Programming an open-source Pentominoes player using brute-force algorithms.

Sebastien Vasey: `sebastien.vasey@gmail.com`
Yann Schoenenberger: `yann.sch@gmail.com`

January 3, 2008

Abstract

Is it possible to use brute-force algorithms efficiently to solve the game “Pentominoes” ?

This document presents our work on programming software able to play the game “Pentominoes” (and some variations of it). Pentominoes is a two-player game where the goal is to be the last one to put a pentomino (piece composed of five congruent squares) on an 8×8 board.

The goal was to write a program able to calculate in advance what would happen in any situation. The software would be able to play against a human and play perfectly if given enough time. Furthermore, the project is open-source: the source code is publicly available on the web and anyone interested can participate and share code.

We were able to write (in C++) a fully-featured program, able to play against a human player. However, the program takes a long time to finish if the user really wants it to play perfectly. We thus concluded that brute-force alone was not sufficient to play against a human player and that alternative strategies had to be considered.

Preamble

Conventions

In this report, the terms “bruteforce”, “brute force” and “brute-force” are considered to be equivalent as well as “opensource”, “open source” and “open-source”.

The “Travail de Maturité”

This report as well as Kinonk, the Software it is about are our “Travail de Maturité”. This a work every student in Geneva (Switzerland) has to do before finishing high school. The subject is freely chosen but it has then to be approved by the school.

Programming is not taught at school. We have it as a hobby, therefore Kinonk’s code might be easily improved by someone who has studied computer science. This would mainly be because of a lack of knowledge rather than bad coding habits, time constraints or any other reason.

English is not our first language. Therefore there might be some English mistakes as well.

The choice of the subject

As we, the authors, are interested in programming (and computer science in general) wanted to do something related to that field. We’ve ended up with the idea of making our computer play Katamino. This report is also an opportunity to make some people discover this very unknown game.

Prerequisites to understand this report

In order to fully understand this report, there are a few things one has to know.

- Basic knowledge in the field of computer science: you should know what an operating system, a programming language or software is.
- Good understanding of the C++ programming language and its features if you want to understand the code.

Copyright

Contents

Part I

Introduction

Chapter 1

Goals

This document is separated into two parts, each dealing with one of the two different goals set up before beginning the project. They are described below.

1.1 Using brute-force to beat a human mind

The original question this work had to answer was “What are the different brute-force algorithms that can be used to compete with the creativity of a human mind in the game of Katamino ?”.

This was to be answered by implementing some algorithm aimed to beat human players. Apart from that, no clear goal was formally defined for Kinonk. (the software’s name.)

1.2 Open-source project

The second goal of this project is to explain what is involved in the development of a relatively “big” project over a relatively“long” period of time. As this project is the biggest one undertaken by both authors yet (October 2007), it is interesting to see what is really needed to succeed in this kind of project.

Chapter 2

Defining our work

2.1 The Game

Kinonk, the program this report is about, is able to play the game of Pentominoes and some of its variations including Katamino. After reading this section, one will be able to understand and play both Pentominoes and Katamino.

2.1.1 Polyominoes

When playing Pentominoes and Katamino, one uses pieces called "pentominoes". In order to know what a pentomino is, one has to understand what a polyomino is.

A polyomino is a polyform with the square as its base form. It is constructed by placing several identical squares in the plane in such a way that at least one edge of each square coincides with an edge of one of the other squares.

Polyominoes are usually sorted according to the number of squares constructing them. The sets of different polyominoes are given names such as "domino" for all polyominoes made with two squares, "heptomino" when seven squares are needed etc.

For example, polyominoes are used in the well-known games of Tetris and Dominoes.

In the game of Pentominoes and Katamino, the players use polyominoes made of five squares: "pentominoes". Figure 2.1 is a picture of all existing pentominoes (without the symmetries or the rotations).

2.1.2 The Rules

Pentominoes and Katamino have very similar rules. Katamino is a variation of Pentomino, it includes all the rules of Pentominoes plus a few more subtleties.

