it also allows serious game analysis and time critical computations of combinatorial games as the applications in Chapters 7 and 8 demonstrate.

1.5 Contributions of this Thesis

With the two case studies on chess endgames and zero-sum games, we extend the range of CGT applications in *computational game theory*. With applications to king and pawn endgames in chess we present a general computation

model that enables dealing with a class of games that contain global threats. In a sum of such games, a local move can be globally decisive. In the second case study we investigate the use of heuristic evaluation functions in local combinatorial search of zero-sum games with perfect information. Based on the same heuristic evaluation functions, decomposition search seems to be a promising alternative to standard alpha-beta search when playing games that can be partitioned into independent components.

With the Game Bench, we provide a general framework for the implementation of combinatorial game programs. Its game independent algorithms and data structures enable the game programmer to implement a specific combinatorial game much faster than if she had to start from scratch. Further, its extensible game independent search engine makes the Game Bench an ideal testbed for various sum play algorithms. The Game Bench has proven its usefulness in the field of education where it is successfully used in game theory courses at ETH Zürich and as a demonstration program for game tree search on the EducETH web server [12]. It has also served as a program basis in the diploma projects of Biella [8] and Schneider [47].

1.6 Structure of this Thesis

Chapter 2 deals with the theoretical background of this thesis. We review combinatorial game theory, as developed by Conway [11] and others, and discuss algorithms for sum game play. Chapters 3 to 5 describe the Game Bench, an application framework for combinatorial game programs. Chapter 3 discusses goals, requirements and design concepts of the Game Bench. Chapter 4 illustrates its architecture and discusses implementation issues, and Chapter 5 shows how to use the Game Bench on the basis of an example application. Chapters 6 to 8 describe the case studies on chess endgames and zero-sum games in general. Chapter 6 introduces a new computation model for local games with global threats. Chapter 7 presents an application of this model to king and pawn chess endgames involving Zugzwang. Chapter 8 discusses the application of combinatorial game theory to zero-sum games and the use of heuristic evaluation functions in local game tree search. Chapter 9 concludes this thesis with a summary and a look at directions for future research.

Chapter 2

Divide and Conquer in Game Theory

In this chapter we review *combinatorial game theory* (CGT) as a method of applying the *divide and conquer* paradigm to game tree search. In Section 2.1 we give an example of a sum game that is easily solved with divide and conquer. In Section 2.2 we present the basics of combinatorial game theory. In Section 2.3 we introduce *decomposition search*, an algorithm that computes minimax decisions with the help of CGT. Finally, in Section 2.4 we discuss the complexity of playing sums and possible applications of CGT in computer game playing.

2.1 Playing Sums of Games

A *sum* or *disjunctive compound* [11] is a natural way to play two or more games simultaneously. At his turn, each player chooses one of the component games and makes a move in it. When analyzing a sum game, we can often take advantage of its special structure. We break it into its components (*divide*) and combine the results of the subgames to compute the result of the whole sum (*conquer*). One of the earliest known examples of a sum game with a mathematical solution is *Nim*. Bouton [9] has solved it in 1902.

2.1.1 Nim - an Example of a Sum Game

Nim is played with heaps of tokens (in the game's origins coins were used). The two players alternately choose a heap and remove a number of tokens from it. It is only allowed to remove tokens from one heap, but it is possible to take a whole heap. The player who takes the last token wins, or equivalently,

a player unable to move loses. Figure 2.1 shows a Nim game with four heaps.

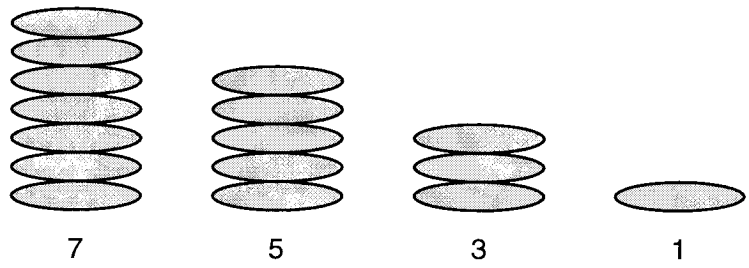


Figure 2.1: Nim, played with heaps of 7, 5, 3 and 1 tokens.

If we do not take the game's structure as a sum into account, we are able to handle only small instances of Nim. The size of its game tree grows exponentially in the number of tokens and heaps.[1] The divide and conquer approach on the other hand allows the solution of instances of virtually any size.

We consider each heap separately and define its *Nim value* as the number of tokens it contains (*divide*). To determine whether a given position is a win or a loss for the player to move (in Nim there are no draws), we perform a *Nim addition* of the Nim values of all heaps. If we get zero, the position is a loss for the player to move, if not, it is a win (*conquer*). Nim addition of two numbers is carried out as the bitwise exclusive or on their binary representation. In the example of Figure 2.1 we get

$$\mathbf{7}(111) \oplus \mathbf{5}(101) \oplus \mathbf{3}(011) \oplus \mathbf{1}(001) = \mathbf{0}(000)$$

Thus, the given position is lost for the player to move. Bouton's method is based on the facts that

1. in each position of value $v \neq 0$, it is possible to move to a position of value $v' = 0$.

2. in a position of value $v = 0$ all moves lead to positions of value $v' \neq 0$.

3. the empty position that contains no tokens is of value $v = 0$.

[1] In the special cases of one and two heaps optimal play can also be ensured by simple rules that avoid computing the game tree.

2.2 Combinatorial Game Theory

Bouton's solution of Nim was the first step towards a mathematical theory of combinatorial games. In the 1930s, Sprague and Grundy [22] developed a general theory of *impartial games*. Impartial games offer both players the same options in every position. The *Sprague-Grundy theorem* states that every impartial game is equivalent to a Nim heap of a certain size. Later, the theory was extended to *partizan games* in which the players usually have different options in a given position. Some first results were published by Milnor [36] and Hanner [25] who extended minimax calculations to sums of games. In 1976, Conway [11] introduced *combinatorial game theory* (in short *CGT*), a complete mathematical theory of combinatorial games and their sums. In the following sections, we present fundamental definitions and results of finite CGT based on Guy's article "What Is a Game?" [23] and Conway's book [11].

2.2.1 What is a Game?

Conway defines the term *game* in a slightly different way than classical game theory. According to Conway, a *combinatorial game* must satisfy the following conditions:

1. There are two players called *Left* and *Right*.
2. *Left* and *Right* move alternately in the game as a whole.
3. Under the *normal play convention* (or *normal termination rule*), the player unable to move loses.
4. The *ending condition* states that there are no draws due to (infinite) repetition of the same position. A game will always come to an end.
5. Both players have *complete information* about what is going on in the game.
6. There are no chance moves (like dealing cards or rolling dice).

Many popular games do not satisfy these conditions entirely. In card games like *Poker* or *Bridge* the players have no information about their opponents' cards. *Backgammon* has complete information, but contains chance moves. In *chess* a player unable to move does not lose the game, and in *Go* the player who controls the larger territory wins. Often, however, it is possible to map such games to equivalent combinatorial games. For example

in *Go* we suspend the capturing rule in the end and keep filling the board. The player who controls less territory will first run out of moves and lose the game.

The *ending condition* (4) makes sure there are no infinite, *loopy* games to which some of the concepts discussed below only apply in a restricted form, or not at all. Below we use the term CGT for *finite* CGT. Exceptions (Chapters 6 and 7) are explicitly pointed out.

2.2.2 The Formal Definition of a Game

A game is defined by its left and right *options*, that is the games to which *Left* and *Right* can move.

$$G = \{L_1, \ldots, L_n \mid R_1, \ldots, R_m\} \tag{2.1}$$

or in short

$$G = \{G^L \mid G^R\} \tag{2.2}$$

This definition of a game is recursive. L_i and R_j are again games while G^L and G^R stand for sets of games. The basis of recursion is the empty set. The game $\{\emptyset \mid \emptyset\} = \{\,|\,\}$ has neither any left nor any right options and is called 0 (*zero*). The player to move loses. On the first level of recursion another three games are created (see Figure 2.2):

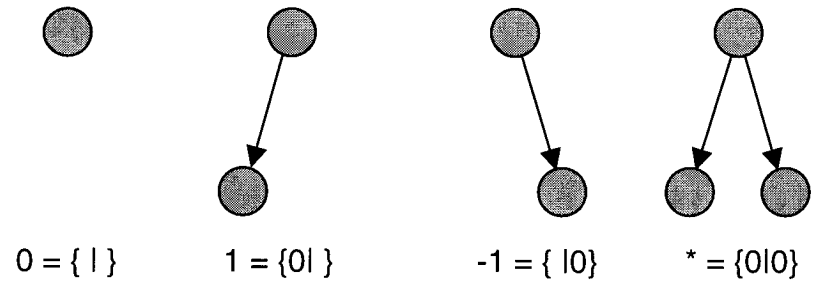


Figure 2.2: The four simplest games: 0 is the basis. 1, -1 and $*$ are created on the first level of recursion.

- The game $\{0 \mid \}$ offers *Left* a move to 0 while *Right* has no options. *Left* has one spare move, and so the game is called 1.

- In $\{ \mid 0\}$ on the other hand, *Right* can move to 0 whereas *Left* has no moves. This game is called -1.
- In $\{0 \mid 0\} = *$, the game *star*, both players have exactly one option, the move to 0. $*$ is an impartial game, equivalent to a Nim heap of size one.

2.2.3 The Four Outcome Classes

Under the *normal play convention*, every game belongs to one of four *outcome classes*:

- $\{\mathbf{G = 0}\}$ (*zero*) The *second player* wins. Games that belong to the equivalence class $\{G = 0\}$ are also known as *mutual Zugzwang*[2] (mZZ).
- $\{\mathbf{G > 0}\}$ (*positive*) *Left* wins.
- $\{\mathbf{G < 0}\}$ (*negative*) *Right* wins.
- $\{\mathbf{G \parallel 0}\}$ (*fuzzy*) The *first player* wins.

Of the four games shown in Figure 2.2, each one falls into a different outcome class. The game 0 is of course in class $\{G = 0\}$. The game 1 is won by *Left* no matter who starts. *Right* to play loses immediately as he has no move, and *Left* to play moves to 0 when again *Right* has no move. Thus, 1 belongs to the outcome class $\{G > 0\}$ and symmetrically -1 belongs to class $\{G < 0\}$. The game $*$ is a fuzzy game (we also say "is *confused* with 0"). The player who starts wins by moving to 0 when the opponent has no move.

Note that unlike in classical game theory, we examine game positions without defining which player has the move. Thus, we distinguish four outcome classes in contrast to the three (win, loss, draw) defined by Zermelo [54] although in CGT there are no draws.

2.2.4 Inverses and Sums of Games

The *inverse* or *negative* of a game $(-G)$ is constructed by "swapping sides". Its definition

$$-G = \{-G^R \mid -G^L\} \tag{2.3}$$

[2]German, meaning "a situation where one is forced to move".

where $-G^R$ stands for $\{-G^{R_1}, -G^{R_2} \dots\}$ is recursive as well, and again the empty set builds the basis of the recursion ($0 = \{\,|\,\} = -0$).

We have already defined the *sum* or *disjunctive compound* of two games G and H in Section 2.1. A player moves either in G or in H and leaves the other game unchanged. This leads to the following, again recursive, definition of the sum $G + H$ where the notation $G^L + H$ stands for the set of options $\{G^{L_1} + H, G^{L_2} + H \dots\}$:

$$G + H = \{G^L + H, G + H^L \mid G^R + H, G + H^R\} \tag{2.4}$$

With the help of the inverse of a game and the sum of two games we now define *equality* of two games: The games G and H are equal if the sum $G + (-H)$ belongs to the equivalence class $\{G = 0\}$.

$$G = H \Leftrightarrow G + (-H) = 0 \tag{2.5}$$

Sums of games are commutative and associative. With the neutral element 0 ($G + 0 = G$, $G + (-G) = 0$) the set of all games has the mathematical structure of a *group*.

Further, there is a *partial order* on games. We define $G > H$ as $G + (-H) > 0$, that is, G is greater than H if *Left* wins the sum $G + (-H)$ no matter who starts. On the other hand, if $G + (-H) \parallel 0$ we say that G and H are *incomparable* and write $G \parallel H$. In this case G is neither greater, nor equal, nor smaller than H. As one would expect $G \geq H$ is defined as $(G > H) \vee (G = H)$.

2.2.5 Composing and Simplifying Games

We create a new game G by composing a set G^L of left options and a set G^R of right options, $G = \{G^L \mid G^R\}$ as shown in Section 2.2.2. This is the way all games are created.

A set of options may contain redundant games that can either be deleted or at least replaced by simpler games. There are two ways to *simplify* a game: deleting dominated options and replacing reversible options:

1. **deleting dominated options**: if in a game $G = \{A, B, C, \cdots \mid Z, Y, X, \dots\}$, we find that $A \geq B$, then A *dominates* B and B can be omitted. Symmetrically, if $Z \leq Y$, then Z *dominates* Y and Y can be omitted.

2. **replacing reversible options**: A player's option is *reversible* if it allows the opponent to move to a game that is at least as good for her as the original game was. If a right option R in a game G contains a move R^L with $R^L \geq G$, we replace R by the list of all right options of R^L.

In the following example, we simplify the game

$$G = \{-1, * \mid 0, *\}$$

$*$ dominates -1 as a left option which we verify by checking that $* - (-1) \geq 0$. Of the two right options, neither dominates the other, as $0 \parallel *$. Thus, after *deleting dominated options*, we get

$$G = \{* \mid 0, *\}$$

Now, we find that the right option $*$ is *reversible*, as its set of left options $*^L$ contains the game 0 which is $\geq G$. (It is easy to verify that G is a win for *Right* and therefore less than zero ($G < 0$).) Thus, we replace the right option $*$ by the set of all right options of $0 = \{ \mid \}$ which is empty. Finally, we get

$$G = \{* \mid 0\}$$

It is not possible to further simplify the game $G = \{* \mid 0\}$. It is a very common game and has its own name *down*, written $\downarrow$. Analogously, the game *up* is defined as $\uparrow = \{0 \mid *\}$. Each finite combinatorial game G has a unique simplest form, the *canonical form* [11] of G.

2.2.6 Numbers

Any positive *integer* n can be interpreted as the game where *Left* can move n times whereas *Right* has no move. (We have already met the numbers 0 and 1 in Section 2.2.2.)

$$1 = \{0 \mid \} \tag{2.6}$$
$$n+1 = \{n \mid \} \tag{2.7}$$

There are also fractional numbers, such as $\frac{1}{2} = \{0 \mid 1\}$ and $\frac{1}{4} = \{0 \mid \frac{1}{2}\}$. In general, a game G in its canonical form is a number, if and only if

1. all its options G^{L_i} and G^{R_j} are numbers.

2. no left option G^{L_i} is $\geq$ any right option G^{R_j} .

The set of all numbers is a true subset of the set of all games. Looking at the four games of Figure 2.2), we already find that the game $* = \{0 \mid 0\}$ is not a number, as its left option 0 is $\geq$ its right option 0.

2.2.7 Loopy Games

In this section we briefly discuss the loopy games *on* and *off* (*Winning Ways* [5], chapter 11) which we use in Chapters 6 and 7. Loopy games are infinite games that do not meet the ending condition (see Section 2.2.1), i.e. they allow infinite sequences of moves to be played.

Two of the simplest loopy games (they consist of only one position) are the games $on = \{on \mid\}$ and $off = \{\mid off\}$ shown in Figure 2.3.

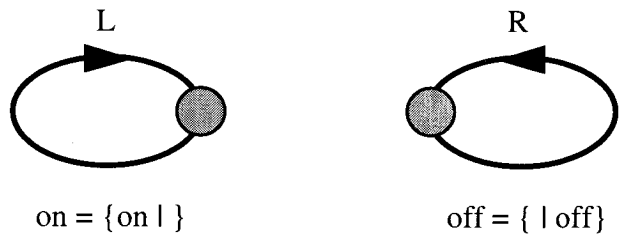


Figure 2.3: The loopy games *on* and *off* offer one of the players an infinite number of moves.

The game *on* is greater than any finite game G, as *Left* wins any sum $(on + (-G))$ by just playing in *on* until *Right* runs out of moves in $(-G)$. The game $(-G)$ allows *Right* only a finite number of consecutive moves. Analogously, any finite game G is greater than the game *off*.

In the sum $(on + off)$, both players will always have a move available, therefore no one will lose:

$$on + off = \{on + off \mid on + off\} = dud$$

The result *dud* ("deathless universal draw" [5]) offers a pass move for both players (Figure 2.4), so any sum S that contains a component of value *dud* will never be brought to an end $(dud + G = dud)$.

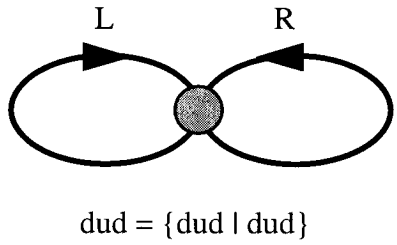


Figure 2.4: The loopy game *dud* allows both players to "pass".

2.3 Decomposition Search

Decomposition Search (Müller [41]) is an algorithm that computes minimax solutions to games that decompose into independent subgames. It uses combinatorial game theory to combine the results of locally restricted searches. This divide and conquer approach to game tree search often allows the exact solution of much larger problems than a full-width search as performed by alpha-beta and similar algorithms can handle. Decomposition search consists of four steps:

1. *Game decomposition* and subgame identification: decompose game G into its components: $G = G_1 + G_2 \cdots + G_n$.
2. *Local combinatorial game search*: for each local game produce its game graph $graph(G_i)$.
3. *Local game evaluation*: starting from the leaf nodes of $graph(G_i)$ compute the combinatorial values of all interior nodes leading to the computation of the root value $val(G_i)$.
4. *Sum game play*: make a move decision in the sum game $G = G_1 + G_2 \cdots + G_n$ based on the local combinatorial values $val(G_i)$.

In the following sections, we discuss these steps in detail and show how they are implemented.

2.3.1 Game Decomposition and Subgame Identification

There are two preconditions for applying decomposition search to a game. The game must allow a decomposition into independent components and the

subgames must meet the definition of combinatorial games given in Section 2.2.1. In some games, for example in *Nim* (Figure 2.1), the decomposition is obvious and follows directly from the rules. In board games like *Domineering* (see Chapter 5) or *Go* where the board gets filled with stones, a decomposition arises when enough moves have been played. With the help of specific knowledge of a game, we may also achieve a heuristic decomposition, for instance if we assume that in a chess position a certain piece cannot leave some area of the board (see Chapter 7).

2.3.2 Local Search and Evaluation

In contrast to minimax search, we must consider both players' moves at every node (game position). Successive moves by the same player are possible as the opponent might move in another local game. After evaluating all options $G^L = \{G^{L_1}, G^{L_2} \ldots G^{L_n}\}$ and $G^R = \{G^{R_1}, G^{R_2} \ldots G^{R_m}\}$, we compute the value of the actual node as $G = \{G^L \mid G^R\}$. The following algorithm implements recursive depth-first game tree traversal and combinatorial game evaluation of a local game.

```
function LocalSearch(): TGameValue;
/* computes the combinatorial value of a local game */
begin
  G^L ← { }; G^R ← { };
  forall left moves m do /* recursively evaluate all left moves... */
    ExecMove(m);
    G^L ← G^L ∪ LocalSearch();
    UndoLastMove();
  endfor;
  forall right moves m do /* recursively evaluate all right moves... */
    ExecMove(m);
    G^R ← G^R ∪ LocalSearch();
    UndoLastMove();
  endfor;
  return {G^L | G^R};
end LocalSearch;
```

The algorithm uses the operation of composing and simplifying a combinatorial game from two sets of left and right options. Conway [11] defines how this is done (see also Section 2.2.5), and Wolfe's *Gamesman's Toolkit*

[52] implements this and many other CGT functions. We emphasize the following points about the algorithm's implementation:

- Local search is *game independent*. The required game functionality consists only of operations to execute and take back moves and a move generator.
- As in standard minimax search, we can use *hash tables* or *transposition tables* to recognize and store previously evaluated game positions. Additionally, we can hash previously evaluated combinatorial game expressions which again improves the performance of the search. The *Gamesman's Toolkit*, for instance, does this automatically.

2.3.3 Sum Game Play

Finally, based on the evaluation of the component games, we make a move decision. Combinatorial game theory provides different ways to do this:

- The *incentive* of a move is a measure for how much this move improves the position. If there is a move whose incentive dominates all other moves' incentives, this move is proven optimal. Incentives are computed *locally*: if *Left* moves from G to G^{L_i}, the left incentive is defined as $G^{L_i} - G$, the right incentive on the other hand is defined as $G - G^{R_i}$. Unfortunately, it is not always possible to find a move whose incentive dominates all other moves' incentives. In this case, we decide on a move as shown below.
- The straightforward way to find an optimal move is to compute the sum of all local games as defined in Section 2.2.4. Although we can sometimes compute sums efficiently, this usually loses all advantages of the divide and conquer approach.
- Finally, there are heuristic local methods of finding a good move in a sum. Examples of algorithms for heuristic sum game play are *hotstrat* (see Section 8.4.5), *thermostrat* [5] and *sentestrat* [4].

2.4 Discussion

2.4.1 The Complexity of Playing Sums

Most games are "hard" in terms of complexity. The question whether player L has a win in a certain game position involves a sequence of alternating

quantifiers: "*is there* a move for player L such that on *every* move of player R there is a move for player L ... such that player L wins?" (see Figure 2.5). In contrast to difficult (*NP* complete) existential problems, the solution of a game consists of a whole subtree of the solution tree, not just of a path. Many games, in their generalized versions to arbitrary input sizes, are P-Space hard (see for example [17] for a result in generalized $n \times n$ chess).

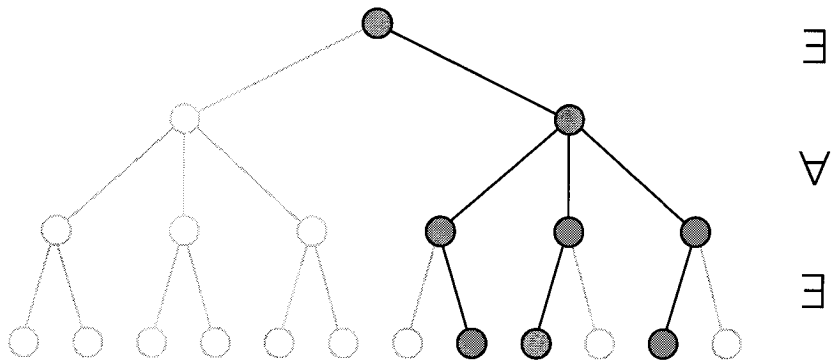


Figure 2.5: The question whether a player has a win involves a sequence of alternating quantifiers.

Can we improve on this asymptotic complexity by applying divide and conquer? In general, the answer is "no". In fact, Morris [37] proves that playing sums of even simple combinatorial games is P-Space complete as well. On the other hand, significant improvements are possible in practice. And for some games, for instance *Nim*, the divide and conquer approach even results in an efficient (polynomial time) algorithm.

The following two computation steps determine the complexity of the divide and conquer approach.

1. *Local search*: compute the values of all component games.

2. *Sum game play*: make a move decision based on the values of the local games.

In the case of *Nim* (see Section 2.1.1), we compute both steps in linear time. Minimax evaluation of the game tree of *Nim* on the other hand takes exponential time in the number of tokens.

Usually, however, local search must traverse the whole game tree of a local game to determine its value. But the branching factor (the average number of possible moves in a position) and the maximum search depth are smaller

than in the sum game. As a result, the number of nodes to be searched decreases considerably.

In the following three cases, we perform the second step of making a move decision efficiently, thus reducing the complexity of the sum game to the complexity of its biggest component game.

- We can play the sum game based on local decisions only. After pruning locally dominated moves and moves with dominated incentives, we are left with only one possible move.
- We use a heuristic sum play algorithm that makes move decisions based on local information only.
- We compute the sum of all local games efficiently. An example of a whole class of games where this is possible are *impartial games*. The value of a sum of impartial games is computed as its *Nim sum* [22].

We conclude that the divide and conquer approach often yields a considerable improvement on standard minimax evaluation.

- Although we do not improve on the asymptotic complexity in general, we often reduce the complexity of a sum to the complexity of its biggest component. For instance, Müller [41] states: "An application of decomposition search to Go has demonstrated perfect play in long endgame problems which far exceed the capabilities of conventional game tree search methods."
- An additional advantage of decomposition search is the re-usability of partial results. We can store values of local games that frequently occur in temporary hash tables or persistent databases (see also Section 4.2.2, *local hashing*).