The game of Pentominoes has no formal rules. It is more of a concept involving polyominoes: You try to make pentominoes fit in a grid and if you can't, then you lose. This implies that there is a lot of freedom concerning the interpretation of the rules.

On the other hand, Katamino is a commercial game.There is a set of precise rules which can not be modified, otherwise it is not Katamino anymore.

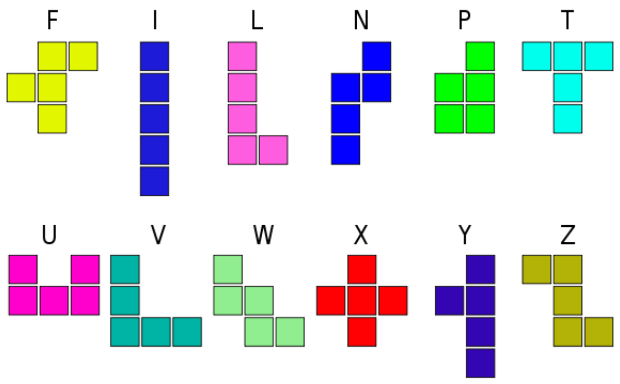

Figure 2.1: All twelve unique pentominoes (Picture from Wikipedia [31])

Pentomino

Pentomino is played on a finite rectangular grid. The players play one after the other in a predetermined order. Each player has to put one pentomino on the grid at a time. The goal is to be the last player to put a pentomino on the board.

Pentominoes is a multi player game. The number of players is only limited by the number of pentominoes in use. Usually, all pentominoes are available, but it is possible to decide not to use some of them.

For a move to be legal, it has to satisfy all of the following conditions.

- The played pentomino must not be already used. All unique pentominoes can be used only once.
- The pentomino must fit totally in the board.
- The pentomino can not overlap any of the other pentominoes already on the board.

Note that the size of the board is not limited, but still, in practical cases it is more interesting to use a board in which it is very unlikely to put all the pentominoes and then having a winner just because there are no pieces left. The only restriction regarding the board's size is that at least one of the twelve unique pentominoes fits so that there is a winner at the end.

It is also worth pointing out that the number of payers is, in theory, only limited by the number of pentominoes in use, but again in practical cases, it is better to make sure that all players are likely to play at least once.

Katamino

Katamino adds a few rules to Pentominoes.

- The size of the board is 8×8 squares.
- The number umber of players is limited to 4.
- For the first move to be valid, it has to overlap at least one of the four central squares.
- For all moves (except the first) to be valid, the played pentomino has to touch at least one other pentomino. Note that it is not compulsory to touch an edge of an other pentomino,touching just one corner is all right

Additional information about Katamino can be found on the official website:
`http://www.katamino.co.uk`

2.2 Algorithms

Wikipedia gives the following (informal) definition of algorithms [19]:

> "In mathematics, computing, linguistics, and related disciplines, an algorithm is a procedure (a finite set of well-defined instructions) for accomplishing some task which, given an initial state, will terminate in a defined end-state."

An algorithm is simply a list of steps (like a recipe). The algorithm takes some input, follows the steps according to the list and then outputs the result.

An example would make things clearer. Imagine you have a set of diamonds. For some reason, you're allowed to take one for free. You should obviously take the most expensive... To choose the diamond you want to take, you will use an algorithm:

Algorithm 1 Find the most expensive element in a set of diamonds

```
1: Take a list of all the available diamonds.
2: Take the next diamond in the list (the first one if you just began).
3: Look at the price of the diamond (If this is the first diamond in the list, then go back to step 2).
4: Compare the price of the diamond you have just picked up with the price of diamond you were already
   holding.
5: Drop the least expensive diamond.
6: if The end of the list has not been reached. then
7:     Go back to step 2.
8: else
9:     The diamond you are still holding is the most expensive. This is the one you are going to keep !
10: end if
```

In this case, the input is the list of diamonds and their price (step 1), the output is the diamond you should keep (step 9).