2.4.2 Applications of CGT in Computer Game Playing

Combinatorial game theory defines games in a slightly different way than classical game theory as introduced by von Neumann and Morgenstern [50]. Many popular games do not meet the *normal termination rule* which states that a player unable to move loses. One of the main goals of this thesis is to investigate algorithms and computation models that apply the powerful methods of combinatorial game theory to computer analysis and solution of a wide range of games. In Chapter 7, we present a computation model that allows to compute values of games that contain globally winning moves. In

the second case study (Chapter 8), we use heuristic evaluation functions to compute approximate combinatorial game values of zero-sum games without Zugzwang, an approach that has proven successful in minimax evaluation of these games.

Chapter 3

The Game Bench: Goals and Design Concepts

In this chapter we introduce the *Game Bench*, an application framework for combinatorial game programs. In Section 3.1 we discuss general requirements of a game application framework and the main goals of the Game Bench. In Section 3.2 we introduce two main design concepts of the Game Bench, the *game kernel* and the *game application*. In Section 3.3 we discuss related work in the field of combinatorial game programming.

3.1 Goals and Requirements

The *Game Bench* is an application framework that supports implementation and analysis of combinatorial games on the computer. Its main goal is to provide the game programmer with a game independent *search engine* that contains algorithms and data structures implementing the various steps of *decomposition search*. Further, the Game Bench framework supports the implementation of fundamental game playing functionality and the game applications' user interfaces.

We identify the following general requirements of a combinatorial games application framework: *extensibility*, *portability*, *efficiency* and *usability*. Below we show how these requirements are met by the Game Bench and discuss possible conflicts among them.

3.1.1 Extensibility

Extensibility is a very important requirement for application frameworks in general. Only an extensible framework is flexible enough to adapt to the

different needs of its various applications. In case of a game application framework, this specially applies to the "search engine": it should for instance be possible to add a new, general game playing algorithm and make it available to all game applications based on the framework.

The Game Bench guarantees extensibility by its object-oriented design and choice of programming language (Java). The Game Bench is extensible on two different levels:

1. By organizing the common components of search algorithms in basic classes, the Game Bench provides an *extensible search engine*. The programmer can easily add a new sum play algorithm by extending a basic sum play algorithm that already provides the functionality to decompose a game and compute and store local game results.

2. The Game Bench does not prescribe in detail how a final game application must be designed, but it provides a *basic game application* which the programmer extends and tailors to her own needs. She adds the desired functionality to an *application skeleton* that already implements basic operations common to all game applications.

To some extent, extensibility conflicts with the goal of efficiency. For instance, a game specific, highly optimized search engine usually outperforms an extensible, general-purpose search engine. We have tried to find a good balance between the two requirements although we consider extensibility to be of slightly greater importance.

3.1.2 Portability

Portability is a standard design requirement in software development. Without being portable, software cannot survive in its rapidly changing environment. The choice of *Java* as its programming environment makes the Game Bench portable not only by source code, but also by object code. This is an advantage, if a user wants to run a game application, but has not installed the necessary programming tools to compile the source code on her machine.

As does extensibility, portability also conflicts with the goal of efficiency. The Game Bench uses Wolfe's *Gamesman's Toolkit* [52] (see also the next section) in form of a pre-compiled dynamic library as an engine for all basic CGT computations. The toolkit is provided for *Windows 95/98/NT*, *Solaris* and *Linux* platforms. However, it compiles on any platform that supports the *java native interface* [32], for example *MacOS* and other versions of *Unix*.

3.1.3 Efficiency

Efficiency is important in game applications. The following factors make the Game Bench reasonably fast and thus also suited to attack "big" search problems (like for instance some of the complex pawn structures analyzed in Chapter 7).

- On most platforms, *just in time compilation* enables Java byte code to be executed in a compiled form. Although not quite as fast as object code generated by a *C* compiler or assembly code, the gap is very small in contrast to interpreted Java code.

- The Game Bench uses the *Gamesman's Toolkit* [52] as its basic combinatorial game calculator. The toolkit is provided as a pre-compiled library and dynamically linked to the game application by using the *java native interface* [32]. If portability by object code is not the user's primary goal, other time critical routines, for example a game's move generator, may be implemented as native methods and pre-compiled for a local machine as well.

3.1.4 Usability

"Software which is difficult to use wastes the time of its users, makes them angry, leads to high support expenses, or is not used at all." (A. Marzetta [33]). We have tried to make the Game Bench as simple as possible while still providing a useful set of combinatorial game features. By extending a few basic classes which implement the rules of a game, an application is developed step by step. A first quickly developed version enables the user to play moves on a game board. Later, other functionality, for example the use of hash tables by the search engine can be added.

A measure for usability is the time which users must invest to obtain a productive result [42]. The use of the Game Bench as a programming base in student projects (for example [8], [47]) has shown that within a week, it is possible to create a simple application that plays a combinatorial game.

3.2 Design Concepts

The goal of separating game independent from game specific algorithms and data structures is reflected in the design of a game application framework. All game specific functionality, for example executing and taking back moves on a game board, should be encapsulated in abstract data types in order to

hide game specific details and reduce the game specific functions to a set of operations common to all games handled by the framework. This approach has proven successful in the *Smart Game Board* [29] which is based on a modular design.

However, we feel that the separation of game independent and game specific components is ideally modeled with an *object-oriented design.* The framework specifies game specific functionality in a set of abstract classes visible to all its components. The game programmer extends these classes and implements the prescribed function prototypes (see Figure 3.1). The advantages of an object-oriented design are:

- Abstract classes serve two purposes simultaneously: they specify the required game specific functionality (in the form of function prototypes without implementation), and they can provide fully implemented support functions.
- The design naturally reflects the logical separation of game independent and game specific functionality. The abstract classes specify the operations common to all games whereas their extensions implement the game specific details.
- The game independent components (for example a game tree search algorithm) only need to "know" and work with the abstract classes that define game specific functionality. As a consequence, there is no need to recompile such a component when adding a new game, and the components can even work with different games at the same time.
- Last but not least, an object-oriented design is elegant and transparent which has a positive effect on the time needed to get acquainted with and use the framework.

The Game Bench provides two main levels of abstraction. The *game kernel* implements basic functionality required for playing a game. The *game application* connects the game kernel to other components such as the search engine and the user interface (see Figure 3.1).

3.2.1 The Game Kernel

The game kernel is the core of a game application. It implements the *game state*, the application's central data structure. The game state represents the actual position of a game. Operations on the game state include executing and taking back moves and detecting whether a game is finished.

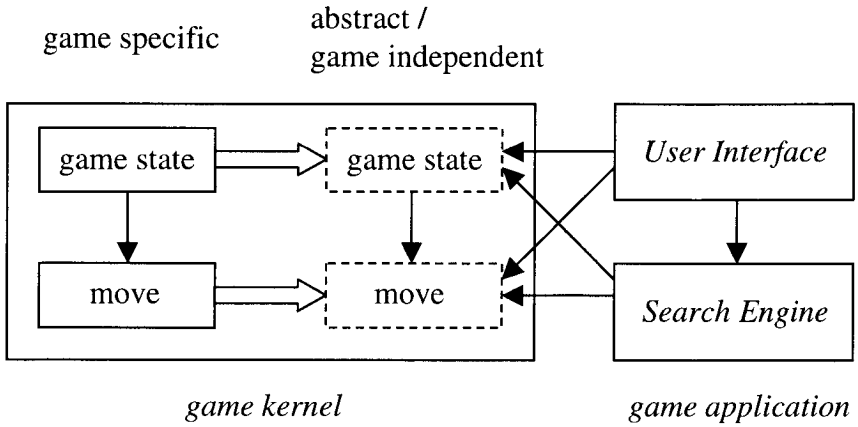


Figure 3.1: The design concepts of *game kernel* and *game application*: The abstract game kernel defines all game specific functionality and serves as an interface to the search engine and the user interface.

We distinguish the *abstract game kernel* provided by the Game Bench and its extension, the specific game kernel of a game application (see Figure 3.1). The abstract game kernel serves two purposes:

1. It specifies the *game playing interface* that defines the game kernel's game specific functionality. This interface connects the game kernel to other components of the Game Bench such as the search engine and the user interface.

2. It provides game independent support for the implementation of the functionality defined by the game playing interface.

The design of the game kernel leads to a clear and logical separation of game independent and game specific functionality. All game specific operations are defined in abstract classes as abstract methods. The programmer adds the functionality of a certain game by extending these abstract classes and implementing the prescribed abstract methods.

3.2.2 The Game Application

The task of the game application is to connect and control its main components (see Figure 3.2):

- the *game kernel* that implements the game state and its operations as discussed in the previous section.

- the *search engine* that provides the tools for the implementation of decomposition search: the basic CGT calculator, algorithms and data structures for local game tree search, and an extensible set of sum play algorithms.

- the *user interface* that displays the game state and enables the user to control the application, for example by entering a move or starting the search engine.

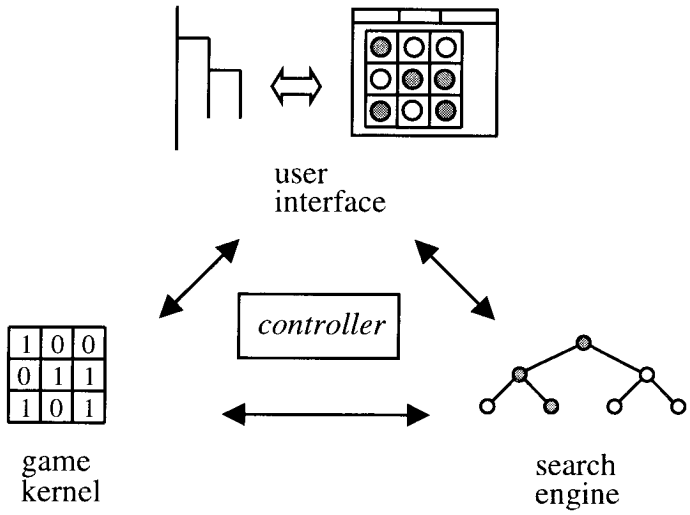


Figure 3.2: The main components of a game application: the *game kernel* holds the game state, the central data model of a game. The *search engine* implements decomposition search. The *user interface* enables interaction with the application.

By providing a *basic game application* the Game Bench directs the programmer in creating his own game application. The basic game application serves as a base for all game applications. Its design is based on the *model-view-controller* paradigm [31], a concept widely used in object oriented programming. The *central data model*, the game state, is represented by one or more *views*. The game state notifies its views of any changes in order to make them update their contents. The *controller* commands the model and/or the views to change as appropriate, for example when the search engine wants to execute a move on the game state.

The basic game application provides a simple user interface that consists of a main application window and the following components (see Figure 5.3):

- one or more *game views* that display the game state and enable the user to interact with it, for example by clicking on a square where a

stone should be placed.

- an extensible set of menus and menu commands that enable the user to control the application.

- a *tree of moves* data structure that stores lines of play entered by the user and a *tree navigator* panel that enables the user to move in this data structure.

3.3 Related Work

3.3.1 The Gamesman's Toolkit

David Wolfe's *Gamesman's Toolkit* [52] implements most of finite combinatorial game theory presented in *Winning Ways* [5]. The toolkit provides abstract data types for games and lists of games, and all the operations (and more) we have discussed in Section 2.2 .

The toolkit is used in two different ways:

1. A textual interface enables the toolkit to be used interactively as a *games calculator*. For instance the user may type "$G = \{0 \mid 1\}$" in order to assign the game $\{0 \mid 1\}$ to the variable G. The toolkit evaluates the game to $\{0 \mid 1\} = 1/2$ and stores it in G. Used as a games calculator the toolkit is enormously valuable when analyzing combinatorial games "by hand".

2. The second way to use the toolkit is as a *library* of CGT functions. The toolkit is written in the *C* programming language under Unix, but compiles on every platform that provides an *ANSI-C* compiler. The Gamesman's Toolkit does not play games itself, but many game playing programs use it as a basic games calculator.

3.3.2 The Smart Game Board

Anders Kierulf's *Smart Game Board* [29] is a workbench for game playing programs, developed on the Apple Macintosh under Modula-2. Its main focus is a *game tree* data structure that enables the user to edit and store *game documents* in a similar way a word processor edits and stores text documents.

The Smart Game Board is based on a modular design that specifies game specific functions in *definition modules* and hides their implementation from its game independent components. It provides a game independent search

engine that implements minimax evaluation using techniques such as *iterative deepening*, *transposition tables*, *alpha-beta* and *scout* tree search etc.

On the Smart Game Board, many popular games have been implemented such as *Go* and *Othello* which originally came with the Game Board, and later *Nine Men's Morris* (Gasser [20]) and *chess* (Wirth [51]). Some of these games, for instance *Go* (see below) and *chess*, have been further developed as specific game applications.

Explorer

Explorer is a Go program running on the Smart Game Board. It was originally written by Kierulf and Chen [30], and further developed by Müller [39] who designed and implemented a sum game model for computer Go. The Explorer program applies a divide and conquer approach to the game of Go in two different areas:

1. It uses a *heuristic* sum game model for the entire game of Go.
2. It computes *exact* solutions to late endgame positions with the help of combinatorial game theory.

Both applications are based on algorithms and techniques of combinatorial game theory. Explorer uses a port of the Gamesman's Toolkit [15] to perform basic CGT operations, and extends the toolkit's functionality with a *thermograph* data structure and *thermostrat* sum game play.

The Simple Game Board

The *Simple Game Board* [28] is a "mini version" of the Smart Game Board designed as an educational program that introduces computer science students to minimax evaluation and game tree search. Like the Smart Game Board it is based on a modular design and provides a game independent search engine that implements alpha-beta search.

3.3.3 Gamesman

Dan Garcia's *Gamesman* [18] is a system for generating graphical parameterizable game applications. Based on a description of the rules of a game, it automatically generates a stand-alone application that plays this game. Small games can be solved using *exhaustive search*, bigger games are played using *minimax evaluation*. Gamesman is written in the *Tcl/TK* script language under *Unix* and generates *X-Window* applications. It supports finite,

2-person games with perfect information and comes with the games *1210*, *TicTacToe*, *TacTix* and *Dodgem*.

Xdom

Dan Garcia's *Xdom* [19] is an *X-Window* based front-end for playing Domineering (see Section 5.1.1 for the rules of this game). It is written in *Tcl/TK* and uses the Gamesman's Toolkit to compute a game's value. Xdom allows the user to read game positions from files and to play against human or computer opponents. Further, it can display a game's value and give hints about good moves.

3.3.4 What does the Game Bench provide?

In contrast to the above described game application frameworks, the main focus of the Game Bench is its extensible search engine. The Game Bench provides an ideal environment for designing and testing divide and conquer algorithms based on combinatorial game theory. Its extensible design enables the Game Bench to adapt to the needs of specific game applications and yet provide general functionality common to all combinatorial games. Its portability makes the Game Bench available on all popular platforms and decreases its chances of dying together with its programming environment or operating system which unfortunately happened to the (old) Smart Game Board[1] and the Simple Game Board.

[1]Kierulf is developing a new commercial version for Windows under C++ with Go as its main focus.

Chapter 4

The Game Bench: Architecture and Implementation

In this chapter we describe the architecture of the Game Bench in detail. We distinguish three main components (see also Figure 3.2): the *search engine* (Section 4.2), the *game kernel* (Section 4.3) and the *game application* (Section 4.4). Each component consists of several Java classes, some also contain abstract classes and interfaces. We describe the most important classes, data structures and functions of each component. A complete description of all classes is available at the Game Bench web-site [34].

4.1 Conventions

For discussing the Game Bench's architecture and implementation we use the following conventions:

- *Class hierarchy diagrams* (see Figure 4.1) show the connections and dependencies of the Java classes within a component or an application.

- Java source code is typeset with a `mono-spaced font`. Names of *constants* start with a '`k`', for example the player `kLeft`. Names of *class variables* and *instance variables* start with a '`f`' (field), for example the flag `fUseHashing`. Names of *classes* always start with capital letters, names of *methods* with lower-case letters.

For an explanation of Java specific and object-oriented terms see the glossary in appendix A. A detailed description of all the Game Bench classes and their hierarchy can be found on the Game Bench web-site [34].

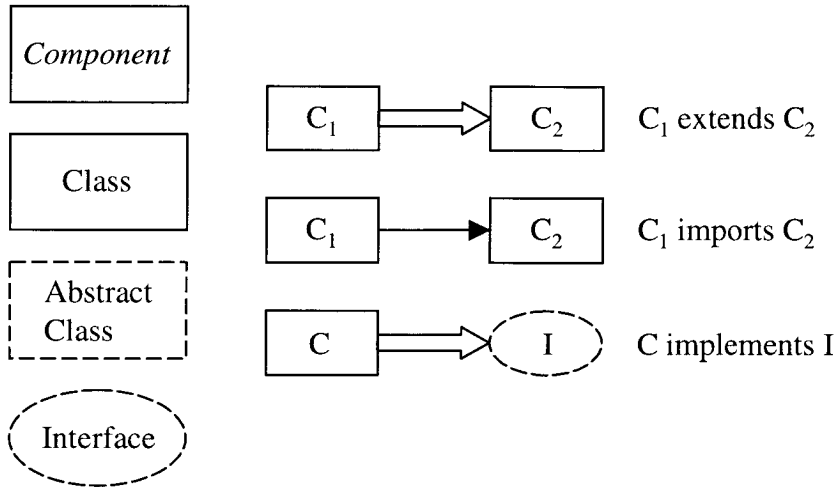


Figure 4.1: Symbols and conventions used to describe hierarchies of Java classes.

4.2 The Search Engine

The *search engine* is the core of the Game Bench. It implements *decomposition search* of general combinatorial games that meet the specifications of the game playing interface (see Section 4.3.1).

4.2.1 Requirements of a Game Independent Search Engine

Providing a universal, game independent search engine that meets the demands of every specific game is a very difficult, if not impossible task. Especially heuristics, even generally applicable ones, do not produce the same results with different games. What works fine in one game, might be unsuccessful in another. Thus, a general, game independent search engine must be *flexible* and *extensible*, otherwise it will be abandoned in favor of a game specific solution.

On the other hand, even highly specialized, game specific search algorithms share the same base. Also different game independent algorithms may have much in common. For instance, the sum play algorithms *hotstrat* and *thermostrat* both start by decomposing a game, then compute local values, and only differ in the way they evaluate the local results to make a move decision.

The search engine's main goal is *extensibility*. It provides game independent algorithms and data structures that implement decomposition search in

a general way, but yet allow the game programmer to adapt it to her special needs. In the following sections we discuss the design and implementation of the search engine's main components: *local search*, *sum game play* and the *CGT kernel*.

4.2.2 Local Search

Local combinatorial search produces a local game's tree up to its terminal positions (or up to an artificially defined search horizon) and backs up the leaf values in order to compute the value of the root. The basic local search algorithm is described in Section 2.3.2.

Local Game Data

Local search gathers the necessary data used by sum play algorithms to make move decisions. Depending on the various sum play algorithms the data collected for a node (position) in the game tree includes:

- the node's *value*.
- the node's *temperature*.
- the node's *leftscore* and *rightscore*.
- the node's *incentive*, that is the incentive of the move from the parent node to this node.

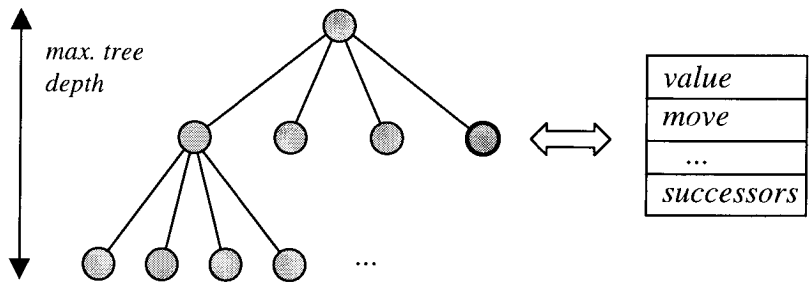


Figure 4.2: The Local Tree Data Structure stores the game tree up to a certain depth. A node (game position) stores a game value, the move that leads to this node, and links to its successors. Optionally additional data, for example the position's temperature, is stored.

In order to adapt to the different requirements of various sum play algorithms, the Search Engine provides a local tree data structure that stores

the results of local search (see Figure 4.2). A parameter passed to the local search algorithm determines up to which depth the local game tree is stored. For instance a "simple" sum play algorithm only needs to access the first level of the local game trees.

Hashing

In order to prevent evaluating the same game position more than once, for instance due to a transposition of moves, search algorithms use hash tables to store previously evaluated game positions and their values. The search engine supports hashing on a game specific as well as on a game independent level:

- On the game specific level, the search engine supports *global hashing* and *local hashing*. Games that implement the interface *Hashable* (see also Section 4.3) supply a hash function that maps the overall position of the sum game to a 64-bit integer value. Games that implement the interface *LHashable* provide a hash function that maps independent local games to 64-bit integer values. Local hashing is important in decomposition search as it allows to identify previously evaluated local games independent of their sum game context (see Figure 4.3).
- On the game independent level, the *Gamesman's Toolkit* [52] which serves as the search engine's CGT kernel (see Section 4.2.4) manages a hash table of evaluated combinatorial game expressions.

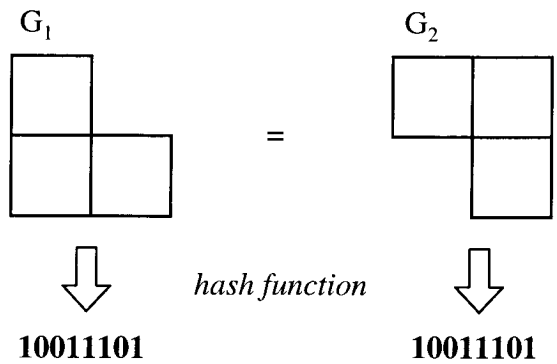


Figure 4.3: Local Hashing: The hash function maps the (identical, mirrored) local *Domineering* (see Section 5.1.1 for the rules of this game) games G_1 and G_2 to the same hash code. The computed game value $G_1 = \{0 \mid 0\} = *$ is stored in the hash table and looked up when the search arrives at the game position G_2.

Heuristics

The technique of using heuristic evaluation functions that estimate game values of non-terminal positions in order to create an artificial search horizon is well known from minimax evaluation. It allows to compute approximate values of games too big to be searched entirely.

There are also combinatorial games that permit heuristic evaluation functions (see Chapter 8). The interface *CombHEval* specifies a function that yields heuristic values for game positions. Games that implement this interface are searched with an iterative deepening algorithm that successively increases the search depth until it is stopped or it has computed the exact value of the game.

Multi-Threading

Divide and conquer algorithms suggest computing their sub-tasks (the divide phase) in parallel. An implementation of decomposition search should therefore support running several independent local searches at the same time. The search engine is designed for parallel local search:

- Each instance of the local search algorithm is automatically run in its own *thread*. Running on a multi-processor machine, the java runtime system can evenly distribute the search tasks on the available processors.

- Global data structures used by several search threads at the same time, for instance a hash table, can be protected by *semaphores*. As a result, simultaneous read and write access to these data structures is possible.

- Local game trees of games that admit heuristic evaluation functions are searched depth-first using *iterative deepening*. As a result, parallel decomposition search can yield approximate results at virtually any time of its computation, not only when all local search tasks are terminated.

4.2.3 Sum Play Algorithms

The implementation of decomposition search consists of three main steps (see Figure 4.4):

1. *Game decomposition*: The game is divided into independent components, the local games G_i.

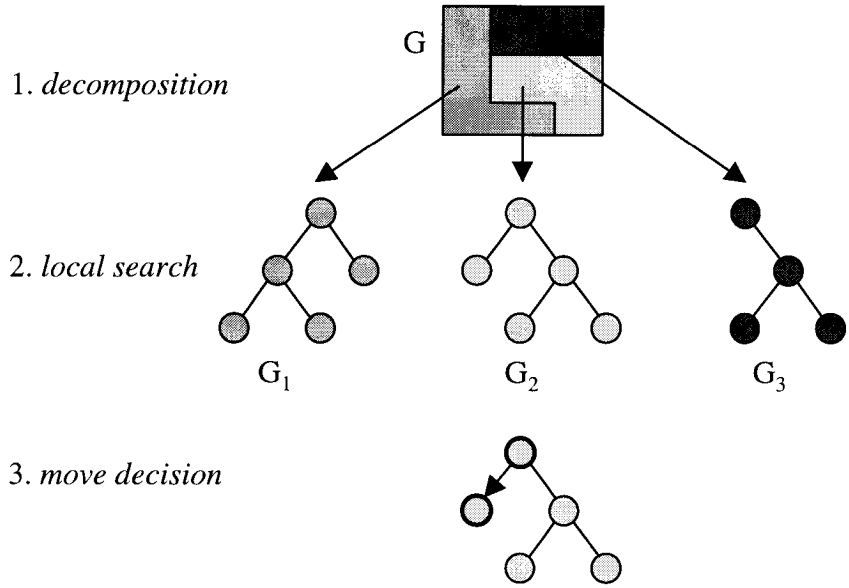


Figure 4.4: The sum play controller organizes the steps of decomposition search: 1. the game is divided into independent subgames. 2. the local games are evaluated. 3. a move decision is made based on the local results.

2. *Local search*: The local game trees are evaluated and the results stored. (see also Section 4.2.2.)

3. *Sum game evaluation*: Based on the local results, a move decision in the sum game $G = G_1 + G_2 \cdots + G_n$ is made. This may or may not involve actually computing the sum.

Steps one and two are common to all decomposition search algorithms except for differences in what local data is required in order to evaluate the sum game. The search engine provides a basic sum play algorithm (class *SumPlayController*) that performs steps one and two, and an extensible set of algorithms that are built on this basic algorithm. This structure allows the programmer to easily add algorithms constructed for a specific game, as well as game independent algorithms that expand the search engine's functionality. An extension of the search engine's set of algorithms does not require any of the existing classes to be changed.

4.2.4 The CGT Kernel

The CGT kernel performs the basic combinatorial game operations such as composing a new game from two lists of left and right. The most important

requirement for the CGT kernel is efficiency. Every time the search engine evaluates a node in a game tree, one or more basic operations are performed.

The search engine includes and uses Wolfe's *Gamesman's Toolkit* [52] as its CGT kernel. The following points are important to note about this approach:

- The Gamesman's Toolkit has been developed and improved over a period of several years. It implements virtually almost the entire scope of finite combinatorial game theory presented in *Winning Ways* [5]. The task to implement a kernel of the same quality and scope would be extremely difficult and time consuming.

- The Gamesman's Toolkit attaches great importance to efficiency, for example by using a hash table in order to avoid evaluating the same expression more than once. Further, the toolkit is written in the *C* programming language which for the time being produces more efficient object code than Java compilation.

- The cost of including an efficient native library in the search engine is a loss of portability on the object-code level. However, the toolkit library compiles on any platform that supports the java native interface including most Unix versions, Windows 95/98/NT, and MacOS, thus the most popular operating systems.

The implementation, that is the inclusion of the Gamesman's Toolkit in the search engine is based on the *java native interface* [32]. The class *CGTCalc* provides the whole set of functions implemented by the Toolkit in the form of Java methods.

An interface written in *C* (*TKInterface.c*) communicates with the Java side and calls the required toolkit functions (see Figure 4.5). This interface and the original toolkit sources are compiled and linked into a dynamic library which is loaded by the Java runtime system. The Game Bench provides pre-compiled versions of the toolkit library for *Solaris*, *Linux* and *Windows 95/98/NT* platforms.

4.2.5 Class Hierarchy and Descriptions

Figure 4.6 shows the hierarchy of the search engine's main classes. Below we briefly discuss their functionality.

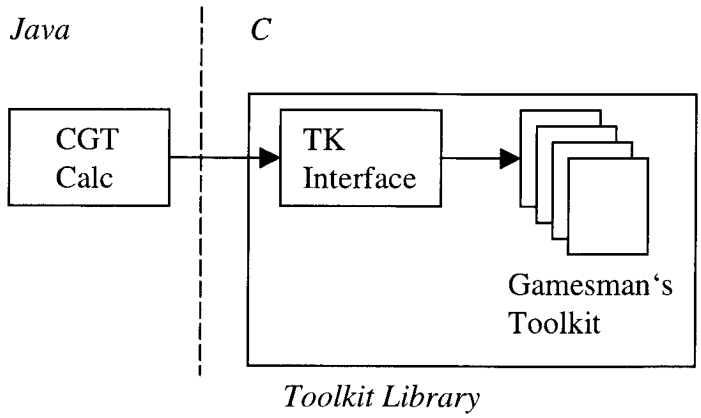


Figure 4.5: The CGT Kernel consists of the java class *CGTCalc*, the native interface *TKInterface* and the Gamesman's Toolkit object files.

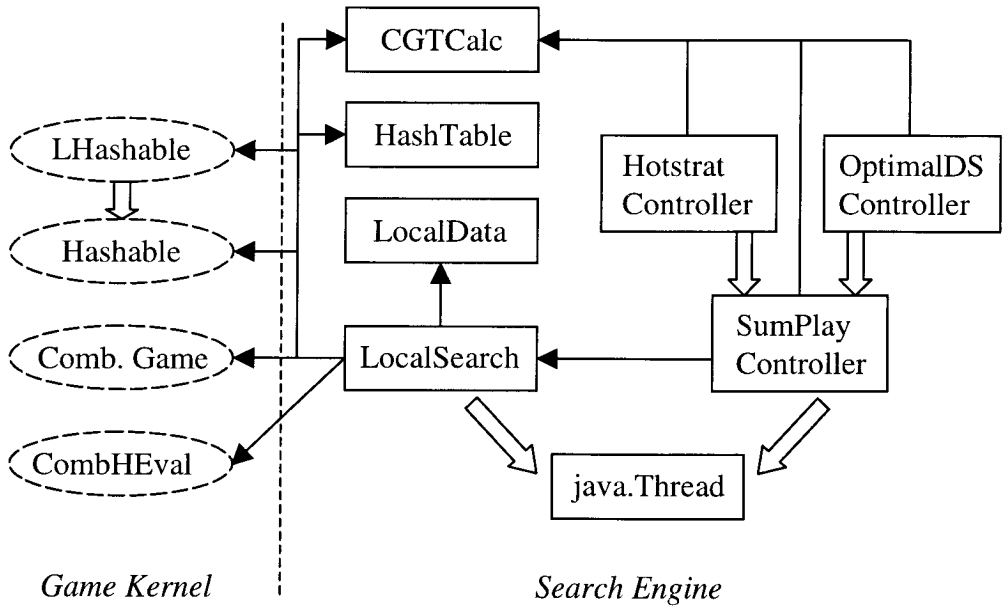


Figure 4.6: The class hierarchy of the search engine and its interface to the game kernel.

class HashTable

The class *HashTable* implements a hash table that stores information (class *HashEntry*) on game positions identified by 64-bit hash codes. Games that provide a function which maps game positions to hash codes implement the interface *Hashable* (global hashing) or *LHashable* (local hashing). These two interfaces are part of the game kernel and are discussed in Section 4.3.

class CGTCalc

The class *CGTCalc* implements the java interface to the *Gamesman's Toolkit* [52] that is used to perform all basic CGT operations. The toolkit itself is linked to the Game Bench in the form of a dynamic library.

class LocalData

The class *LocalData* stores local game information gathered by the local search algorithm. This data includes the game value, temperature, leftscore and rightscore of the actual position, the incentives of both players' options, and the whole tree of possible moves up to a specified depth.

class LocalSearch

The class *LocalSearch* implements local combinatorial game tree search (see Section 2.3.2) and computes values of partizan and impartial combinatorial games. Games to be searched must implement the interface *CombinatorialGame* (move generation and termination detection). Optionally, the search uses a heuristic evaluation function (interface *CombHEval*) that estimates values of non-terminal nodes in order to create an artificial search horizon. By extending the system class *java.Thread*, the class *LocalSearch* implements the interface *java.Runnable* and therefore may be run in parallel to the game application and to other search threads.

class SumPlayController

The class *SumPlayController* implements the basic sum play algorithm. It provides operations common to all sum play algorithms such as decomposing the game into its independent components and gathering local information. Further, the sum play controller also manages the possible parallel execution of the local search threads. The subclasses of class *SumPlayController* build the Game Bench's library of sum play algorithms.

class OptimalDSController

The class *OptimalDSController* extends the class *SumPlayController* and implements *optimal sum play*. If after deleting all moves with dominated incentives, more than one option is left, the sum of all local games is computed in order to determine the best move.

class HotstratController

The class *HotstratController* extends the class *SumPlayController* and implements *hotstrat*, a heuristic sum play algorithm based on local game information. Hotstrat always plays in the local game with the highest temperature. This simple version of the algorithm chooses randomly among all possible non-dominated move options. Extensions of class *HotstratController* might use more sophisticated local criteria to select a move, for example by considering game specific properties.

4.3 The Game Kernel

We have introduced the game kernel as a main design concept of the Game Bench in Section 3.2.1. The game kernel provides the essential game playing functionality: it implements the *game state*, the data structure on which a game is played, and the rules of a game.

A game playing kernel is a central component of any game application framework, and it should meet the following requirements:

- *Structure*: The game kernel must separate game independent and game specific operations. It gives an exact specification of the required game specific functionality to other components of a game application.
- *Extensibility*: The design of the game kernel should enable the programmer to extend its game independent, as well as its game specific functionality.

In the Game Bench, the separation of game independent and game specific functionality is performed by the *abstract game kernel* which serves two purposes:

1. It specifies the *game playing interface*, i.e. the functions that a the final game kernel must implement.
2. It provides game independent support for the implementation of the game playing interface's functionality.

4.3.1 The Game Playing Interface

The game playing interface specifies the game kernel's functionality and makes it available to the other components of a game application. We distinguish basic functionality and optional features a game kernel may additionally provide. The basic functionality is specified by the abstract class *GameState*, the base class of any game specific kernel. Additional features are specified in several interfaces that the game specific kernel may optionally implement.

It is important to separate the specification of game independent and game specific operations. For instance, the geometric decomposition of a "game board" may be performed independent of any specific game and provided by the abstract game kernel. Generating all moves in a local game, on the other hand, is a game specific operation that must be implemented by the abstract kernel's extension.

Basic Functionality

The basic functionality of the game kernel is defined in the abstract classes *GameState* and *Move* which any game kernel must use as its base classes. Basic operations include:

- executing and taking back moves.
- resetting the game to an initial default state (the operation "new game").
- determining whether play has arrived at a terminal position.

Game Play

If a game application is to use the Game Bench's search engine, its game kernel must implement the functionality specified in the interface *CombinatorialGame* which includes:

- generating both players' possible moves in any game position.
- determining the values of terminal positions. Strictly speaking, there is only one terminal value $0 = \{ \mid \}$, but in practice any value is possible. We might for instance consider a game terminated as soon as it simplifies to a number as "playing out" numbers is trivial.

An extension of *CombinatorialGame* is the interface *ImpartialGame* which does not specify additional functionality, but serves to identify a game as an impartial game. In this case, the search engine can optimize searching the game's tree.

Hashing

The interface *Hashable* specifies a hash function that maps game positions to 64-bit integer values. An extension of *Hashable* is the interface *LHashable* which defines hash functions for local games.

Heuristics

The interface *CombHEval* specifies a heuristic evaluation function that yields game values for non-terminal positions. This function is used to create an artificial search horizon when searching local games that are too big to be searched exhaustively.

Game Decomposition

In order to apply decomposition search, we must partition a game, i.e. identify its independent components, the local games. The interface *Partitionable* defines the required operations for partitioning a game. This includes:

- partitioning the whole game and enumerating its local components.
- re-partitioning a certain local game, identified by its part number.

It is important to note that for certain classes of games this functionality can be implemented independent of the actual game. For instance the process of identifying local components in *Domineering* (see also Section 5.1.1) can be generalized to similar games played on rectangular boards (class *PartRectBoard*).

Further, we need a move generator that generates moves for local games. This function is game specific and is specified by the interface *PartitionGame*, an extension of *Partitionable*.

4.3.2 Support Classes

The second function of the abstract game kernel is to offer game independent support for the implementation of the functions specified by the game playing interface. Many games are played on *rectangular boards* or on *graphs*. The Game Bench provides base classes that offer general functionality for such games. In the case of rectangular boards, it is also possible to provide base classes that implement further interfaces of the game playing interface:

- *Hashing*: Hash codes for games played on sets of points or squares are best generated by an algorithm described by Zobrist [55]. The main

advantage of Zobrist's method is the possibility to update the hash code of a game position incrementally whenever moves are executed or taken back. The class *HashCoder* implements Zobrist's algorithm and enables the abstract class *RectBoard*, an extension of the abstract class *GameState*, to automatically generate hash codes (interface *Hashable*) for rectangular boards.

- *Game Decomposition*: Games whose moves consist of placing immobile stones on a board all decompose in more or less the same way. The abstract class *PartRectBoard* represents such games and automatically identifies independent local games, thus implements the interface *Partitionable*.

4.3.3 Class Hierarchy

Figure 4.7 gives an overview of the abstract game kernel's most important classes. The game programmer creates the game kernel of a specific game by extending the abstract classes *Move* and *GameState* (or one of its subclasses, e.g. *RectBoard*), and by optionally implementing the interfaces for game playing, hashing and game decomposition.

abstract class GameState

The abstract class *GameState* specifies the basic game playing functionality that every game must implement: executing and taking back moves, and detecting game termination.

abstract class Move

The abstract class *Move* is the base class of any game specific implementation of a move in a game. It stores the player who made the move and specifies abstract methods for copying and textual output. Game specific extensions add their required move data, for instance a "Domineering move" (see also Section 5.2) might store the coordinates of the squares on which a domino stone is placed.

interface CombinatorialGame

The interface *CombinatorialGame* specifies the minimal functionality required for a game to use the search engine. This includes a move generator and a function that maps terminal positions to combinatorial game values.

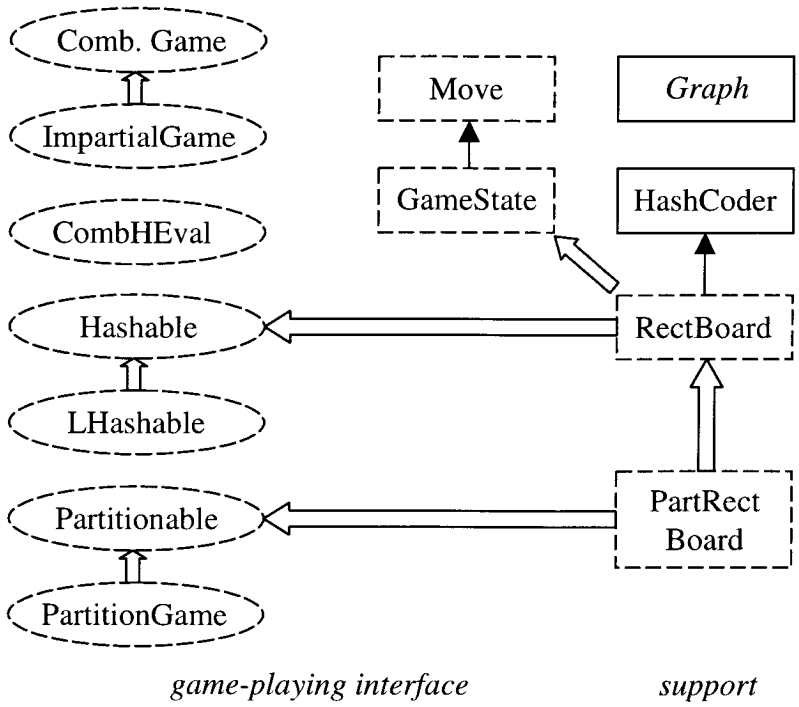


Figure 4.7: The class hierarchy of the abstract game kernel.

interface ImpartialGame

The interface *ImpartialGame* extends *CombinatorialGame* and defines an impartial game. It does not specify additional functionality, but is used to identify impartial games in order to allow search optimizations.

interface CombHEval

The interface *CombHEval* specifies a heuristic evaluation function that estimates the combinatorial game values of non-terminal positions.

interface Hashable

The interface *Hashable* specifies a hash function that maps game positions to 64-bit hash codes. Games that implement this interface enable the search engine to use transposition tables that store the values of previously evaluated game positions.

interface LHashable

The interface *LHashable* extends *Hashable* and specifies a hash function that maps local games (game partitions) to 64-bit hash codes (see also Figure 4.3).

interface Partitionable

The interface *Partitionable* specifies games that implement decomposition into independent subgames (local games). The basic functionality defined in this interface can be extended by the operations defined in the interfaces *PartitionGame* and *LHashable*.

interface PartitionGame

The interface *PartitionGame* extends *Partitionable* and specifies a move generator for local games.

abstract class RectBoard

The abstract class *RectBoard* extends *GameState* and implements the basic functionality common to rectangular grid-boards. Further, *RectBoard* provides an automatic hash code generator (class *HashCoder*) and support for bit-boards (a data structure that represents each square of the rectangular board by one bit).

abstract class PartRectBoard

The abstract class *PartRectBoard* extends *RectBoard* and implements rectangular boards that allow a geometrical decomposition. An example of such a game is *Domineering* (see Section 5.1.1). The abstract class *PartRectBoard* implements the interface *Partitionable*.

component Graph

The class *Graph* stores and manages a graph data structure that may be used to play a game on. The class *GraphNode* implements the basic node in the graph data structure. The node is identified by a unique (within a graph) number, and it keeps a list of edges to other nodes.

4.4 The Game Application

We have introduced the design concept of the *game application* in Section 3.2.2. Its task is to combine and control the main components game kernel, search engine and user interface. An application framework should not only provide the necessary components that build an application, but also direct the programmer at organizing these components. In order to facilitate the implementation of a final application, the *basic game application* provides basic functionality common to all game applications:

- a simple *user interface* that specifies the basic view component and implements the control mechanism as defined by the *model-view-controller* paradigm.
- a *tree of moves* data structure that stores lines of play entered by the user or generated by the search engine.

4.4.1 The Tree of Moves

The tree of moves is organized as a tree of *property lists*, a concept introduced in Kierulf's *Smart Game Board* [29]: Each node of the tree stores a *move property* and optionally some further properties such as comments on moves or information on the time used by the players (see Figure 4.8). Some properties are *executable*, they alter the game state when a node is executed or undone. An example of an executable property is the move property, the stored move is executed when moving forward and taken back when moving back in the tree.

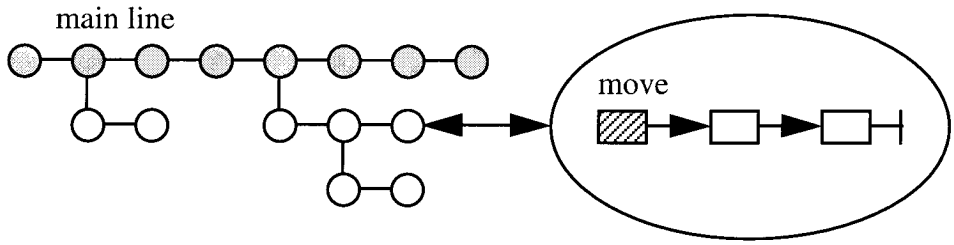


Figure 4.8: The tree of moves consists of the main line (grey) plus some sidelines which represent alternate variations of play. Each node stores a move property plus optionally some other properties that describe the node.

The *tree controller* manages the tree of moves. It stores the tree and keeps track of the *current node* which is displayed at a given moment. It

provides commands to move in the tree (i.e. to change the current node) like *forward*, *left* or *back*, and it allows new moves to be added to the tree.

4.4.2 The User Interface

The user interface of the basic game application performs two main tasks:

1. It specifies the *basic view object*, a graphic component representing a certain data model, and it implements the *view controller* which stores and manages an application's view objects. The programmer adds a game specific view (for instance a graphic representation of the game board) to the application by extending the basic view object and registering the new view with the view controller.

2. It provides the class *application* which is the base class of any game application implemented on the Game Bench framework. The class *application* implements functionality common to all game applications:

   - It creates and opens the main application window and allocates and initializes the required controller objects such as the tree controller and the view controller.
   - It manages an extensible set of menus and menu commands that enables the programmer to add his own commands to the provided default commands.
   - It implements a graphic *control panel* that enables the user to navigate in the tree of moves by sending the tree controller commands like *forward*, or *back* and to switch the right to move between the players.

The user interface components of the basic game application are shown in Figure 5.3 with *Domineering* as an example.

4.4.3 Class Hierarchy

Figure 4.9 shows the class hierarchy of the basic game application.

class Application

The class *Application* extends *java.awt.Frame* (the standard Java class for a "window") and is the main class of the basic game application. It controls the game kernel and the search engine and provides a simple, extensible user interface. The class *Application* is the superclass of all Game Bench applications.

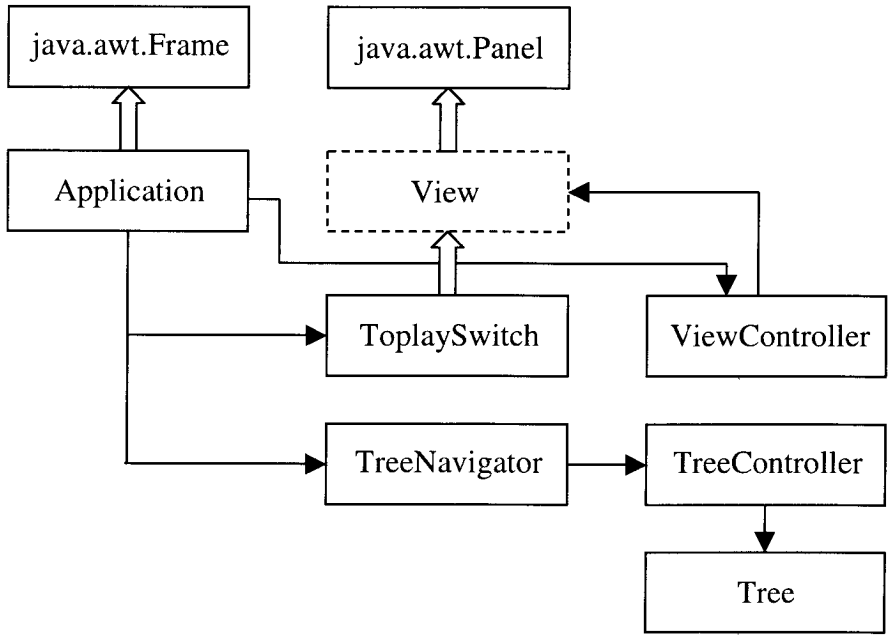


Figure 4.9: The class hierarchy of the basic game application.

abstract class View

The abstract class *View* specifies the basic view object. In the *MVC Model*, a view represents a certain data model and handles update-messages that provide information on changes of this data model.

class ViewController

The class *ViewController* stores and manages an application's view objects. The application's view controller is automatically allocated and initialized by the *Application* object.

class Tree

The class *Tree* implements the dynamic data structure that stores the *tree of moves*. The tree of moves saves lines of play entered by the user or generated by the search engine.

class TreeController

The class *TreeController* holds an application's tree of moves and keeps track of the *current node* that holds the actual game position. An application's

tree controller is automatically allocated and initialized by the *Application* object.

class TreeNavigator

The class *TreeNavigator* is a graphic component that implements an interface to the tree controller. It enables the user to click her way through the stored tree of moves.

class ToplaySwitch

The class *ToplaySwitch* is a graphic component that displays and enables the user to switch the right to move between the players.

Chapter 5

The Game Bench: Domineering, a Programming Example

In this chapter we show how the programmer creates her own game application using the Game Bench framework. As a simple programming example we implement the game *Domineering*. In Section 5.1 we introduce the rules of Domineering and discuss the necessary implementation steps. In Section 5.2 we show implementation details of the game kernel, the game view, and the main application. In Section 5.3 we summarize the class hierarchy of the Domineering application and present statistics on the size of its components' source code.

5.1 Introduction

5.1.1 The Game Domineering

The game *Domineering* [5] is played with standard domino stones on a (usually rectangular) board. At his turn, each player places a stone in such a way that it covers exactly two adjacent squares. *Left* must place his stones vertically, *Right* horizontally. The first player unable to move, loses. Figure 5.1 shows a game of Domineering played on a 6×7 board.

Domineering is a partizan combinatorial game. The game is likely to decompose into independent components during play as the example in Figure 5.1 shows. Thus, it suggests the application of *decomposition search*.

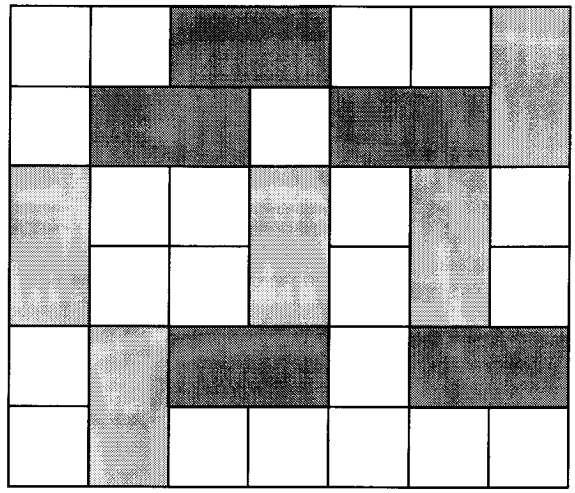


Figure 5.1: Domineering on a 6×7 board.

5.1.2 Implementation Steps

The implementation of a Domineering program on the Game Bench framework requires the following programming steps:

1. Create the Domineering *game kernel* by extending the abstract game kernel provided by the Game Bench. This step involves defining a move format, designing a data structure for the Domineering game board, and implementing functions for executing moves, taking back moves, and detecting game termination. Further, we also want to implement hashing and board partitioning.

2. Implement the Domineering *game view*, a graphic component that displays the board and reacts to user input (mouse, keyboard ...).

3. Create the main program, the *Domineering application*, by extending the basic game application. This step consists of designing a layout for the graphic components of the user interface on the one hand, and of organizing the application's main components on the other hand.

5.2 Implementation

5.2.1 The Game Kernel

First, we define data structures that represent the Domineering *board* and a Domineering *move*. Then, we implement the functionality specified by the

game playing interface (see Section 4.3.1). Finally, we add functionality for *hashing* and *board decomposition.*

Move and Board Data Structures

Domineering is played on a rectangular board. Each square is either empty or covered by a domino stone of one of the players. With the class *RectBoard*, the Game Bench provides a data structure tailored for such boards. As the game can decompose into subgames, we choose to extend the class *PartRectBoard* that automatically handles board partitioning (see Section 4.3).

A Domineering move consists of placing a domino on two adjacent squares of the board. The class *DominoMove* extends the base class *Move* and additionally stores these two squares:

```
public class DominoMove extends Move
{
   public int fSquare1;
   public int fSquare2;
   ...
   /* methods for
        - comparing moves
      - copying moves
      - textual output of a move
   */
}
```

Game Playing Routines

The class *PartRectBoard* provides the data structure for the rectangular game board. Now, we must implement functions to execute and undo moves, to generate all possible moves (interface *CombinatorialGame*), and to detect whether the game is finished. In order to be able to take back moves, we need a *stack* that stores the played moves. A simple way to implement this stack is to use an array (of moves) and an index that points to the top entry.

The function *execMove* places a domino on the board, updates the hash code of the actual position (see also next section), and pushes the move onto the stack. The function *undoMove* works analogously. The game is finished when neither player has any moves left. In this case, the resulting game value is $G = \{ \mid \} = 0$ and the player to move loses. (It is of course also possible to detect more complex "terminal positions" such as certain numbers, for instance represented by corridors that only allow one of the players to move.)

The method *endOfGame* tests whether there are any adjacent empty squares left on the board. The *move generator* works exactly in the same way as termination detection. We look at all squares and their neighbors, and each time we find two empty adjacent squares, we add a new move to the list of generated moves.

Hashing

In order to implement the *Hashable* interface, we need a function *actualHash* that returns a 64-bit hash code for the actual position. The Game Bench provides the class *HashCoder* that automatically generates hash codes for games played on rectangular boards. All we have to do is to allocate a hash-coder object and update it when the board is changed (in the methods *execMove* and *undoMove*). The return value of the method *actualHash* is the hash code generated by the HashCoder object.

```
public long actualHash()
{
   return fHashCoder.fHashCode;
}/*actualHash*/
```

Board Decomposition

The class *PartRectBoard* already provides the functionality to decompose the board into independent subgames (partitions). In order to use the search engine, we must implement the interface *PartitionGame* that specifies a move generator for local games.

```
public java.util.Vector pGenerate(int player, int partNr);
```

This move generator is easily realized by altering the original (global) move generator in such a way that it only generates moves to squares that belong to the partition specified by *partNr*.

5.2.2 The Domineering Game View

In order to display the actual game state we need a graphic representation, a *view* of the Domineering board. The class *DominoBoardView* displays the board and enables the user to enter new moves by clicking the squares where a domino should be placed.

The board representation consists of a grid, the background, and of the squares which can be empty or occupied by one of the players. Figure 5.2 shows how the Domineering board view is organized.

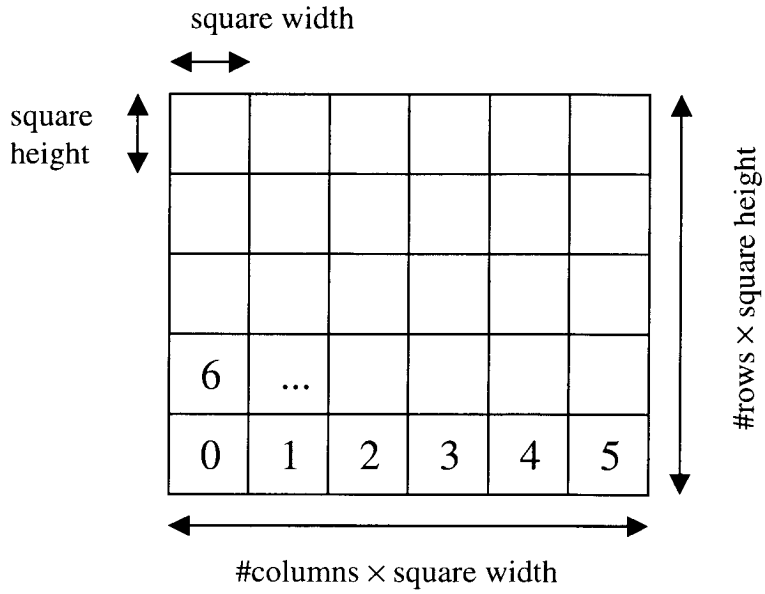


Figure 5.2: The Domineering board view.

One way to implement move input is to register mouse clicks on the boundaries of adjacent squares. If the clicked squares produce a legal move, this move is generated, added to the tree of moves, and executed on the board.

5.2.3 The Domineering Application

The Game Bench provides the *basic game application*, an application skeleton that connects the game kernel and the search engine with a simple user interface. The implementation tasks of a specific game application are:

- allocate the main data structures: the *game state* and at least one *view* of the game state.
- design the layout of the main application window and add the game view to its graphic components.
- choose one of the *search controllers* provided by the Game Bench or write a new game specific search controller and enable its execution in the menu commands *startSearch* and *stopSearch*.

Figure 5.3 shows the final Domineering application "in action".

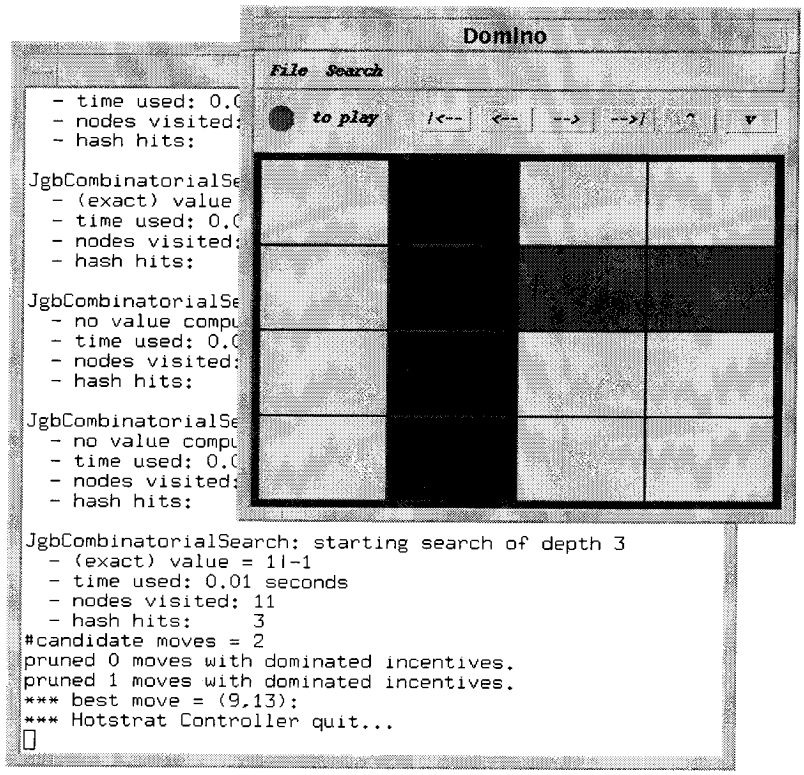


Figure 5.3: The final Domineering application consists of a game window and a text window used for output.

The Search Engine

The basic game application provides "menu hooks" that enable the programmer to install functions that start and stop the search engine. All the extension must do, is allocate a search controller and start respectively stop it in the methods *startSearch* and *stopSearch*. The Domineering application uses the standard *HotstratController* provided by the Game Bench.

```
private HotstratController  fHSController;
...
private void startSearch()
{
   fHSController = new HotstratController(fBoard);
   fHSController.start();
}/*startSearch*/
```

The hotstrat controller automatically decomposes the game board, executes the local searches, and makes a move decision based on the computed local results.

5.3 Class Hierarchy and Statistics

The Domineering application consists of four classes added to the Game Bench framework (see Figure 5.4):

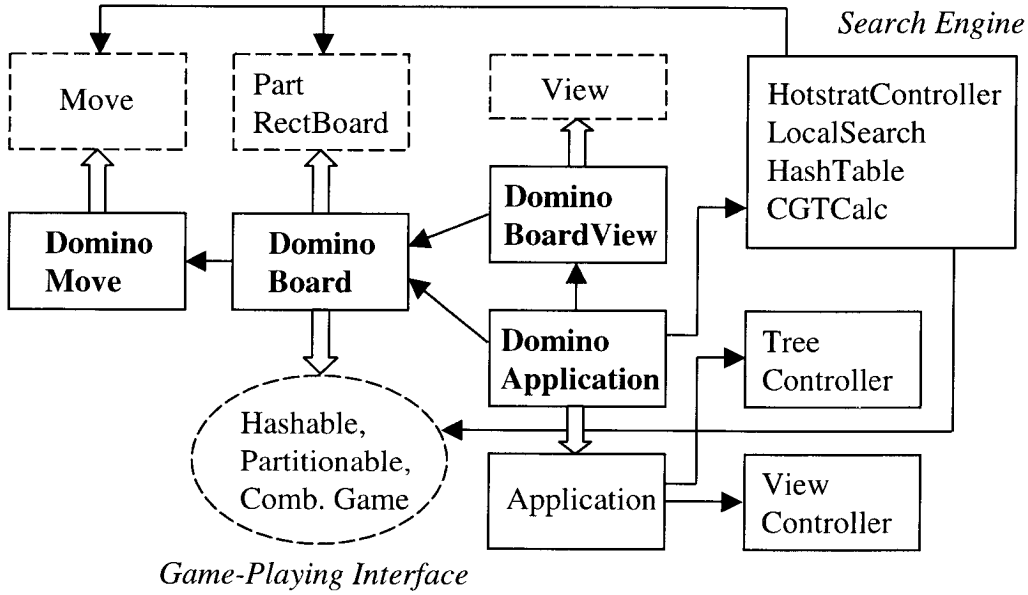


Figure 5.4: The class hierarchy of the Domineering application.

- class *DominoMove*: The data structure for a Domineering move.
- class *DominoBoard*: The game state that provides operations for executing and taking back moves, termination detection, move generation, hashing, and board decomposition.
- class *DominoBoardView*: The graphic representation of the board that enables the user to enter new moves.
- class *DominoMain*: The main application that controls the user interface, the game state, and the search engine.

The table shown in Figure 5.5 displays the "size" of the various Game Bench components, measured in numbers of lines of the source code.[1] The game specific part of the Domineering application amounts to only 11% of the total size of the source code. A strong argument in favor of the separation and generalization of game independent functionality.

	Game Bench	Domineering	Percentage
game kernel	1356	321	19%
search engine	3665	0	
user interface	1487	455	23%
total	6508	776	11%

Figure 5.5: The number of lines used by the source code of the various components, divided into a game independent *Game Bench* section and a game specific *Domineering* section. The third column displays the portion of the game specific code in relation to the total (game specific + game independent).

[1]The *C* source code of the *Gamesman's Toolkit* [52] is not taken into consideration in the total displayed for the search engine.

Chapter 6

Local Games with Global Threats

In this chapter we introduce a computation model for games that contain *global threats*. In sums of such games, a move in a local game can lead to an overall win in the sum of all games. This new approach allows us to deal with *entailing moves* (see [5], [13]) in the game tree of local games. In Section 6.1 we define global wins and global threats. In Section 6.2 we model global wins with the help of infinite *loopy games*. Loopy games, however, are difficult to calculate with. In Section 6.3 we present an algorithm for game tree search that avoids computing with loopy game values. Instead, it cuts off branches of the game tree that lead to global wins. In Chapter 7 we present an application of this computation model in king and pawn chess endgames.

6.1 Global Threats

The end of play in a sum of combinatorial games is determined by the *normal termination rule*: A player unable to move loses. Thus, in a sum of games, no single move or game can be decisive by itself. In this chapter, we extend this model by investigating a class of games where a move in a local game may lead to an overall win in the sum of all local games. We call such a move a *global threat* [35].

Definition 6.1 A *global threat* is a move in a local game G_i which has a decisive effect on the sum of which G_i is a part. Both players will prevent the opponent from playing such a move if possible.

Examples of possible global threats are moves that capture a vital opponent piece, such as *checkmate*[1]; moves that promote a piece to a much more powerful one (e.g promoting a checker to a king in a *checkers* variation analyzed by Berlekamp [2], or promoting a pawn to a queen in *chess*); or moves that "escape" in games where one side has to try to catch the other side's pieces like in the game *Fox and Geese* ("Winning Ways" [5], Chapter 20). Note that we do not define what happens if both players can execute a global threat in different local games. We focus on dealing with global threats in the analysis of local games independent of their context.

We are mainly interested in games where none of the players can win by executing a global threat if the opponent defends optimally. Such games are finally decided by the normal termination rule, and have finite combinatorial values.

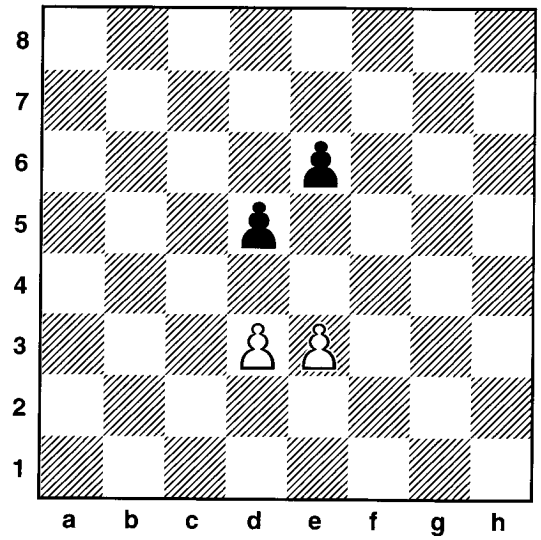


Figure 6.1: Global threats in a chess position: Both players have to prevent the opponent from promoting a pawn to a queen. Play will end when the pawns are blocked and neither player has any moves left.

In the chess example shown in Figure 6.1 we consider promoting a pawn to a queen a global threat. With best play from both sides, however, this position is decided by the normal termination rule. Either the pawns get completely blocked as for instance in the line 1.e3–e4 d5–d4 2.e4–e5 or we

[1]Although checkmate does not actually capture the enemy king, it creates the unstoppable threat to do so.

arrive at a position of *mutual Zugzwang* as after 1.d3–d4 when any move would allow the opponent to finally promote a queen.

With respect to chess endgames, Elkies [13] writes: "The analysis of such positions is complicated by the possibility of pawn trades which involve entailing moves: an attacked pawn must in general be immediately defended, and a pawn capture parried at once with a recapture. Still we can assign standard CGT values to many positions ... in which each entailing line is dominated by a non-entailing one." In this chapter, we introduce algorithms that solve this problem in general.

6.2 A CGT Model based on Loopy Games

A natural way to model global threats in combinatorial game theory is to use the loopy games *on* and *off* which are greater, respectively smaller than any finite game G (see Section 2.2.7). A global threat executed by *Left* is represented by the value *on*, a global threat executed by *Right* by the value *off*.

$$\forall G \in \{\text{finite games}\} : \mathit{off} < G < \mathit{on} \tag{6.1}$$

Now we can analyze the chess position given in Figure 6.1. For instance, White's move e3–e4 leads to a symmetrical position where both players have the choice either to capture or to push the more advanced pawn. We compute this position's value as $\{*, \{\mathit{on} \mid *\} \mid *, \{* \mid \mathit{off}\}\}$. White's move d3–d4 and Black's move e6–e5 both lead to zero positions. It turns out that both other moves (White's e3–e4 and Black's d5–d4) are *reversible* (see Section 2.2.5), and the resulting game value is $G = \{0 \mid 0\} = *$, a *finite* value.

6.2.1 Implementation Issues

Modeling global wins with loopy games works fine in theory, but there are practical problems. We map finite combinatorial games "one to one" to data structures by their inductive definition $G = \{G^L \mid G^R\}$ with the basis $0 = \{\,|\,\}$. When following a path of left and right options, we are sure that finally the zero game will be reached. Many algorithms that work on combinatorial games are based on their inductive nature. Obviously, loopy games do not fit into this computation model. Either we must extend it in order to handle loopy games, or we must avoid loopy games in our computations.

In the next section, we pursue the second way. We present a computation model for local games with global threats based on cutoffs in the game tree:

rather than modeling the situation that arises after a global threat is played, we exclude these situations from the game tree.

6.3 A Computation Model based on Cutoffs in the Game Tree

In this section, we present a computation model that cuts off the branches of the game tree that lead to the execution of global threats. The model is based on the following two lemmas:

1. If a player has the chance to execute a global threat, he will always do so.

2. Any move to a position in which the opponent can play a global threat is "bad". Such a move need not be considered when evaluating a player's options.

Both lemmas follow from the rules for simplifying games (see Section 2.2.5). The first one is easily deduced. For any game G, the equation $\mathit{off} \leq G \leq \mathit{on}$ implies that a global threat always *dominates* any other move. The second lemma is deduced from the rule of replacing reversible moves. If in the game $G = \{G^L \mid G^R\}$, *Left* plays a move to G^{L_i} which contains a right move to *off*, then *Left's* move to G^{L_i} is *reversible* (as $G \geq \mathit{off}$) and is replaced by all left options of *off*. As *off* has no left options, *Left's* move to G^{L_i} is simply omitted. The same holds of course for a right move to a game G^{R_j} that contains a move to *on*.

The chess position in Figure 6.2 illustrates this. Both players have exactly one move which leads to a position where the two pawns attack each other. The player who captures his opponent's pawn will go on to promote his own pawn. In the context of king and pawn endgames, we consider promoting a pawn to be globally winning[2]. The value of this position is computed as $G = \{\{\mathit{on} \mid \mathit{off}\} \mid \{\mathit{on} \mid \mathit{off}\}\}$ which simplifies to $\{ \mid \} = 0$.

The normal approach to compute the value of game G is to produce its game tree up to the terminal positions *on* and *off*, and then back up these values to the root. Instead, based on lemma 2, we can immediately cut off both players' moves as they allow the opponent to execute a global threat and we already know that they are reversible (see the left side of Figure 6.2). This directly leads to the same value $G = \{ \mid \} = 0$. There are two evident advantages of this approach. First, we avoid most calculations with loopy

[2]This is true for a vast majority of pawn endgames, and we limit our attention to these.

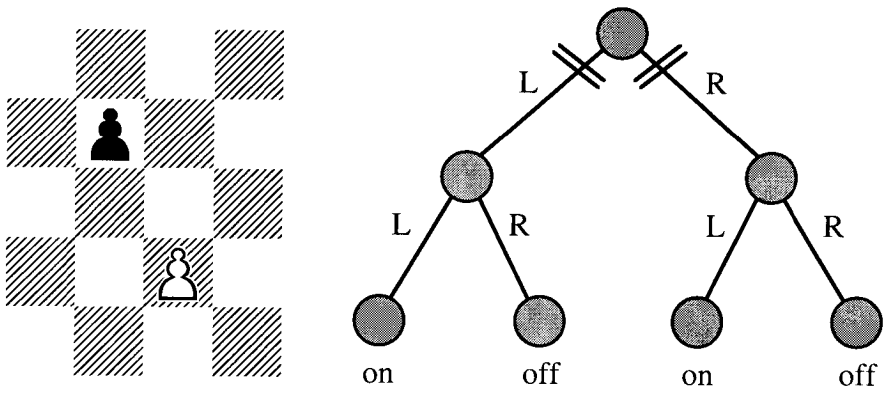


Figure 6.2: A game of value zero. Both players' moves lead to immediate opponent wins. This is equivalent to having no moves at all. $G = \{\{on \mid off\} \mid \{on \mid off\}\} = \{ \mid \} = 0$

games. In this simple example, we do not have to deal with loopy games at all. And second, thanks to the cutoffs, we minimize the number of nodes to be searched in the game tree.

6.3.1 An Algorithm for Evaluating Local Games with Global Threats

Now we are ready to formulate an algorithm for evaluating local games with global threats. Similar to local search (see Section 2.3.2), we consider both players' options in each position. Additionally, however, we make use of the information who made the last move. This enables to perform the above described cutoffs of global threats. It might seem unusual to make use of to-play information in combinatorial game tree search, but this also occurs implicitly in conventional CGT search. The same rule of replacing reversible options that allows to cutoff the game tree is based on "good replies to an opponent's move", thus also makes use of to-play information.

Result Types

In contrast to finite combinatorial games, the value of a game that contains global threats might be *on* or *off* i.e. a forced global win for one of the players. This is the case if a player cannot prevent his opponent from finally playing a global threat no matter how he defends (see Figure 6.3). We define the following *result types* of local games:

- Type *win*: If the players move alternately including the right to pass, *Left* will win by executing a global threat no matter how *Right* defends and no matter who starts.

- Type *loss*: If the players move alternately including the right to pass, *Right* will win by executing a global threat no matter how *Left* defends and no matter who starts. A game of result type *loss* is smaller than any finite combinatorial game G.

- Type *CGT*: None of the players can force a win by global threat. In this case, we can compute a finite combinatorial value $G = \{G^L \mid G^R\}$ for the actual game position. These are the games that we are most interested in.

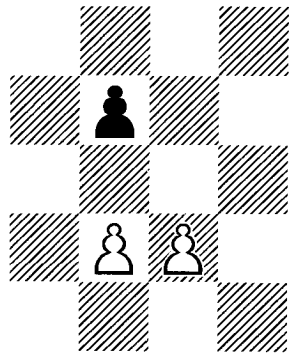


Figure 6.3: A game of result type *win*. White will always promote a pawn no matter who starts. The possibility to pass (i.e. play in another local game) does not help Black either.

Game Tree Search

In every position, we recursively evaluate both players' options. As the possible result types are ordered ($win > CGT > loss$ from *Left's* point of view and $loss > CGT > win$ from *Right's* point of view), we can determine the best result types both players can get if they have the move. If a player's best result type is *CGT*, we also compute his combinatorial game options (G^L respectively G^R). Combined with the information on who is to play, we compute the result type of the actual position as shown in Figure 6.4. In

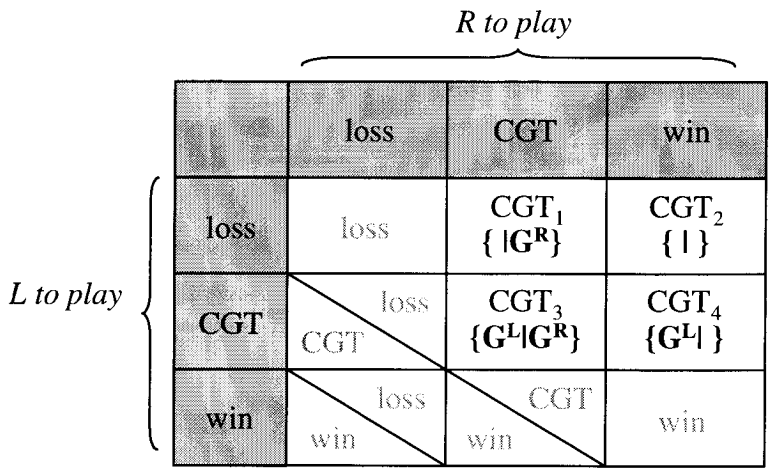


Figure 6.4: Result types of *global threats evaluation*: the table indicates the result type of a local game depending on the best result types of *Left's* (rows) and *Right's* (columns) options The split entries show the result types for *Left* to play (lower left) and *Right* to play (upper right).

case the actual result type is *CGT*, we also compute the combinatorial game value of the position.

In the four highlighted cases, we compute a finite combinatorial value for the actual game position.

- CGT_1
  All *left* options lead to games of type *loss* while *Right* has at least one move that leads to a finite combinatorial game. As *Left* has no good moves (i.e. moves that do not allow the opponent to force the execution of a global threat), the value of the actual position is $G = \{ \mid G^R\}$.

- CGT_2
  Neither player has any good moves. The actual position is a *mutual Zugzwang*. $G = \{ \mid \} = 0$. We have already seen an example of such a situation in Figure 6.2.

- CGT_3
  Both players' best options all lead to finite values. The actual game value is $G = \{G^L \mid G^R\}$.

- CGT_4
  In contrast to the first case, *Right* has no good move while *Left* has at least one good option. $G = \{G^L \mid \}$.

If both players' best result types are *win* (resp. *loss*), the result type of the actual game position is of course also *win* (resp. *loss*). In the remaining three cases, the result type of the actual position depends on who has the right to move. If the player to move can move to a winning position, he will of course do so and the result type is determined as a win in his favor.

Of special interest are the positions where the player to move has one or more moves that lead to games of type CGT while his opponent would be winning if he was to play. These are the only cases where the loopy games *on* and *off* occur in our combinatorial game values which are either $\{on \mid G^R\}$ or $\{G^L \mid off\}$). Fortunately, *on* and *off* only appear as *threats*. For example, if *Left* plays a move to a position of value $\{on \mid G^R\}$, then *Right* immediately has to move to one of the options in G^R as *Left threatens* to move to *on*.

We conclude that if a game with global threats has a finite value, the algorithm computes it without evaluating loopy games. These are the games that interest us the most. If a game does not have a finite value, the algorithm computes the values of its finite options and the information who wins depending on who moves first.

6.3.2 Example: Application to a Chess Position

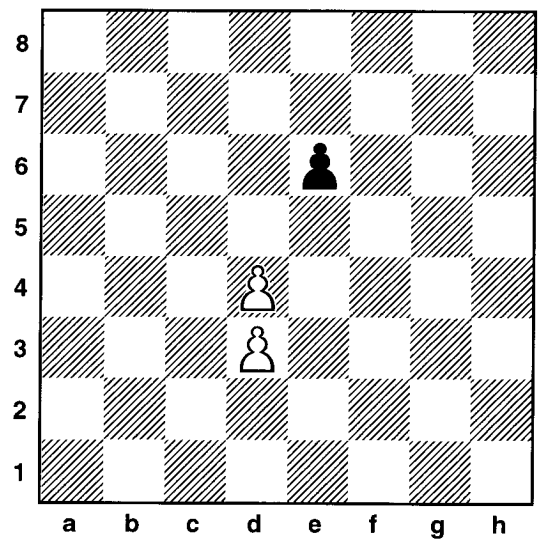


Figure 6.5: An example of global threats evaluation: the value of this position is $G = \{\{on \mid *\} \mid \} = 1$.

The example shown in Figure 6.5 illustrates how the algorithm works.

Either one of the players manages to promote a pawn (result type *win* or *loss*), or the pawns get blocked (result type CGT). We start by analyzing both players' options in the current position. The resulting game tree is shown in Figure 6.6.

- White (*Left*) to play has one move, d4–d5. If he had the move again, he would play d5×e6 winning immediately (result type *win*). If, on the other hand, Black was to play in the position after d4–d5, he would play e6×d5 leading to a position of result type CGT and of value $\{0 \mid 0\} = *$. As we know that Black is to play, the result type of the position after d4–d5 is CGT, and its value is $\{on \mid *\}$. This is White's only option, thus $G^L = \{\{on \mid *\}\}$.

- Black to play also has only one move, e6–e5. In the resulting position, White to play has the move d4×e5 leading to result type *win*. Black to move again, on the other hand, plays e5×d4 leading to result type CGT and value $\{ \mid \} = 0$. We know that after e6–e5 White has the move, thus Black's best result type is *win*, a loss for him.

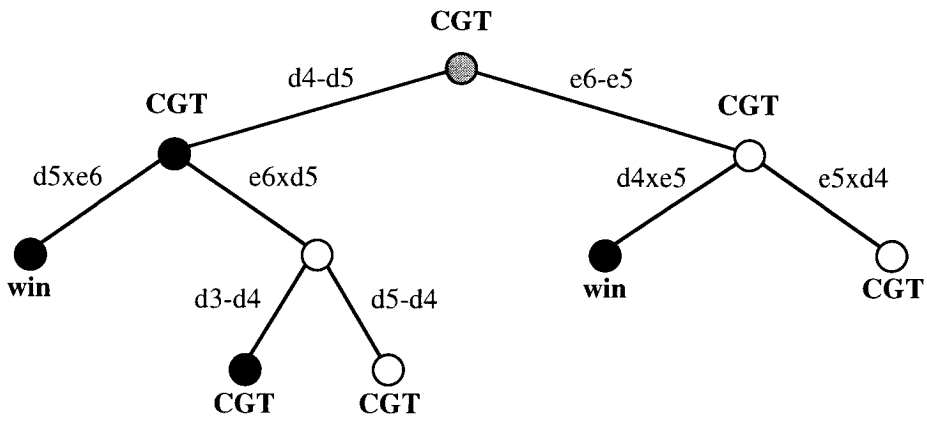


Figure 6.6: Game tree of the example shown in Figure 6.5: the result types at the leaves are backed up in order to compute the result types of the inner nodes.

According to Figure 6.4 (the entry labeled 4), the value of the current position is computed as $G = \{\{on \mid *\} \mid \}$ which simplifies to $G = \{0 \mid \} = 1$ as *Left's* option is reversible.

6.3.3 Implementation

The function *GTSearch* searches the game tree depth first and computes result types and combinatorial game values of local games that contain global threats. Its specification is

- **in**:
  - *toplay*: the player (constants *kWhite*, *kBlack*, *kNoPlayer*) who has the move. In the starting position, the *root* of the game tree, *kNoPlayer* is passed. At all other nodes of the tree, the right to move is determined.
- **out**:
  - *return value*: the result type (constants *kWin*, *kLoss* or *kCGT*) of the current position.
  - *value*: the combinatorial game value of the current position. This value is "valid" only in case *kCGT* is returned.

The algorithm performs the following steps (numbers refer to comments in the code):

1. Check termination:
   determine if the actual position is a global win for one of the players. If so, we are finished and return *kWin* resp. *kLoss*.
2. Recursively evaluate *Left's* options:
   We store the best result type ($kWin > kCGT > kLoss$) achieved so far in the variable *bestL*. For every option of type *kCGT*, we include its combinatorial value in the set G^L. If we find an option leading to result type *kWin*, we can skip the remaining options (cutoff!). (The *forsome* statement has the same functionality as *forall*, but it allows to exit the loop with a *break* statement.)
3. Recursively evaluate *Right's* options:
   Analogously to step 2 we compute the values *bestR* and G^R.
4. Compose the result:
   According to the table of Figure 6.4, we compute the result type of the actual position. In case the result type is *kCGT*, we also compute the combinatorial game value $\{G^L \mid G^R\}$ and return it in the out-parameter *value*.

```
function GTSearch(toplay: TPlayer; var value: TGameValue): TResultType;
begin
  if GlobalWin(kLeft) then return kWin; endif; /* 1 */
  if GlobalWin(kRight) then return kLoss; endif;
  G^L ← { }; bestL ← kLoss;
  forsome left moves m do /* 2 */
    ExecMove(m);
    res ← GTSearch(kRight, val);
    UndoLastMove();
    if res = kWin then bestL ← kWin; break endif; /* cutoff! */
    if res = kCGT then bestL ← kCGT; G^L ← G^L∪ val endif;
  endfor;
  G^R ← { }; bestR ← kWin;
  forsome right moves m do /* 3 */
    ExecMove(m);
    res ← GTSearch(kLeft, val);
    UndoLastMove();
    if res = kLoss then bestR ← kLoss; break endif; /* cutoff! */
    if res = kCGT then bestR ← kCGT; G^R ← G^R∪ val endif;
  endfor;
  return ComposeResult(bestL, bestR, value, G^L, G^R, toplay); /* 4 */
end GTSearch;
```

Chapter 7

Combinatorial Chess Endgames

In this chapter, we present *king* and *pawn* chess endgames as an application of the global threats model introduced in Chapter 6. Elkies [13] shows that in situations of mutual Zugzwang, these endgames often admit CGT solutions. Section 7.1 gives an overview of his work. As an open problem, Elkies mentions the entailing nature of captures and threats to capture. In Section 7.2, we show how, using the global threats model, we can compute values of complex pawn structures whose analysis contains entailing moves such as captures and promotions. In Section 7.3, we discuss the application of decomposition search [41] to king and pawn endgames. We compare the divide and conquer approach to standard minimax search and to the methods of classical chess theory. Finally, we present a selection of endgames from tournament practice with CGT solutions in Section 7.3.5.

7.1 Divide and Conquer in King and Pawn Endgames

In general, Combinatorial Game Theory does not apply to chess, mainly because the 8×8 chess board is too small (or too crowded) to decompose into independent subgames. Moreover, the long range pieces queen, rook and bishop can traverse the whole board in one move, again preventing a decomposition. In some *endgames*, however, with the long range pieces gone, we can identify independent subgames on different sides of the board. In this chapter, we focus on *KP endgames* where each side only has his *king* and a number of *pawns*. The king has the ability to go from one end of the board to the other, although slowly. In situations of *Zugzwang*, however, a king move leads to a disadvantage big enough to lose the game.

7.1.1 Mutual Zugzwang

In a situation of *mutual Zugzwang* (mZZ), both players would rather pass than make a move. In the position shown in Figure 7.1, a king move by either side results in the loss of a pawn and consequently in the loss of the game. mZZ is modeled by the combinatorial game $0 = \{ \mid \}$ where none of the players can move. Thus, both players will make pawn moves on the a, b, and h-file until the pawns are blocked and the side to move loses.

Positions of mutual Zugzwang, although rare[1], have always interested chess analysts and are found in many books on endgame theory. Elkies [13] shows that combinatorial game values also occur in chess, and that it is possible to solve positions of mutual Zugzwang with the help of CGT. We give a summary of his analysis of the position shown in Figure 7.1 (Sveda - Sika, Brno 1929).

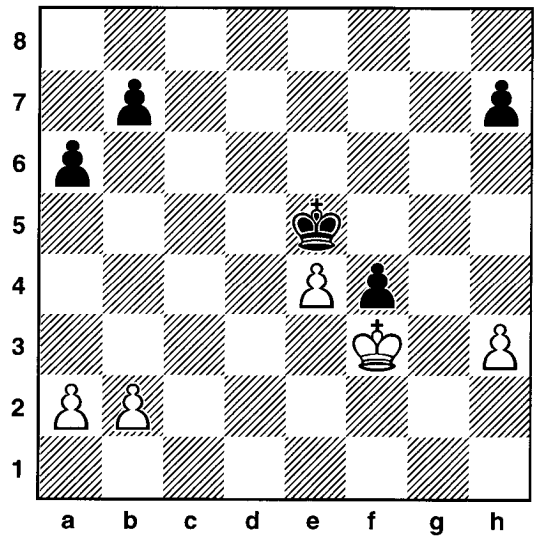


Figure 7.1: Mutual Zugzwang: Sveda - Sika, Brno 1929. The player who first moves his king loses. Both players move their queenside and kingside pawns until one of them must give way. The value of this position is $G = \downarrow *$, a first player win.

- The *center* with the two kings and a pawn each is a *mutual Zugzwang.* The player to move has to give up the defense of his pawn and loses. Its value is $G_1 = \{ \mid \} = 0$.

[1] rare in tournament play that is, but very popular in the world of chess problems and endgame studies.

- The *kingside* (h-file) favors *Black* who still has the choice of moving his pawn by one or by two squares. The value of the kingside is $G_2 = \{\downarrow | 0\} = \downarrow\downarrow *$. (In all chess examples, we define White playing *Left* and Black playing *Right.*)

- The *queenside* (a and b-files) is the most complicated of the three subgames. White who has both pawns on their original squares is slightly favored. The queenside's value is $G_3 = \{0 \mid *\} = \uparrow$.

The whole game is of value $G = G_1 + G_2 + G_3 = \downarrow *$, a first player win. White to move wins by playing 1.h3–h4! moving in G_2 to ${G_2}^L = \downarrow$. The resulting sum is $G_1 + {G_2}^L + G_3 = 0 + \downarrow + \uparrow = 0$, thus Black to play now is in Zugzwang. Black to play first, on the other hand, plays 1. ... a6–a5! moving in G_3 to ${G_3}^R = *$. The resulting sum $G_1 + G_2 + {G_3}^R = 0 + \downarrow\downarrow * + * = \downarrow\downarrow$ is negative, therefore won for Black now matter who is to play.

7.1.2 Analysis of Pawn Structures

In the Sveda-Sika game, we have seen that it is decisive to make the last pawn move when all other pieces are bound by Zugzwang. This subgame of chess where both players only have pawns and try to blockade the opponent can be mapped to a combinatorial game as defined by Conway [11]. Furthermore, as pawns, except for capturing moves, cannot change their files (columns), chunks of pawns on different locations of the chess board do not interfere and can therefore be considered *independent* games.

Elkies has constructed pawn structures that represent all kinds of combinatorial values (*infinitesimals*, *integers*, *fractions*, *switches* ...) and has systematically analyzed some simple structures by hand such as the one against one situations shown in Figure 7.2.

When two pawns move towards each other on the same file, the value of the position is either $G_{even} = \{* \mid *\} = 0$ with an even number of squares between the pawns or $G_{odd} = \{0 \mid 0\} = *$ with an odd number of squares between the pawns (Figure 7.2, positions 1 and 2), unless at least one of the players still has the right to make a double step with his pawn. This choice is of course an advantage for the player who still has it: The value of position 3 is $\{0, * \mid *\} = \uparrow$, a positive value, while position 4 $\{\downarrow | \ 0, *\} = \downarrow\downarrow *$ is a negative game. If both pawns can make a double step, the value of the position is 0, except for very small boards. If the pawns are separated by one row only, the resulting value is $\{0 \mid 0\} = *$. If the pawns are separated by two rows, we get $\{0, * \mid 0, *\} = *2$, a game equivalent to a Nim heap of size 2.

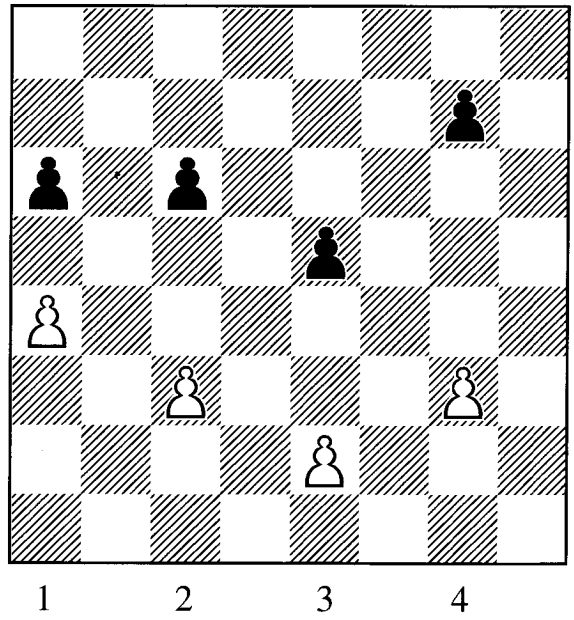


Figure 7.2: Opposite pawns on the same file: position 1 = $*$, position 2 = 0, position 3 = $\uparrow$, position 4 = $\downarrow\downarrow *$.

Given a white pawn on its starting square and a black pawn that no longer has the option of making a double step separated from the white pawn by s squares on an "arbitrarily large" chess board, the game value G of the position is

$$G = \begin{cases} 0 & s = 0, \\ * & s = 1, \\ (s-1) \cdot \uparrow & s > 1, s \text{ is even}, \\ (s-1) \cdot \uparrow + * & s > 1, s \text{ is odd}. \end{cases} \tag{7.1}$$

More complex to analyze, however, are positions with more than one pawn on each side when captures and promotions are possible. Addressing *open problems*, Elkies [13] writes: "In other directions, one might also hope for a more systematic CGT-style treatment of en passant captures and entailing chess moves such as checks, captures entailing recapture, and threats to capture ... ".

7.2 Global Threats in Local Chess Games

Let's have a look at more complex pawn structures. Positions like the one shown in Figure 7.3 contain *entailing moves* such as attacks and captures which usually force an immediate reaction from the opponent.

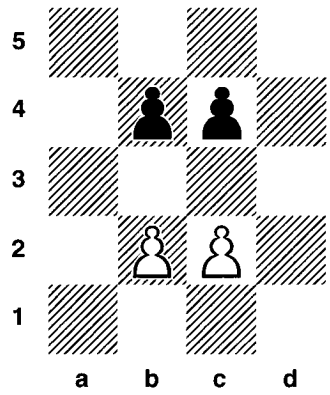


Figure 7.3: A two vs. two pawn structure of value 0.

This position is a mutual Zugzwang, $G = \{ \mid \} = 0$. The player who starts gets blocked if his opponent does not capture but pushes his attacked pawn forward, for example 1.b2–b3 c4–c3. The analysis of such positions is complicated by attacks and captures which involve entailing moves. In our example, Black might answer 1.b2–b3 with c4×b3 forcing White to immediately recapture with 2.c2×b3. The key to the analysis of these structures is that not the capturing move itself is an "entailing factor", but the *global threat* to bring a pawn to the last rank where it is promoted to a queen. As already stated in Chapter 6, we consider the promotion of a pawn a global win in king and pawn endgames.

Using the computation model presented in Chapter 6 we compute the value of the position shown in Figure 7.3 as:

$$G = \{\{on \mid 0, \{0 \mid off\}\} \mid \{0, \{on \mid 0\} \mid off\}\} = 0$$

Any move by one of the players gives the opponent the choice either to move to 0 by pushing his attacked pawn, or to set up a global threat by capturing. Note that neither option dominates the other. The original game G however simplifies to $G = 0$ as the second player always has a move to 0. The capturing option turns out to be *reversible*. Although some lines of play lead to global wins, the value of this pawn structure is finite. Neither

player can force a win. In this simple example, the rule that a capture must be answered with an immediate recapture applies. In the next one (Figure 7.4) it does not.

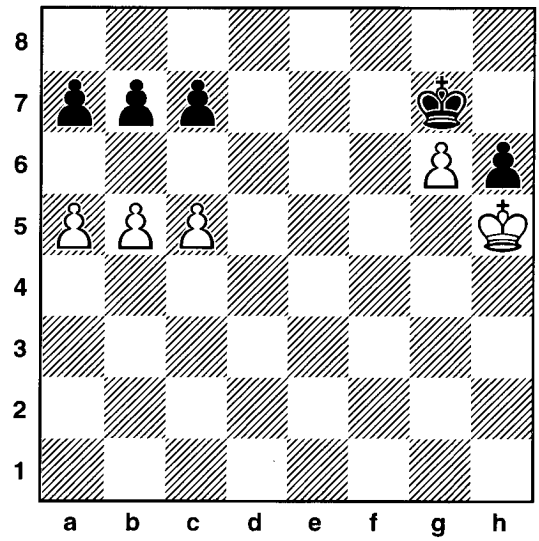


Figure 7.4: The *breakthrough*: White to play sacrifices two pawns in order to promote the third one.

The kingside structure with king and pawn each is "the same" mutual Zugzwang that we have already seen in the Sveda-Sika game. On the queenside, however, thanks to his far advanced pawns, White to move forces a global win by sacrificing two pawns in order to promote the third one. After 1.b5–b6! a7×b6 (c7×b6 2.a5–a6! etc.) 2.c5–c6! b7×c6 3.a5–a6! the a-pawn is unstoppable. In chess literature, this maneuver is known as a *breakthrough*. Black to play, on the other hand, cannot do the same, as White would be much faster promoting one of his pawns. Black's only move that does not allow White to win by global threat is 1. ... b7–b6 leading to the value $G^{L_1} = \{0, \{on \mid 0\} \mid off\}$. After 1. ... b7–b6 White has a choice of two symmetrical lines, for instance 2.a5×b6! a7×b6! and now 3.c5–c6 leaves Black in Zugzwang. The value of the whole game is $G = \{on \mid \{0, \{on \mid 0\} \mid off\}\}$.

We conclude that the reason why captures and attacks often are *entailing moves* is that they usually lead to a *global threat* in form of a promotion. In many cases, however, especially if both sides have an equal number of pawns, none of the players can force the promotion of a pawn. In this case, we can compute a *finite* combinatorial game value for the given pawn structure.

7.3 Results and Discussion

7.3.1 Decomposition Search in Chess Endgames

In Section 2.3, we introduced *decomposition search* as a method of applying combinatorial game theory to games that decompose into independent subgames. It includes the following steps:

1. *Decomposition*: divide the game into independent components (local games).

2. *Local Search*: compute the combinatorial values of the local games.

3. *Evaluation*: compute a result (sum game value, move decision ...) based on local game information.

The process of identifying the local games in a chess position involves two main steps, dividing the pawn structure and detecting situations of mutual Zugzwang.

Dividing the Pawn Structure

Dividing the pawn structure into independent chunks is the easier of the two steps. It does not involve any search, just geometry:

- Two pawns of opposite color *interact* if they have not passed each other, that is, if the rank[2] of the white pawn is smaller than the rank of the black pawn.

- Two pawns of any color *interact* if they are on the same or on neighboring files.

For pawns of opposite color to interact, both rules must apply, for pawns of the same color only the second one. Interaction between pawns is transitive. The independent pawn chunks on the chess board correspond to the equivalence classes of the relation of interaction.

[2]In chess terminology we use *rank* for rows and *file* for columns of the chess board.

Detection of Mutual Zugzwang

We must analyze the kings' positions[3] in order to find out whether there is a situation of *mutual Zugzwang*. If both kings are *bound* to a pawn chunk, we can apply CGT. Usually such a situation is similar to the one in the Sveda-Sika game (Figure 7.1), but Elkies has shown many different positions where the kings or other pieces are bound.

The game examples presented in Section 7.3.5 were computed using an "oracle" in the form of a human expert to detect the Zugzwang positions. As we have seen in Elkies' work, this approach is successful in analyzing KP endgames, as well as in composing endgame studies and problems. By means of CGT, chess positions that do not seem to fit in any strategic scheme can be mathematically solved and explained.

In order to automatically detect Zugzwang situations, we propose a *locally restricted* minimax search. If no player can move his king without worsening his position (for example losing a pawn), we have detected a mZZ. The local search can be exhaustive or limited by time or search depth. For example, in the Sveda-Sika game (Figure 7.1), even a two-ply search combined with material evaluation will show that it is bad for either player to move his king.

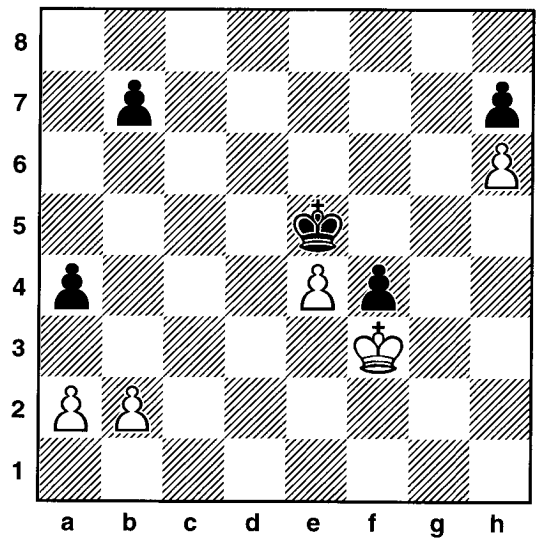


Figure 7.5: Sveda - Sika revised: Black to play tries to break out of the mZZ position with Ke5-f6.

[3]and the positions of all other pieces if we want to extend this method to general chess endgames.

However, it is important to note that no matter how a Zugzwang is detected, the conclusion that it will be constant is of a heuristic nature. In fact, the decomposition of a chess position is always a *heuristic decomposition.*

Let's again have a look at the Sveda-Sika game: If White starts, we reach the position shown in Figure 7.5 after the moves 1.h3–h4! a6–a5 2.h4–h5! a5–a4 3.h5–h6! Black could now consider abandoning his f4-pawn and instead attacking the white pawn on h6. After the moves 3...♔e5–f6 4.♔f3×f4 ♔f6–g6 5.♔f4–e5 ♔g6×h6, material is balanced, but White easily wins with 6.♔e5–f6 when his e-pawn is unstoppable. Thus, our initial assumption was correct: the side to move its king first loses. But in other cases, breaking out of the Zugzwang might upset the decomposition and thus the combinatorial evaluation of the position.

7.3.2 Divide and Conquer vs. Full Width Search

In the following game example, we compare decomposition search with minimax evaluation as used by chess playing programs.

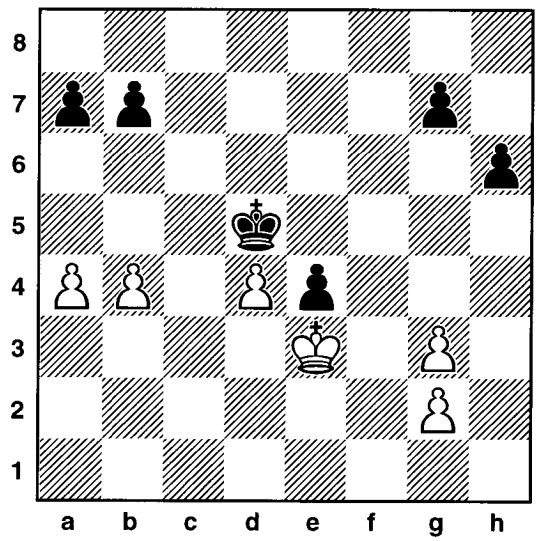


Figure 7.6: A first player win: Popov - Dankov, Albena 1978. The queenside and center are both of value 0. On the kingside, the first player to move forces his opponent into a Zugzwang position. The kingside and therefore the game is of value $G = \{0 \mid -1\}$.

Figure 7.6 shows a position from the game Popov vs. Dankov, Albena 1978. The relative position of the two kings is the same as in the Sveda-Sika

game. The player who moves his king first loses his central pawn which in this case is always decisive. The game is decided by the local games on the queenside and kingside.

- The queenside is a game of value $G_{QS} = 0$. With the white pawns advanced to the fourth rank, Black gets no advantage from the double step option. The player to move is immediately blocked, for example 1.a4–a5 a7–a6!.
- The kingside is more complex. Black to play gains a considerable advantage with 1. ... h6–h5. In fact, due to White's option to sacrifice a pawn for a move with g3–g4, it "only" leads to a value of -1. White to play has only one move. Thanks to the possibility of sacrificing one of the doubled pawns for a move, it leads to a position of value 0. The main line runs 1.g3–g4 g7–g6 2.g4–g5 h6×g5 3.g2–g4 and a zero position is reached. The value of the kingside is $G_{KS} = \{0 \mid -1\}$.

The sum $G_{QS} + G_{KS} = \{0 \mid -1\}$ is a first player win. The following results are computed on a PC (466 MHz Intel Celeron, 128 MByte Ram) running Linux.

- Decomposition search requires a total of less than 1000 evaluated nodes to compute the values of the kingside and the queenside.
- Combinatorial evaluation of the combined (kingside and queenside without divide and conquer) pawn structure yields the same result, but takes much longer. Almost 200,000 nodes need to be evaluated.
- In order to illustrate the complexity of a full-width alpha-beta search, we ran *Crafty* [27] on the game position with White to move. Only after evaluating more than $2 \cdot 10^9$ nodes, the program indicated 1.g3–g4 leading to a white advantage.

Note that both computed results are heuristic: the result of decomposition search due to the heuristic decomposition of the chess board, the result of alpha-beta search due to the heuristic evaluation of non-terminal positions.

7.3.3 KP Endgames involving Zugzwang in Chess Literature

Positions of mutual Zugzwang where pawns try to block each other are popular, but only vaguely described in chess literature. The units of calculation

used by the authors are *spare tempi* which correspond to extra moves or *integers* in CGT. But as we have seen, the game values are often *rationals*, *infinitesimals* and sums of those. It is therefore not surprising that the authors cannot give useful instructions on how to compute spare tempi. Statements like "with pawns on one or two files, computing spare tempi is not too difficult ... " (Awerbach [1][4]) or "by accurate play White makes sure it is Black who first runs out of moves ... " (Müller [38]) illustrate this.

On the other hand, the authors recognize the possible application of divide and conquer. Speelman [49], for instance, discusses certain pawn structures separately from any actual game position. The analysis of sums of such games is quite obscure though. Speelman, on the Sveda-Sika game (Figure 7.1) writes: "In fact, whoever is to move wins by first forcing the kings and centre pawns into zugzwang and then 'correcting' the situation on the side of the board on which he is at a disadvantage." Although he correctly assesses the position as a first player win, he does not define the terms "correct the situation" and " disadvantage". Nevertheless, strong human chess players handle Zugzwang positions well as the examples in Section 7.3.5 show.

7.3.4 Summary and Conclusions

As Elkies [13] has shown, in chess endgames, especially in king and pawn endgames, the presence of mutual Zugzwang can lead to positions where CGT applies. The fight of opposite pawns trying to block each other can be mapped to a combinatorial game as defined by Conway [11]. In a sum game, the local promotion of a pawn to a queen leads to an immediate overall win. With the help of the *global threats* model presented in Chapter 6, however, we can compute game values of complex pawn structures that contain *entailing moves* such as captures and promotions.

In combination with an algorithm to detect Zugzwang positions, such as a local minimax search, a calculator that computes combinatorial values of pawn structures implements *decomposition search* of king and pawn endgames. The range of chess positions where divide and conquer applies is certainly too small for a stand-alone chess playing program based on decomposition search. On the other hand, we see useful applications of a decomposition search component as a part of a chess playing program.

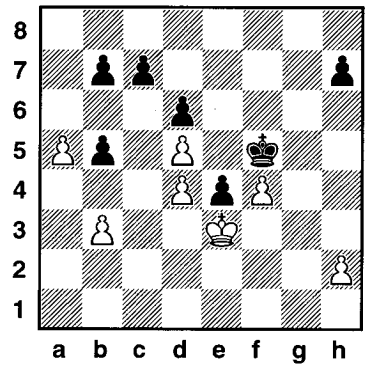
Further, a program that computes combinatorial game values of pawn structures and Zugzwang positions is a useful tool for the chess analyst as combinatorial game theory can provide mathematical solutions to chess positions that classical chess theory fails to explain in detail (see Section 7.3.3).

[4]translated from German

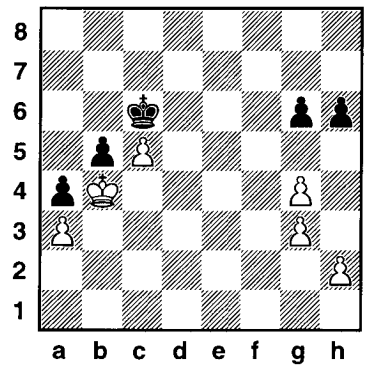
7.3.5 Games Selection

In this section, we present a selection of twelve endgames, some of them played by strong grandmasters, that can be solved by CGT. The level of difficulty ascends from "fairly easy" (positions 1 and 2) to "rather difficult" (positions 10–12). Chess players may find the examples a good test for their endgame abilities.

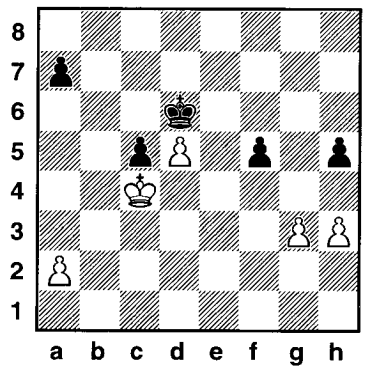
The solutions were computed with a program based on the Game Bench framework. The table presented in Figure 7.7 shows how many nodes in the game tree had to be evaluated to compute the results.



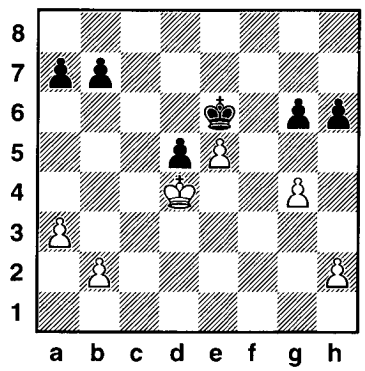
1. Velimirovic - Smejkal
Rio de Janeiro 1979.
White to play.



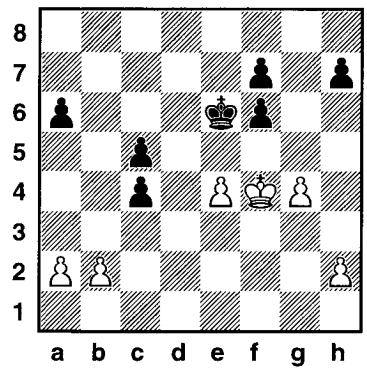
2. Turov - Jagupov
Rostov 1993.
White to play.



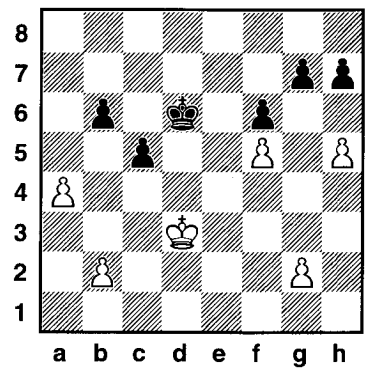
3. Spraggett - Eslon
Zaragoza 1996.
White to play.



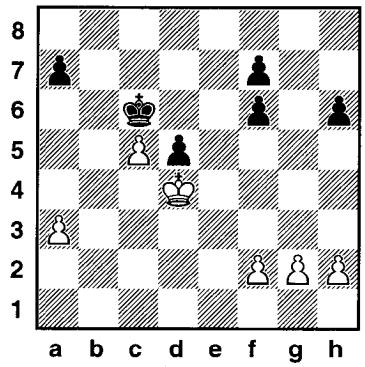
4. Paoli - Michel
Vienna 1950.
White to play.



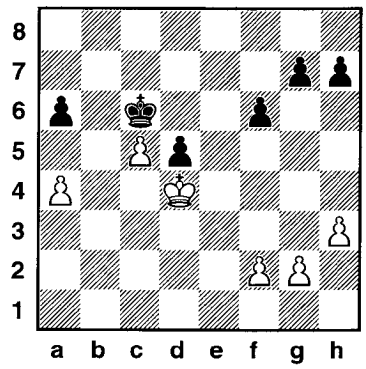
5. Bronstein - Rajna
Budapest 1977.
Black to play.



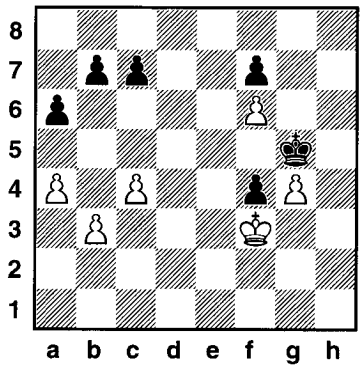
6. Thomas - Maroczy
Nice 1930.
Black to play.



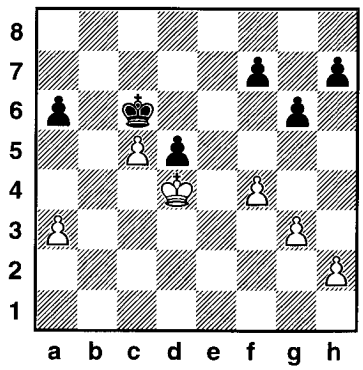
7. Pirc - Kolski
Lodz 1938.
Black to play.



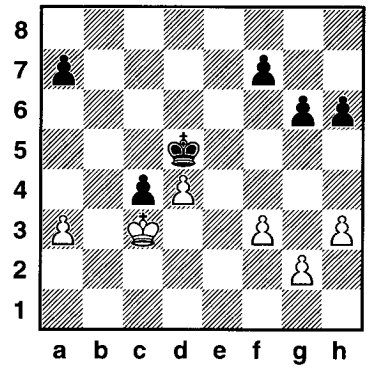
8. Ranits - Wirius
Austria 1998.
White to play.



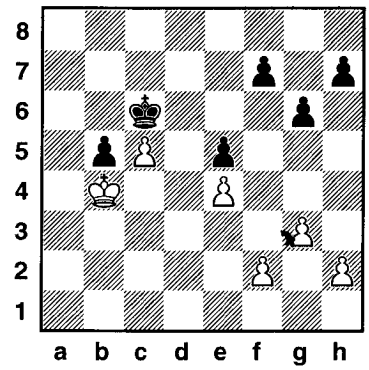
9. Nehlert - Precour
Baden 1992.
White to play.



10. Jacek - Miler
Czech Rep. 1996.
Black to play.



11. Schüssler - King
Gausdal 1990.
White to play.



12. Scheske - Kiefer
Bingen 1998.
Black to play.

Solutions

1. **Velimirovic - Smejkal, Rio de Janeiro 1979**
   The position looks more complex than it actually is. The kings are in the same mutual Zugzwang that we already know from previous examples. The h-file has value 0, the player who moves first gets blocked. On the queenside only the white pawn on b3 and the black pawn on b5 can move. All other pawn moves result in immediate losses e.g. 1. ... c7–c6? 2.d5×c6! b7×c6 3.a5 a6 and the a-pawn promotes in another two moves. The value of the queenside and also of the whole game is $G = *$. The game concluded **1.b3–b4! h7–h5 2.h2–h4!** ♔**f5–g4** Black attacks the white h-pawn, but this comes too late. **3.**♔**e3×e4** ♔**g4×h4 4.**♔**e4–f3!** and Black resigned.

2. **Turov - Jagupov, Rostov 1993**
   The queenside with the two kings is a mutual Zugzwang. On the kingside, the extra pawn, although doubled, gives White the advantage. Its value is $G_{KS} = G = 2$. In the game, Black resigned after **1.h2–h3**. A possible continuation is 1. ... g6–g5 2.h3–h4 and each move by Black allows White to create a passed pawn.

3. **Spraggett - Eslon, Zaragoza 1996**.
   The pawns on the a-file and the configuration of the two kings are both of value 0. The outcome of the game is decided on the kingside. Black to play has the winning move 1. ... h5–h4, a *global threat*! White to play, on the other hand, moves to 0 with 1.h3–h4. The value of the kingside and of the whole game is $G_{KS} = G = \{0 \mid \text{off}\}$. Spraggett forced his opponent to resign after **1.h3–h4! a7–a6 2.a2–a3!**.

4. **Paoli - Michel, Vienna 1950**
With two configurations of 2 against 2 pawns, this position is more complex than the previous ones. Again, the kings are bound in a mutual Zugzwang. The queenside structure has already occurred in the Sveda-Sika game with colors reversed. Its value is $G_{QS} = \downarrow$. On the kingside, due to the double-step option, White has the advantage: he can move to 0 (h2–h4) while Black can't. The value of the kingside is $G_{KS} = \{0 \mid \{\uparrow, \{\text{on} \mid \uparrow\} \mid -2\}\}$. The sum game $G = \{0 \mid \{0, \{\text{on} \mid 0\} \mid -2 \downarrow\}\}$ is greater than zero, a win for White no matter who starts. Paoli found the correct moves over the board: **1.a3–a4! b7–b6 2.b2–b4 a7–a6 3.h2–h4 a6–a5 4.b4–b5! h6–h5 5.g4–g5** and Black resigned.

5. **Bronstein - Rajna, Budapest 1977**
In this position, the configuration of the kings is different. White wins if he can get his king to f5. If he has to go back, on the other hand, he only has a draw. Thus, we also have a mutual Zugzwang, but with only half a point at stake. On the kingside, White has two extra moves, $G_{KS} = 2$. On the queenside, thanks to his extra pawn, Black has the advantage, although it is not big enough to force a passed pawn. White, however, must by all means prevent the black a-pawn from reaching a3. The queenside value is $G_{QS} = -1 + \{0 \mid \text{tiny}\}$.[5] Overall, the position is a win for White. The game concluded **1. ... a6–a5 2.a2–a4 h7–h6 h2–h3 f6–f5** Desperation, but 3. ... c4–c3 4.b2×c3 c5–c4 5.h3–h4 also wins for White. **4.g4×f5+ ♔e6–d6 5.f5–f6 ♔d6–e6 6.e4–e5 ♔e6–d5 7.♔f4–f5 h6–h5 8.h3–h4** and Black resigned.

6. **Thomas - Maroczy, Nice 1930**
Here, White wins if his king gets to b5, whereas Black draws if White does not succeed in doing so. To achieve his goal, White needs two extra moves. The value of the kingside is $G_{KS} = 1$, thus with Black to move, White can just make it. In fact, in this position, Black resigned. A possible line of play would be **1. ... ♔d6–d5 2.b2–b3! ♔d5–d6 3.♔d3–c4! ♔d6–c6 4.g2–g3! h7–h6 5.g3–g4!** and White wins.

7. **Pirc - Kolski, Lodz 1938**
The situation on the queenside has occurred in the Sveda-Sika game. Its value is $G_{QS} = \downarrow\downarrow *$. On the kingside, White has a big advantage because of Black's crippled pawns, $G_{KS} = \{2 + 3 \,.\, \uparrow \mid \{1 + 3 \,.\, \uparrow \mid 0\}\}$[6] Here, having the move is favorable as the temperature of the kingside is

[5]The value *tiny* is the smallest positive value there is: tiny = $\{0 \mid \{0 \mid \text{off}\}\}$)

[6]The expression 3 . ↑ stands for ↑ + ↑ + ↑, a notation used in "Winning Ways" [5].

positive: $t_{KS} = 3/4$. However, the right to move does not save Black as his disadvantage is too big. **1. ... f6–f5 2.f2–f4** The move 2.g2–g3 was even stronger (f2–f4 leads to a kingside value of 1, g2–g3 to $1 + 3 \cdot \uparrow$), but the move played is good enough to win the game. **2. ... f7–f6 3.a3–a4 h6–h5 4.g2–g3** and Black resigned.

8. **Ranits - Wirius, Austria 1998**
The queenside has the value $G_{QS} = \{0 \mid 0\} = *$. The 3 against 3 structure on the kingside looks complex, but has a simple value, $G_{KS} = 0$. In the game, White found the winning moves. **1.a4–a5! h7–h5** A tougher defense was 1. ... f6–f5 2.h3–h4! g7–g6 3.g2–g3! h7–h6 4.f2–f3! and Black is in Zugzwang. **2.f2–f4** This leads to a kingside value of 0 which is good enough in this sum game. The move 2. g2–g3 was stronger and would have led to a kingside value of 1/4. **2. ... g7–g6 3.g2–g3 g6–g5 4.f4–f5** and Black resigned.

9. **Nehlert - Precour, Baden 1992**
In this game, the constellation of the two kings is different. If White moves his king, he loses the g4-pawn and the game. Black, however, can capture the white pawn on f6 and keeps the material balance. Thus, White must bring Black into Zugzwang on the queenside to avoid losing. As in position 7, the 3 against 3 structure on the queenside is a *hot* game. Its value is $G_{QS} = \{\{1/2 \mid \{0 \mid \{0, \{\text{on} \mid \downarrow\} \mid \text{off}\}\}\} \mid \{-1 \mid -2\}\}$ and its temperature is $t_{QS} = 7/8$. In the game, White correctly played **1.b3–b4! a6–a5** Another line is 1. ... b7–b6 2.c4–c5! b6×c5 3.b4×c5 a6–a5 4.c5–c6 and Black is blocked. **2.b4×a5! c7–c5 3.a5–a6! b7×a6 4.a4–a5 ♔g5×f6 5.♔f3×f4** and although White had the advantage, the game ended in a draw after a further 38 moves.

10. **Jacek - Miler, Czech Rep. 1996**
As the a-file and the structure with the kings both have value 0, this game is decided on the kingside. Its value of $G_{KS} = 2 \mid \{2, \{\text{on} \mid 0\} \mid \text{off}\}, \{\text{on} \mid 2, \{2 \mid \text{off}\} \parallel 0, \{\text{on} \mid 0\} \mid \text{off}\}, \{\{\text{on} \mid 0\}, \uparrow \mid -2\}, \{0 \parallel \{\text{on} \mid 0\}, \uparrow \mid -2\} \parallel 0, *$ is very complex. G_{KS} is a first-player win. White's only winning move is 1.g3–g4, and Black's only winning move is 1. ... h7–h5. In the game, Black to play found the correct moves and won. **1. ... h7–h5!** 1. ... h7–h6? 2.g3–g4! **2.a3–a4 a6–a5! 3.f4–f5 g6×f5! 4.h2–h3 f7–f6 5.h3–h4 f5–f4 6.g3×f4 f6–f5 7.♔d4–e5 ♔c6×c5** and White had to resign 12 moves later.

11. **Schüssler - King, Gausdal 1990**
Black has the advantage on both sides of the board. The queenside has

a value of $G_{QS} = \downarrow\downarrow *$. The kingside has the very complex value $G_{QS} = * \parallel \{0, \{\{on \parallel 0\}, \{on \parallel *, \{on \mid *\} \mid *, \{* \mid off\}\} \mid \{* \mid off\}, \{*, \{on \mid *\} \mid *, \{* \mid off\}\}\} \mid *, \{* \mid off\}\}, \{0, \{\{on \mid 0\}, \{on \parallel *, \{on \mid *\} \mid *, \{* \mid off\}\} \mid \{* \mid off\}, \{*, \{on \mid *\} \mid *, \{* \mid off\}\}\} \mid *, \{* \mid off\} \parallel 0\} \mid -2 \,|||\, 0$ which is less than zero, thus a black win. In the game, Black won after **1.f3–f4 h6–h5 2.g2–g3 f7–f6** Now, the kingside has a value of 0. **3.a3–a4 a7–a5! 4.g3–g4 h5–h4! 5.f4–f5 g6–g5!** and White resigned.

12. **Scheske - Kiefer, Bingen 1998**
This position is an "exception to the rule". At first sight, everything seems clear, the kings cannot move, thus the player who gets blocked on the kingside loses. The kingside structure has a value of $G_{KS} = 0$ which means Black to play is lost. Here, however, Black has one big advantage, his king is closer to the pawns than White's. In the game, Black manages to produce a position where White's otherwise winning passed pawn on the e-file is stopped by Black's king while his own passed pawn decides the game. **1.** ... **f7–f5! 2.f2–f3** The only move. Otherwise, either White gets blocked, or Black creates a winning passed pawn. For example 1. ... 2.e4×f5 g6×f5 3.h2–h3 e5–e4 4.h3–h4 h7–h5! and White is in Zugzwang. **2.** ... **f5–f4!** A locally bad move that allows White to create a passed pawn, but here it ensures Black's win. **3.g3×f4** After 3.g3–g4 g6–g5 White ends up being in Zugzwang again. **3.** ... **e5×f4 4.h2–h3 g6–g5 5.e4–e5 h7–h5** and White resigned. The black king easily stops the white e-pawn while Black will create a decisive passed pawn on the f-file.

Statistics

pos.	nodes	depth	pos.	nodes	depth
1	715	11	2	342	10
3	65	6	4	1,736	13
5	724	11	6	1,021	12
7	10,845	21	8	57,042	19
9	4,011	13	10	17,511	15
11	15,542	15	12	231,195	20

Figure 7.7: Statistics of the computations: number of nodes evaluated and maximum search depth per position.

The table shown in Figure 7.7 displays for each position the number of nodes evaluated and the maximum search depth required by the combinatorial game tree search. In order to improve the performance of the search, we use an evaluation function that detects global wins before a pawn actually gets promoted. Without these cutoffs the numbers of evaluated nodes would be considerably larger, and computing the values of the more complex positions would take several hours if not days.

Chapter 8

Zero-Sum Games without Zugzwang

In this chapter we discuss the application of combinatorial game theory to two-person zero-sum games with perfect information, a class of games analyzed in classical game theory. In Section 8.2 we show how zero-sum games are mapped to corresponding combinatorial games. In Section 8.3 we discuss the use of heuristic evaluation functions in local combinatorial search of zero-sum games. Section 8.4 demonstrates an application to the game Regio. With the help of a heuristic evaluation function we compute approximate game values of Regio positions (Sections 8.4.3 and 8.4.4) and perform a game playing experiment decomposition search against standard alpha-beta search (Section 8.4.5). In Section 8.5 we summarize results and conclusions.

8.1 Introduction

Two-player zero-sum games are a well known class of games analyzed in classical game theory [50]. The values of a game's terminal positions are determined by a payoff function that returns number values. Player *Max* (or *Left* to use the CGT naming conventions) is interested in the highest possible scores while player *Min* (*Right*) wants the opposite. A final score of 0 is considered a drawn game.

One of the most fundamental results in game theory, the *minimax theorem* states that any non-terminal position[1] in a two-player zero-sum game with perfect information has a determined value. If both players play optimally,

[1]Note that here the term *position* includes the information which player has the move. Later we shall talk about the minimax values of a certain position depending on which player moves first.

the final score will be this value, the *minimax value* of a position. *Minimax evaluation* is a well known technique used to compute minimax values of zero-sum games: it produces a game's tree up to its terminal positions and backs up their values in order to compute the value of the root.

Many popular two-player games are zero-sum games of perfect information, for example the whole class of board games that end in (win, loss, draw) results like *checkers* or *chess*.

8.2 Mapping Zero-Sum Games to Combinatorial Games

Zero-sum games and combinatorial games differ in the way they define termination. In zero-sum games a payoff function determines the values (or scores) of terminal positions while in combinatorial games the player unable to move loses. All the same, it is possible to map zero-sum games to combinatorial games and analyze them with the help of combinatorial game theory, thus taking advantage of divide and conquer.

8.2.1 The Mapping Algorithm

The mapping algorithm, as for example used in Berlekamp's analysis of the game *Blockbusting* [3], is straightforward: First, we produce all possible lines of play of the original zero-sum game and replace each terminal position with score x by the corresponding CGT number x (see Section 2.2.6). Then, we back up these leaf values to obtain the values of the interior nodes of the game tree and finally of the root. The *composing step* consists of collecting the sets of left (G^L) and right (G^R) options and constructing the value of the actual node as $G = \{G^L \mid G^R\}$. Figure 8.1 illustrates this mapping algorithm with a simple example:

Left options are represented by arrows to the left, right options by arrows to the right. We compute the values of non-terminal nodes by backing up the leaf values, $G^{L_1} = \{3 \mid 1\}$, $G^{R_1} = \{-2 \mid -4\}$, and finally $G = \{\{3 \mid 1\}, 3 \mid -3, \{-2 \mid -4\}\}$.

8.2.2 Zugzwang

In order to analyze a zero-sum game with the methods of combinatorial game theory, the game mapping must have the following two properties:

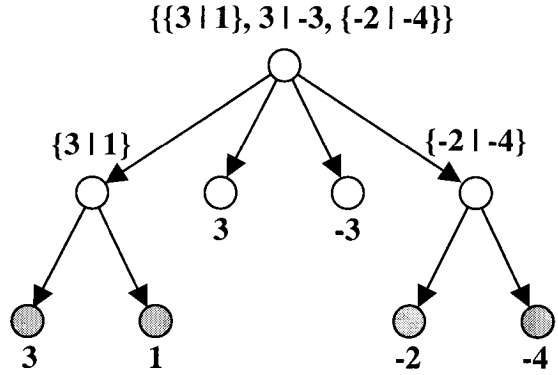


Figure 8.1: Mapping a zero-sum game to a corresponding combinatorial game.

1. *Leftscore* and *rightscore* of the combinatorial game are equal to the minimax values L_0 (player *Left* plays first) and R_0 (player *Right* plays first) of the original zero-sum game.

2. *Disjunctive sums* of zero-sum games can be translated subgame by subgame or as a whole, the result is the same.

As pointed out by Bewersdorff [7], these properties are guaranteed only if the original zero-sum game contains no positions of *mutual Zugzwang* (mZZ). In this case, the rule "never move in a number unless there is nothing else to do", Conway's *Number Avoidance Theorem* [11], makes sure the players never move in a terminal position of the original zero-sum game. When, in a sum of games, the only moves left are numbers, the free moves in the combinatorial sum reflect exactly the score of the original zero-sum game.

In positions of mutual Zugzwang, on the other hand, it is to both players' disadvantage to make a move. In combinatorial game theory, such positions are modeled by the game $0 = \{ \mid \}$ which offers no options to either player. Mapping mZZ positions to combinatorial games will not only result in the loss of their original minimax values (for instance the games $\{-1 \mid 1\}$ and $\{-3 \mid 3\}$ are both mapped to 0), but even worse, leads to completely wrong results when a player prefers to move in a number rather than in a Zugzwang position (see Figure 8.2). The mapping algorithm maps both G_1 and G_2 to 0, leading to a sum of $0 + 0 = 0$, a second player win. In the original zero-sum game, however, the first player wins by moving in G_1 forcing the second player to move in the even more disadvantageous game G_2. In the mapped combinatorial game this won't work: if the first player moves in G_1, she offers her opponent a free move in G_1 with the result that she has to play first in G_2 as well.

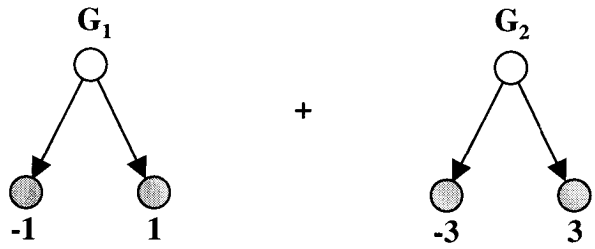


Figure 8.2: Mapping a sum of two mutual Zugzwang positions.

We conclude that we can map zero-sum games to combinatorial games provided they do not contain positions of mutual Zugzwang. A prominent example of a zero-sum game without Zugzwang is *Go* where the players have the right to pass if any move would worsen their situation.

8.2.3 Improved Mapping using the Reduced Canonical Form

When analyzing zero-sum games we are interested in positive and negative scores. In contrast to CGT, here a value of 0 stands for a drawn game, no matter who has made the last move. As a consequence we can significantly simplify the combinatorial games that result from the game mapping by omitting options that fight for the last move, but do not change the final score.

An *infinitesimal* or *small game* G (Conway [11]) is a game for which $-x < G < x$ holds for every possible positive number x. Leftscore and rightscore of infinitesimals are both 0, thus in terms of zero-sum games they are "draws". Provided there are no Zugzwang positions, infinitesimals are a neutral element in sums of zero-sum games: adding any infinitesimal to a sum will not change its final score.

Calistrate [10] defines the the *reduced canonical form* $\overline{G}$ of a game G as the *simplest* game infinitesimally close to G. In this context, "simplest" means the game with the smallest edge-set of the game tree. Algebraically, the reduced canonical forms build a subgroup *Rcf*, and the group of games is the sum $Rcf \oplus I$ where I is the subgroup of infinitesimals. In particular, G and $\overline{G}$ have the same leftscore and rightscore ($L_0(G) = L_0(\overline{G}); R_0(G) = R_0(\overline{G})$).

By using the reduced canonical form we can improve the mapping algorithm discussed in Section 8.2.1 in order to produce the simplest possible games that correspond to the original zero-sum games. When backing up the terminal values, instead of composing an interior node's value as

$G = \{G^L \mid G^R\}$ we compute the node's value as $G = \overline{\{G^L \mid G^R\}}$. This often leads to considerably simpler values for the local games which again result in more efficient local search and more efficient sum game play.

In the example of Figure 8.1 applying the reduced canonical form allows to delete *Left's* first move option leading to the simpler value $\overline{G} = \{3 \mid -3, \{-2 \mid -4\}\}$. Note that under conventional CGT rules, the games $G^{L_1} = \{3 \mid 1\}$ and $G^{L_2} = 3$ are not comparable: $G^{L_1} \parallel G^{L_2}$, and therefore the tree could not be pruned.

8.3 Heuristic Local Search

In order to compute the exact value of a combinatorial game, we must produce its entire game tree. However, the number of nodes in a game tree grows exponentially with its depth. Further, in the worst case, the combinatorial game value of the root requires an amount of memory exponential in the size of the game, as well. For this reason, only relatively small instances of combinatorial games admit exhaustive search, big instances often exceed the limits of running time and memory capacity.

8.3.1 Heuristic Evaluation Functions

A successful technique for computing approximate minimax values of zero-sum games that exceed the limits of exhaustive search is the use of *heuristic evaluation functions*. Instead of producing a game tree up to its terminal positions, we compute estimates of the values of non-terminal positions and back them up as if they were exact values. In zero-sum games it is often possible to find good heuristic evaluation functions that provide rather accurate scores. Combinatorial game values, on the other hand, are much harder to estimate as they are much more complex than "just" numbers. When mapping zero-sum games to combinatorial games, however, we can make use of the very same evaluation functions. If a zero-sum game admits a heuristic evaluation function that estimates the values of non-terminal positions, we can use it in order to map the original zero-sum game to a corresponding approximate combinatorial game.

8.3.2 An Iterative-Deepening Local Search Algorithm

The following algorithm *LocalHSearch* computes approximate combinatorial values of zero-sum games without Zugzwang:

```
function LocalHSearch(int depth, max_depth, gameNum): TGameValue;
/* computes an approximate combinatorial value of a local game */
begin
  if TerminalPosition() return Score();
  if depth = max_depth return Evaluate();
  G^L ← { }; G^R ← { };
  forall left moves m do /* recursively evaluate all left moves... */
    ExecMove(m);
    G^L ← G^L ∪ LocalHSearch(depth+1, max_depth, gameNum);
    UndoLastMove();
  endfor;
  forall right moves m do /* recursively evaluate all right moves... */
    ExecMove(m);
    G^R ← G^R ∪ LocalHSearch(depth+1, max_depth, gameNum);
    UndoLastMove();
  endfor;
  return overline{{G^L | G^R}}; /* reduced canonical form */
end LocalHSearch;
```

The recursion ends if either a terminal position or the specified maximum search depth is reached. The functions *Score* and *Evaluate* both return combinatorial numbers. After computing the values of all left (G^L) and right (G^R) options, we compute the actual game value as the *reduced canonical form* of the composed game $\{G^L \mid G^R\}$.

Ideally the heuristic local search algorithm is embedded in an *iterative deepening* loop:

```
for actual_depth := 1 to max_search_depth do
  forall local games G_i do
    V_i ← LocalHSearch(1, actual_depth, i);
  endfor;
endfor;
```

We start with a search depth of 1 which we increase until a specified depth (or time) limit is reached. The main advantage of the iterative deepening approach becomes obvious when we compute local values of several local games: At any time of the computation, the results computed at the previous depth level are available and allow a sum play algorithm to make a move decision.

Further when computing local game values in parallel, each local search process can iterate its search depth independently and thus automatically adapt to a local game's size.

8.4 The Game Regio

In this section, we analyze the game *Regio*, a zero-sum game without Zugzwang, with the help of combinatorial game theory. We use the mapping algorithm presented in Section 8.2 to transform Regio positions into corresponding combinatorial games. In Section 8.4.5 we discuss a game play experiment between a heuristic decomposition search algorithm and standard heuristic alpha-beta search.

8.4.1 Introduction and Rules

Regio (Müller [40]), a game of territorial control similar to the game *Snort* (*Winning Ways* [5]), models the initial phase of establishing local service for an enterprise. The immediate goal of Regio is to control as many local regions as possible.

The game is played by n players on an undirected graph $G = (V, E)$. Each player has his own color. At the start of the game, all nodes are uncolored. At his turn, a player selects an uncolored node and colors it and all its yet uncolored neighbors with his color. The game is finished when all the nodes are colored, and the player who has colored the most nodes wins.

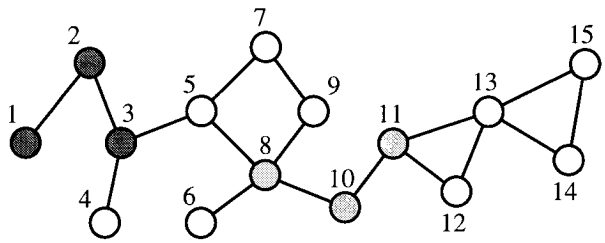


Figure 8.3: *Regio*, played on an undirected graph. After the moves to nodes 2 and 10 the game decomposes into four independent subgames.

During play, Regio decomposes into independent subgames whose sum determines the value of the game. (In the example of Figure 8.3, we already have four subgames after two moves.) Thus, in Regio, *divide and conquer* is a most logical and promising approach. In the following sections we analyze Regio in its two-player version.

8.4.2 General Properties of Regio

Proposition 8.1 *Regio is a zero sum game.*

This becomes obvious if we define the final score of a game G as the difference of the players' number of colored nodes: $S(G) = Lnodes(G) - Rnodes(G)$. The gain of player *Left* equals the loss of player *Right*. *Left* tries to maximize $S(G)$ while *Right* wants to minimize it. A final score of $S(G) = 0$ stands for a drawn game.

Proposition 8.2 *Regio is a symmetric game.*

In any position both players have exactly the same move options. If player *Left* to move can gain a score of x, then player *Right* can gain $(-x)$ if he was to move first. We define $L_0(G)$ and $R_0(G)$ as the resulting *minimax values* of a game G if *Left* respectively *Right* plays first. Proposition 8.2 then results in the equation:

$$L_0(G) = -R_0(G) \tag{8.1}$$

Proposition 8.3 *In Regio there are no Zugzwang positions.*

The proof for this proposition is furnished by the following simple strategy that guarantees the first player a score of at least zero in any Regio position:

- "Always play the move that colors the maximum number of nodes. If there is more than one such move, choose any of them."

On every turn, the second player can at most equal the first player's number of colored nodes. Therefore, the best score the second player can reach is zero. Expressed with the *minimax values* $L_0(G)$ and $R_0(G)$ we get:

$$L_0(G) \geq R_0(G) \tag{8.2}$$

We might conclude that Regio is a very simple game and the best move is always one of the biggest moves that color the maximum number of nodes. But this is wrong! Figure 8.4 shows a counter-example:

The biggest moves are the moves to nodes 1 and 3 that color four nodes each. A move to node 1 is answered with a move to node 3 (and vice versa). This leaves one extra node for the first player. The move to node 2 on the other hand only colors three nodes, but results in three extra nodes for the first player.

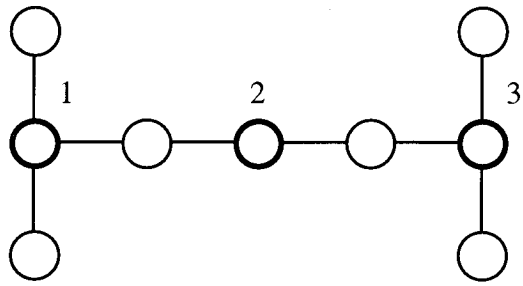


Figure 8.4: In this game, provided it is played by itself and not as a part of a sum, the best move (the move to node 2) is not the move that colors the most nodes.

8.4.3 Applying Combinatorial Game Theory to Regio

We map Regio positions to combinatorial games as described in Section 8.2. The properties of Regio positions discussed in Section 8.4.2 are reflected in their corresponding combinatorial games:

- All games $G \in \mathit{Regio}$ are their own *inverses*: $G + G = 0$, $G = (-G)$. This is easily proven with the help of the *Tweedledee-Tweedledum strategy* [5]: In a sum of two identical Regio games, the second player reaches a final score of zero by answering the first player's moves with analogous moves in the other game (see Figure 8.5). On the other hand, as shown in Section 8.4.2, the first player always has a strategy that guarantees him at least a value of zero (proposition 8.3). Thus, the value of the sum $G + G$ must be 0.

- $\forall G \in \mathit{Regio} : \mathit{leftscore}(G) = -\mathit{rightscore}(G) \geq 0$. This is the same result that we have already seen in Section 8.4.2, equation 8.1. As a consequence, *mean values* of Regio positions are always zero, and *thermographs* of Regio positions are symmetric with respect to the vertical zero-axis.

- *Incentives* are always equal for both players.

8.4.4 Heuristic Local Search in Regio

In Regio a good estimate for the value of a non-terminal position is the actual score, the number of nodes colored by *Left* minus the number of nodes colored by *Right* at this point in the game (see Figure 8.6). This score represents

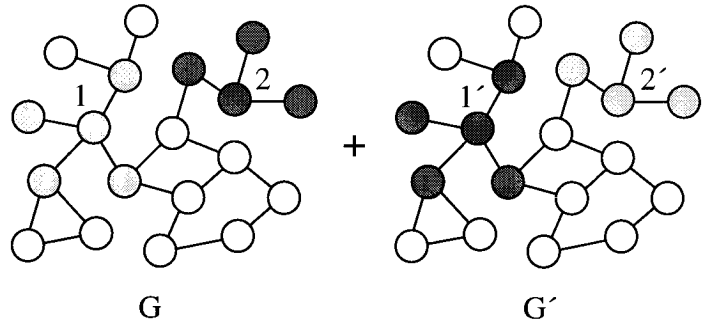


Figure 8.5: The Tweedldee-Tweedledum strategy: If the first player moves to node x in game G, the second player answers with the move to node x' in game G' and vice versa. The final score will be 0 no matter whatever the first player tries.

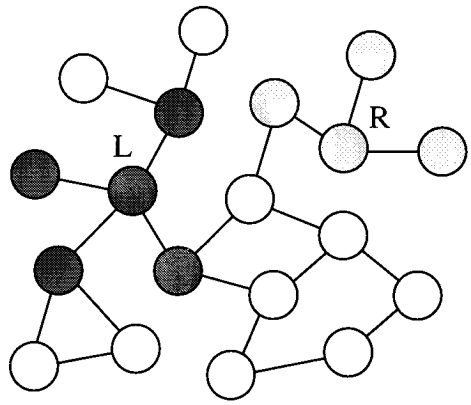


Figure 8.6: Heuristic evaluation in Regio: The actual score is $5 - 4 = 1$ in *Left's* favor. As the game consisting of the remaining uncolored nodes has a mean value of 0, the score is equal to the mean value of the position, thus a good estimate for the exact game value.

the *mean value* of the actual position as the mean value of the remaining uncolored nodes is zero.

A good heuristic evaluation function leads to more accurate game values, the deeper we search before applying it. This holds for heuristic local search as well as for minimax evaluation of zero-sum games.

The example shown in Figure 8.7) illustrates *heuristic local search* at increasing search depths:

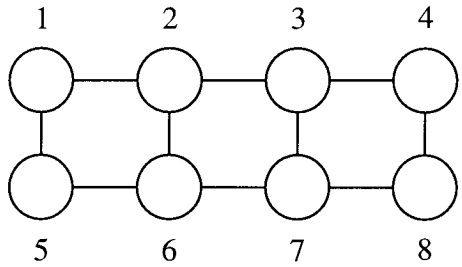


Figure 8.7: Regio: a 4 × 2 rectangle.

- **search depth 1**: Each player has the possibility to color at most four nodes with his first move which results in a heuristic game value of $G_{d1} = \{4 \mid -4\}$. Its leftscore, rightscore and temperature are $L_0(G_{d1}) = 4, R_0(G_{d1}) = -4, t_0(G_{d1}) = 4$.

- **search depth 2**: The second player is allowed to answer the first player's move. The computed values are $G_{d2} = 7 \mid 1 \parallel -1 \mid -7$. $L_0(G_{d2}) = 1, R_0(G_{d2}) = -1, t_0(G_{d2}) = 4$.

- **search depth 3**: At search depth 3 the computed values are $G_{d3} = 8 \mid 6 \parallel 2 \mid 0 \vert\vert\vert 0 \mid -2 \parallel -6 \mid -8$. $L_0(G_{d3}) = 2, R_0(G_{d3}) = -2, t_0(G_{d3}) = 4$.

- **search depth 4**: At search depth 4 all possible lines of play end in terminal positions. The exact value of the game, its temperature, left- and rightscore are the same as already computed at depth 3: $G = 8 \mid 6 \parallel 2 \mid 0 \vert\vert\vert 0 \mid -2 \parallel -6 \mid -8$. $L_0(G) = 2, R_0(G) = -2, t_0(G) = 4$. Note that the value of game G is equal to the sum $\{4 \mid -4\} + \{3 \mid -3\} + \{1 \mid -1\}$.

These game values are computed using the *reduced canonical form* as discussed in Section 8.2.3. The exact game value as computed by conventional rules of CGT is much more complex: $G = 6, \{8 \mid 6\}, \{8, \{8 \mid 6\} \mid 2, \{4 \mid 2\}\} \mid$

$2, \{2 \mid 0\}, \{6, \{6 \mid 4\} \mid 0, \{2 \mid 0\}\} \parallel -2, \{0 \mid -2\}, \{0, \{0 \mid -2\} \mid -6, \{-4 \mid -6\}\} \mid -6, \{-6 \mid -8\}, \{-2, \{-2 \mid -4\} \mid -8, \{-6 \mid -8\}\}$.

A game's leftscore L_0, rightscore R_0 and temperature t_0 are illustrated by its *thermograph*. The thermograph [11] displays leftscore and rightscore of a game depending on the amount t by which it is *cooled*. At $t = t_0$, the game's *temperature*, left and rightscore become identical, the *mean value* of the game. Figure 8.8 shows the thermographs of G computed at increasing search depths from 1 to 4. Step by step they approximate the thermograph of the exact game value.

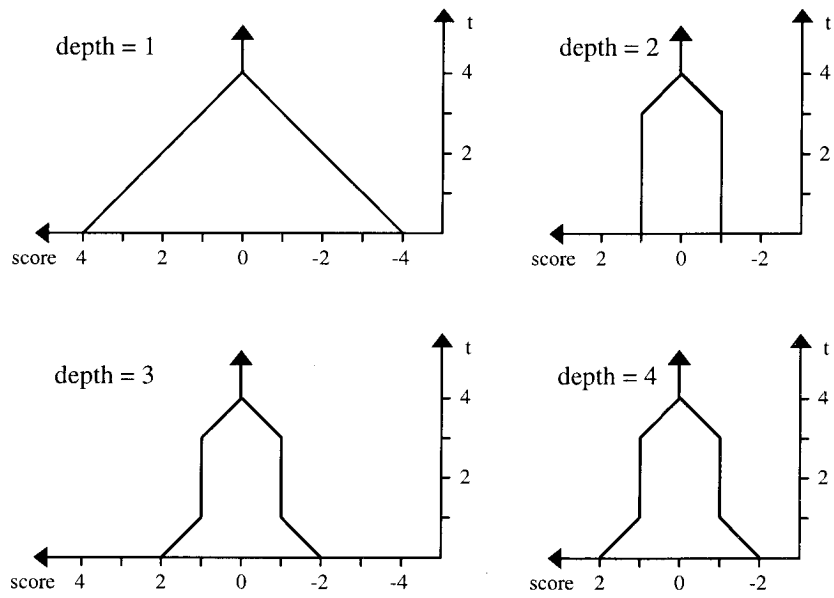


Figure 8.8: The thermograph of the Regio position shown in Figure 8.7 at search depths ranging from 1 to 4.

8.4.5 A Game Play Experiment

In the following game play experiment, we let *greedy hotstrat*, a simple decomposition search algorithm play against standard alpha-beta search. Both algorithms use the heuristic evaluation function discussed in the previous Section, use hash tables of equal size, and are allowed to search up to the same maximum depth of 4. (The depth limit for full-width alpha- beta search is low in games with up to 200 possible moves at each turn.)

The Algorithms

Greedy hotstrat is a very simple heuristic divide and conquer algorithm for playing Regio based on *decomposition search* (see section 2.3 for a general description of decomposition search):

- *Local Search*: We use the heuristic evaluation function discussed in the previous section in combination with *iterative deepening* depth-first search. The search starts with a maximum depth of 1 which is increased until the specified depth limit is reached.

- *Local Game Selection and Move Decision*: If after removing dominated options and moves with dominated incentives more than one candidate move remains, we must make a move decision based on local information. A simple strategy for choosing a local game to play in is *hotstrat* which selects the game with the highest temperature. If there is more than one non-dominated move in this game (or these games), we greedily choose the move that colors the most nodes. If we still have a choice of more than one move, we select the move that leaves a game whose temperature is closest to the maximum temperature of all other local games.

The *alpha-beta* algorithm uses the same game specific operations as *greedy hotstrat*. It does not use heuristics to restrict the number of move options. However, iterative deepening in combination with a hash table yields a very good move ordering, as the best move at depth $d-1$ is most likely also one of the best moves at depth d.

Results

The experiment consists of 10 different starting positions played on a 15×15 grid. The grid is randomly divided into a number of independent local games. (Figure 8.9 shows the first starting position, the others are described in appendix C.) From each position, the algorithms play both sides, first and second player.

The results of the experiment are compiled in Figure 8.10. For every position, the table lists its number of independent local games and the score obtained as first player and the number of nodes searched by either algorithm. Although not statistically conclusive, this experiment indicates that the divide and conquer approach for *heuristic game tree search* is promising: the simple greedy hotstrat algorithm does not score worse than alpha-beta and searches a much smaller number of nodes in the average. Small local

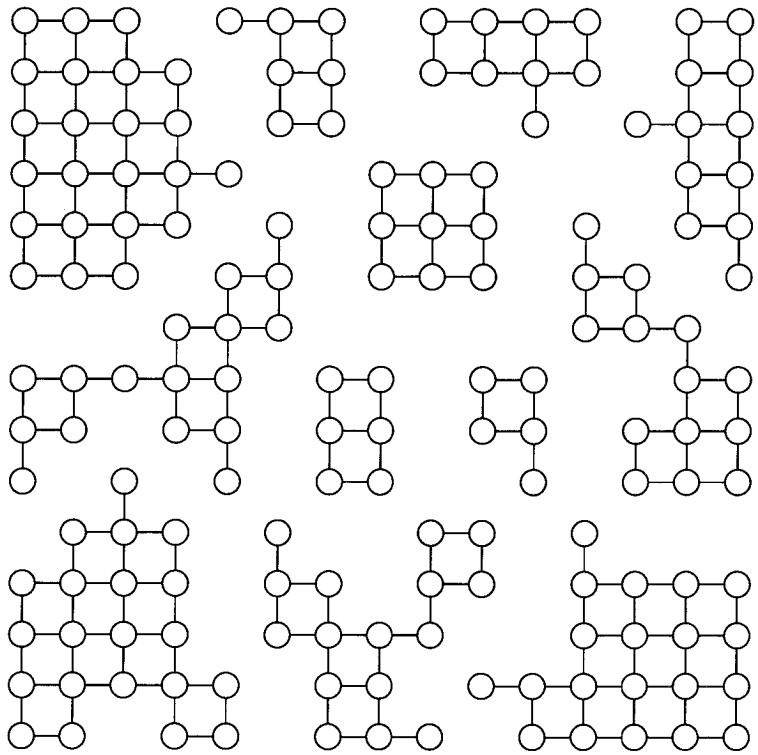


Figure 8.9: Position 1 of the game play experiment: a sum of 12 local games.

games favor the divide and conquer algorithm, as they offer the players less move options. When the local games grow bigger, on the other hand, alpha-beta performs comparably better. The relatively low number of nodes in alpha-beta search (considering the many move options of the players in the sum game) is due to the many cutoffs it can make in Regio. Local combinatorial search, on the other hand, does not make cutoffs but searches the whole local tree up to the specified maximum depth.

8.5 Summary and Conclusions

Two-person zero-sum games are an important class of games analyzed in classical game theory. In these games, terminal positions are assigned a score that determines the outcome of the game. Player *Left* (Max) wants to maximize, player *Right* (Min) wants to minimize this score. A final score of 0 stands for a drawn game. The *minimax theorem* states that every non-terminal position in a zero-sum game has a determined value which results from optimal play by both players.

		greedy hotstrat		*alpha-beta*	
pos.	games	score	nodes	score	nodes
1	12	1	414,914	1	1,116,646
2	20	4	65,714	4	644,677
3	22	5	31,300	3	785,328
4	12	0	5,532	0	3,469,245
5	12	4	197,450	4	1,335,039
6	9	2	839,328	2	1,975,996
7	16	0	51,360	0	1,315,216
8	11	3	413,338	3	1,319,922
9	24	2	34,082	2	628,525
10	11	3	619,136	3	1,876,232

Figure 8.10: Statistics of the game play experiment *greedy hotstrat* vs. *alpha-beta* search.

8.5.1 Mapping Zero-Sum Games to Equivalent Combinatorial Games

Zero-sum games without Zugzwang positions can be mapped to equivalent combinatorial games in such a way that the minimax values of the original zero-sum game correspond to leftscore and rightscore of the combinatorial game. The straightforward mapping algorithm translates the scores of terminal positions of the zero-sum game into combinatorial numbers which are backed up using the operation of game composition. This algorithm can be improved in terms of efficiency with the help of the *reduced canonical form* [10] which yields the *simplest* combinatorial game (the game with the smallest game tree) infinitesimally close to the game computed by the original algorithm.

8.5.2 Heuristic Local Search

The use of heuristic evaluation is essential in game tree search as it allows to compute approximate values of games too big to be searched entirely. In general, it is difficult to find good estimates for combinatorial game values as they can have a very complex structure. In the class of zero-sum games without Zugzwang, however, the same heuristic evaluation functions that are successfully used in minimax evaluation yield good approximate game values in local combinatorial search. Decomposition search based on heuristic local game evaluation is a promising game playing algorithm and performs well

against standard alpha-beta search as the application to Regio demonstrates.

Chapter 9

Conclusion

In this chapter, we summarize the contents and the main contributions of this thesis and give directions for future research.

9.1 Summary and Contributions

Combinatorial game theory is a powerful instrument for the analysis and solution of games. It applies the divide and conquer approach, a fundamental paradigm of computer science, to game tree search. This thesis consists two parts: the two case studies on king and pawn endgames in *chess* and on *zero-sum games* that apply the methods of combinatorial game theory in the field of algorithmic, computational game theory and the *Game Bench* application framework.

- We present a general computation model for local games that contain *global threats*. In sums of such games, a local move can have a globally decisive effect. The computation model handles *entailing moves* [5] [13] that force an immediate reaction in the local game where they are played. In an application to king and pawn endgames in chess we compute game values of Zugzwang positions and apply divide and conquer to game positions that are too complex to be searched exhaustively with standard full-width search as used by conventional chess programs. In these positions, combinatorial game theory provides clear mathematical solutions to a class of endgames that chess literature fails to describe accurately.

- In the second case study we analyze *zero-sum games without Zugzwang*. Using the *reduced canonical form* [10] we improve the standard algorithm that maps such games to combinatorial games resulting in more

efficient local search and sum game play. Zero-sum games often admit heuristic evaluation functions that estimate values of non-terminal positions. In analogy to minimax search, we use such evaluation functions to create an artificial search horizon and compute approximate combinatorial values of local games. The combination of *heuristic* local search and sum play algorithms is a promising application of decomposition search to heuristic game playing.

- The *Game Bench* is an application framework for combinatorial game programs. It combines the basic CGT calculus implemented in Wolfe's *Gamesman's Toolkit* [52] with game independent algorithms for combinatorial game tree search and sum game play. Further, it provides algorithms and data structures that support the implementation of games played on rectangular grid-boards and graphs and their user interfaces. The Game Bench is successfully used as an educational program for game tree search on the *EducETH* [12] web server and has served as a program basis in several game programming projects at ETH Zürich.

9.2 Future Research

Further research is possible in many areas addressed by this thesis. While combinatorial game theory has been subject to mathematical research for many years, there seem to be few applications of the theory in computer science. Directions for further research in *computational game theory* include:

- Computing exact values of combinatorial games involves searching their entire game tree which takes time and space exponential in the depth of this tree. In zero-sum games, heuristics often allow good approximations of game values. Are there other classes of combinatorial games that admit heuristic evaluation functions as well? We are interested in guessing the value or at least the *outcome class* (> 0, < 0, $= 0$, $\| 0$) of a game.

- In Section 8.3.1, in analogy to minimax evaluation, we use a heuristic evaluation function to estimate game values of non-terminal positions of zero-sum games. Are there classes of combinatorial games that admit cutoffs in their game trees similar to alpha-beta cutoffs in minimax evaluation? Such cutoffs of branches in the game tree that lead to dominated options would of course significantly improve the efficiency of local combinatorial game tree search.

- A promising test-field for the application of divide and conquer in computer game playing is the game *Amazons* [53]. In Amazons endgames the overall result is determined by the players' free moves in different local games, but also in certain middlegame positions, a heuristic decomposition seems possible.

The following two issues belong to the field of mathematical and computational chess.

- In Chapter 7, we compute finite combinatorial values of pawn structures in king and pawn endgames. Elkies [13] shows that loopy game values occur in more complex chess positions which include other pieces as well. Find algorithms to identify such positions and compute their values.

- An interesting application of another mathematical approach to chess endgames is the *theory of corresponding squares* as described by Awerbach [1].

Finally, there are all kinds of possible extensions to the Game Bench framework, for example:

- Run and test decomposition search on a parallel computer. The Game Bench's search engine supports running several local searches in parallel, thus no major changes in the software are required.

- Extend the library of game independent algorithms for game tree search and sum game play.

Bibliography

[1] J. Awerbach. *Bauernendspiele.* Sportverlag, Berlin, 1983.

[2] E. Berlekamp. Four Games for Gardner. Presented at the Gathering for Gardner IV, Atlanta, GA, Feb. 2000.

[3] E. Berlekamp. Blockbusting and Domineering. *Journal of Combinatorial Theory*, A 49:67–116, 1988.

[4] E. Berlekamp. The Economist's View of Combinatorial Games. In Nowakowski [44], pages 365–408.

[5] E. Berlekamp, J.H. Conway, and R. Guy. *Winning Ways for Your Mathematical Plays.* Academic Press, New York, NY, USA, 1982.

[6] E. Berlekamp and D. Wolfe. *Mathematical Go: Chilling Gets the Last Point.* A K Peters Ltd, 1994.

[7] J. Bewersdorff. *Glück, Logik und Bluff.* F. Vieweg und Sohn, Braunschweig/Wiesbaden, 1998.

[8] D. Biella. GAG - eine Mehrpersonen-Spielumgebung. Diploma thesis, ETH Zürich, 1999.

[9] C. Bouton. Nim, a Game with Complete Mathematical Theory. *Annals of Mathematics*, pages 35–39, 1901/02.

[10] D. Calistrate. The Reduced Canonical Form of a Game. In Nowakowski [44], pages 409–416.

[11] J.H. Conway. *On Numbers and Games.* Academic Press, London, 1976.

[12] EducETH. *http://educeth.ethz.ch*
a platform for the exchange of teaching materials on the internet.

[13] N.D. Elkies. On Numbers and Endgames: Combinatorial Game Theory in Chess Endgames. In Nowakowski [44], pages 135–150.

[14] Arpad Elo. *The Rating of Chessplayers: Past and Present.* Arco, 1986.

[15] W. Fierz. Go Endgames. Semesterarbeit, ETH Zürich, 1992.

[16] A. Fraenkel. Combinatorial Games: Selected Bibliography with a Succinct Gourmet Introduction. In Nowakowski [44], pages 493–537.

[17] A. Fraenkel and D. Lichtenstein. Computing a Perfect Strategy for n x n Chess Requires Time Exponential in n. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 1981.

[18] D. Garcia. Gamesman: A finite, two-person, perfect-information game generator. Master of Science thesis, UC Berkeley, 1995.

[19] D. Garcia. Xdom: A Graphical, X-Based Front-End for Domineering. In Nowakowski [44], pages 311–314.

[20] R. Gasser. Heuristic Search and Retrograde Analysis: their application to Nine Men's Morris. Diploma thesis, ETH Zürich, 1990.

[21] R. Gasser. *Harnessing Computational Resources for Efficient Exhaustive Search.* PhD thesis, ETH Zürich, 1995.

[22] P.M. Grundy. Mathematics and Games. *Eureka*, 2:6–8, 1939.

[23] R. Guy. What Is a Game? In Nowakowski [44], pages 43–60.

[24] R.K. Guy, editor. *Combinatorial Games.* American Mathematical Society, Rhode Island, 1990.

[25] O. Hanner. Mean play of sums of positional games. *Pacific Journal of Mathematics*, 9:81–89, 1959.

[26] F.-H. Hsu, M.S. Campbell, and A.J. Hoane. Deep Blue system overview. In ACM, editor, *Conference proceedings of the 1995 International Conference on Supercomputing, Barcelona, Spain, July 3–7, 1995*, pages 240–244, New York, NY 10036, USA, 1995. ACM Press.

[27] R. Hyatt. *Crafty v17.9*, a very strong freeware chess program, rated 2499 on the Elo [14] scale by the Swedish Chess Computer Association in August 2000.
http://www.cis.uab.edu/info/faculty/hyatt/hyatt.html.

[28] A. Kierulf. A simple game-playing program. ETH Zürich, 1989.

[29] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* PhD thesis, ETH Zürich, 1990.

[30] A. Kierulf, K. Chen, and J. Nievergelt. Smart Game Board and Go Explorer: A Case Study in Software and Knowledge Engineering. *Communications of the ACM*, February 1990.

[31] G. Krasner and S. Pope. A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[32] S. Liang. *The Java Native Interface.* Addison Wesley, 1999.

[33] A. Marzetta. *ZRAM: A Library of Parallel Search Algorithms and Its Use in Enumeration and Combinatorial Optimization.* PhD thesis, ETH Zürich, 1998.

[34] F. Mäser. The Game Bench Website: *http://www.inf.ethz.ch/~maeser/game_bench.*

[35] F. Mäser. Global Threats in Combinatorial Games: a Computation Model with Applications to Chess Endgames, 2000. to appear in the proceedings of the Combinatorial Games Workshop, MSRI, UC Berkeley, 2000.

[36] J. Milnor. Sums of positional games. *Annals of Mathematics Studies*, 28:291–301, 1953.

[37] F.L. Morris. Playing Disjunctive Sums is Polynomial Space Complete. *Int. Journal of Game Theory*, 10(3/4):195–205, 1981.

[38] K. Müller. *Secrets of Pawn Endings.* Everyman Chess, 2000.

[39] M. Müller. *Computer Go as a Sum of Local Games: An Application of Combinatorial Game Theory.* PhD thesis, ETH Zürich, 1995.

[40] M. Müller. SimCargo: A Simple Model of Air Cargo. SPP-ICS project presentation, 1997.

[41] M. Müller. Decomposition Search: A Combinatorial Games Approach to Game Tree Search, with Applications to Solving Go Endgames. *IJCAI*, 1:578–583, 1999.

[42] J. Nielsen. The Usability Engineering Life Cycle. *Computer*, pages 12–22, March 1992.

[43] J. Nievergelt, R. Gasser, F. Mäser, and C. Wirth. All the Needles in a Haystack: Can Exhaustive Search Overcome Combinatorial Chaos? *Lecture Notes in Computer Science*, 1000:254–265, 1995.

[44] R.J. Nowakowski, editor. *Games of No Chance.* Cambridge University Press, New York, 1996.

[45] J. Nunn. *Secrets of Rook Endings.* B.T. Batsford Ltd., 1992.

[46] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers.* Springer-Verlag New York, Inc., New York, 1997.

[47] M. Schneider. Games as Simple Models for Traffic Problems. Diploma thesis, ETH Zürich, 1999.

[48] C.E. Shannon. Programming a Computer for Playing Chess. *Philosophical Magazine*, 41(4):256–275, 1950.

[49] J. Speelman. *Endgame Preparation.* B.T. Badsford Ltd., London, 1981.

[50] J. von Neumann and O. Morgenstern. *Theory of Games and Economic Behaviour.* Princeton University Press, Princeton, 1944.

[51] C. Wirth. The Smart Chess Board. ETH Zürich, 1990.

[52] D. Wolfe. The Gamesman's Toolkit. In Nowakowski [44], pages 93–98.

[53] W. Zamkauskas. The Game *Amazons*, first published in El Acertijo, issue 4, Dec. 1992.

[54] E. Zermelo. Über eine Anwendung der Mengenlehre auf die Theorie des Schachspiels. *Proceedings of the Fifth Congress of Mathematics, Cambridge*, 2:501–504, 1913.

[55] A.L. Zobrist. A New Hashing Method with Application for Game Playing. Technical Report 88, Univ. of Wisconsin, April 1970.

Appendix A

Glossary

Many terms in the glossary are assigned to one or more of the following categories:

- *CGT*: Combinatorial game theory.
- *GT*: Classical game theory.
- *Java*: Java, object-oriented programming.

Abstract Class (Java) A class that contains *abstract methods.*

Abstract Method (Java) A prototype of a method, declared with arguments and a return type. The implementation of an abstract method is provided by a subclass of the *abstract class* or *interface* in which it is specified.

Alpha-Beta Search (GT) An (exact) algorithm for *minimax evaluation* that cuts off certain branches of the game tree that have no influence on the game value of the root.

Cooling (CGT) A game G is cooled by adding a tax t on every move. $G_t = \{{G^L}_t - t \mid {G^R}_t + t\}$.

Decomposition Search A CGT-based divide and conquer algorithm that computes minimax values of games that can be partitioned into independent components.

Final Class (Java) A class that may not be subclassed.

Global Threat The threat to execute a move in a *local game* G that wins the whole sum of which G is a component.

Hot Game (CGT) A game with a high *temperature.*

Hotstrat A simple strategy for playing a sum of combinatorial games that always moves in the local game with the highest *temperature.*

Incentive (CGT) The gain made by moving in a certain game. Left incentive: $G^L - G$; right incentive: $G - G^R$.

Interface (Java) A set of *abstract methods* that must be implemented by a class in order to implement the interface as a whole.

Leftscore (CGT) The minimax value of a game with *Left* to play first.

Local Game An independent component of a sum-game.

Local Search An algorithm that produces the tree of a (local) combinatorial game and computes its value. In contrast to minimax evaluation both players' options are considered in every position.

Loopy Game (CGT) A game that does not meet the *ending condition* which states that play will always come to an end because some player is unable to move.

Mean (CGT) The number of points a game is worth on average.

Minimax Evaluation (GT) A technique used for computing the minimax value of a two-person zero-sum game with perfect information.

Minimax Theorem (GT) A fundamental theorem of classical game theory which states that any non-terminal position of a two-person zero-sum game with perfect information has a determined value, the *minimax value.*

Normal Termination Rule (CGT) The rule which states that a player unable to move loses.

Rightscore (CGT) The minimax value of a game with *Right* to play first.

Temperature (CGT) A measure how urgent it is to move in a game.

Terminal Position (GT, CGT) A game-position that can be evaluated statically.

Thermograph (CGT) A graphical representation of *leftscore* and *rightscore* of a *cooled* game.

Thread (Java) A single independent stream of execution within a program. The Java runtime environment enables parallel execution of threads.

Zero-Sum Game (GT) A two-player game whose terminal values are determined by a payoff-function. The gain of player *Max* equals the loss of player *Min*.

Zugzwang A situation in which all possible moves worsen a player's position. (German, meaning "being forced to move".)

Appendix B

Chess Notation

We use standard *long algebraic notation* to display chess moves. The board's columns (*files* in chess-jargon) are labeled a to h, the rows (*ranks*) 1 to 8. For instance, in the diagram of Figure B.1 ♔f3–f2 stands for a white king move from square f3 to square f2.

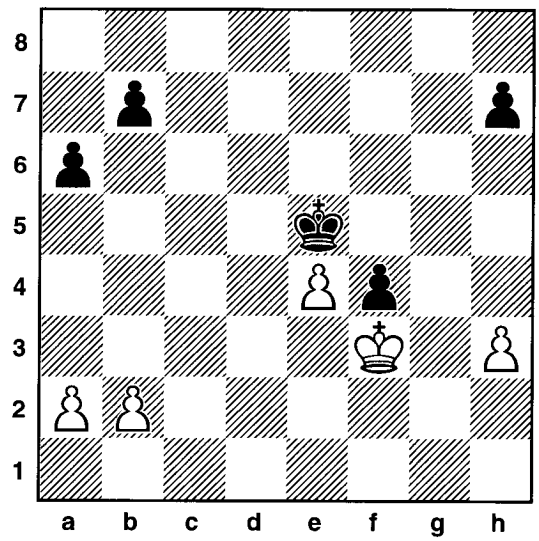


Figure B.1: A position from the game Sveda vs. Sika, Brno 1929. (see also chapter 7)

In order to annotate single moves we follow the conventions introduced by Nunn [45]:

- An exclamation mark ! stands for the only move that does not change the game's result (win, loss, draw). For example in our diagram (Figure

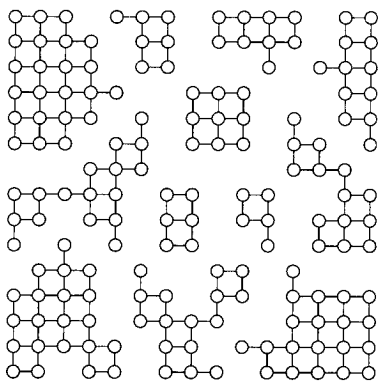
B.1) 1.h3–h4! is White's only winning move. In fact, all other moves lose.

- A question mark ? stands for a move that does change the game's result. If Black was to play first, the move 1. ... h7–h5? would be a losing move in a won position whereas 1. ... a6–a5! wins.

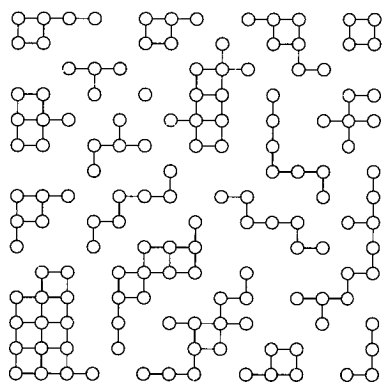
Appendix C

Regio Game Play Experiment

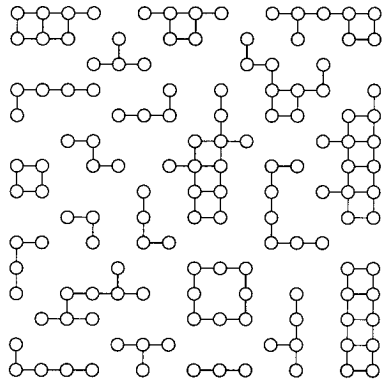
This game play experiment (see also Section 8.4.5) consists of 10 different starting positions played on a 15×15 grid which is divided into a number of independent local games. From each position, *greedy hotstrat* and *alpha-beta* play both sides, first and second player. Both algorithms use the same heuristic evaluation function (see Section 8.4.4), use hash-tables of equal size, and are allowed to search up to the same maximum depth of 4. The results of the experiment are compiled in Figure C.1.



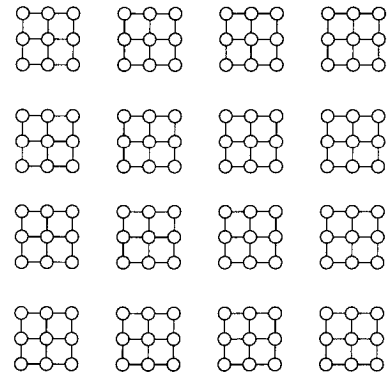
Position 1
12 local games.



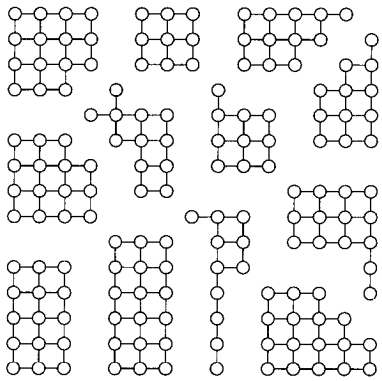
Position 2
20 local games.



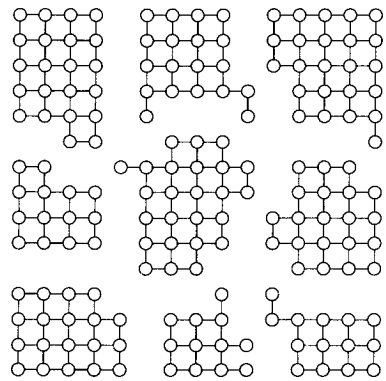
Position 3
22 local games.



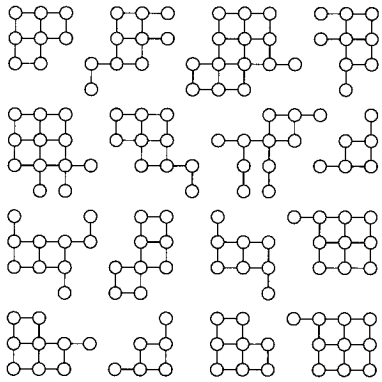
Position 4
12 local games.



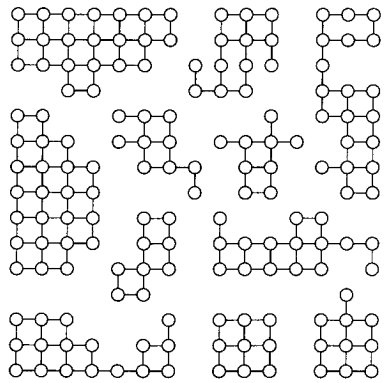
Position 5
12 local games.



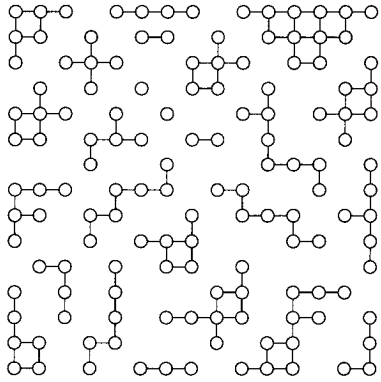
Position 6
9 local games.



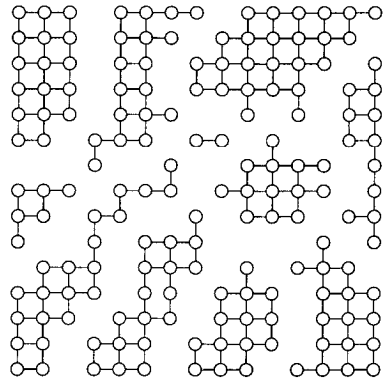
Position 7
16 local games.



Position 8
11 local games.



Position 9
24 local games.



Position 10
11 local games.

		greedy hotstrat		*alpha-beta*	
pos.	games	score	nodes	score	nodes
1	12	1	414,914	1	1,116,646
2	20	4	65,714	4	644,677
3	22	5	31,300	3	785,328
4	12	0	5,532	0	3,469,245
5	12	4	197,450	4	1,335,039
6	9	2	839,328	2	1,975,996
7	16	0	51,360	0	1,315,216
8	11	3	413,338	3	1,319,922
9	24	2	34,082	2	628,525
10	11	3	619,136	3	1,876,232

Figure C.1: Statistics of the game play experiment *greedy hotstrat* vs. *alpha-beta* search.


Seite Leer /
Blank leaf

Appendix D

Curriculum Vitae

1987	Matura Typ B, Gymnasium Oberwil BL.
1987–1993	Major in computer science and minor in photography at ETH Zürich, resulting in the degree of Dipl. Informatik-Ing. ETH.
1989	Software development at ABB research lab. Baden-Dättwil.
1989–1991	Software development at Ciba-Geigy Basel.
1993–2001	Assistant and Ph.D. student (1998) in the research group of Prof. Nievergelt, Institute of Theoretical Computer Science, ETH Zürich.