Algorithms have three important characteristics, among others, that will prove useful when designing Kinonk, the Katamino playing software.

- An algorithm relies on it's logical structure.
  Neither the implementation, nor the speed of the executor (computer,human...) has any importance, since the algorithm will lead to the same result in all cases.
  A consequence of this is that the speed of an algorithm is not measured using the time it takes to execute, but proportionally to the size of its input (constant, linear, logarithmic ...).

- An algorithm does not require "thinking" to be executed.
  An algorithm is (by definition) a sequence of operations which can be performed by a Turing-complete system (i.e capable of performing any computational task. Almost all of today's computers are Turing-complete).

- Provided the same input a given algorithm will always produce the same output.

The above characteristics prove to be crucial when trying to make a computer play Katamino. The only thing to do is to find a "recipe" to make the computer play optimally and as time is not a problem, let the computer look at all the possibilities and keep the best.

Most algorithms used in this project are relatively simple and belong to the class of *search algorithms*.

2.2.1 Backtracking and Brute-force

Brute-force

Brute-force search (exhaustive search) is an intuitive approach to very general problems. The idea is to enumerate all possible candidates for the solution and check if the candidate solves the initial problem.

The main advantages of a brute-force approach is that it will always find a solution if there is one and that it is simple to implement.

However it has one big disadvantage: the time efficiency. As all possibilities are examined, the computation time is proportional to the number of candidates. In real life problems, this number tends to grow very quickly as the size of the problem gets larger. This phenomenon is called "combinatorial explosion".

Brute-force is hence useful only when the size of the problem is relatively small.

This searching method should *not* be confused with backtracking.

Backtracking

Backtracking algorithms are evolved brute-force. In backtracking, lots of candidates can be eliminated without being tested simply using some knowledge of the problem. An example would make things clearer.

If you consider the well known "eight queens problem", one clearly sees the benefit of backtracking.

The idea of the "eight queens problem" is to find a way to put eight chess-queens on a chessboard (64 squares, 8×8) in order that no two queens "attack" each other. It basically means that you have to sort things in a way that there is only one queen per column, row and diagonal.

For this problem, the brute-force approach would require to look at all the positions, thus

$$\frac{64!}{56!} = 178,462,987,637,760$$

Explanation of the formula:
We need the number of permutations of eight queens on a board that has sixty-four squares: $64 - 8 = 56$.

That's brute-force and in this case, the problem would take far too much time to be solved. With a little thought you realize that the order of the queens does not matter. A queen is a queen. So each one of the $178,462,987,637,760$ positions has exactly $8! - 1$ equivalences. The number of positions to look at is therefore reduced to

$$\frac{64!}{56! \times 8!} = 4,426,165,368$$

Explanation of the formula:
We can multiply the denominator of the previous formula by 8!, in order to count as one a set of equivalent positions.

That's a great improvement, but we can do better. If you use the fact that the problem requires no more that one queen per column, the problem is only about permutations of 8 numbers, because there can not be more that one queen per row as well, so the total number of positions to look at is

$$8! = 40,320$$

Explanation of the formula:
We need the number of permutations of eight queens. We don't need to worry about the board anymore. It has been "processed" only by a good knowledge of the problem.

You just have to solve the diagonals for these and you are done. This is trivial with the available technology nowadays (2007). What has been used here is not brute-force anymore. It is called backtracking.

In the case of Pentominoes and Katamino, we can expect to quickly find a winning move using the observations we make on the game. Hence most positions should never be analyzed and the effect of combinatorial explosion would be minimized.

Note that even if the improvement is great when using backtracking, it is very relative. For huge problems, combinatorial explosion is still a big issue.

Chapter 3

Open-source

The aim of this chapter is not to give a full definition of what open-source is. This would be clearly beyond the scope of this report. The goal here is to give a general idea of open-source software and present the licenses we use for this project.

3.1 The philosophy of open-source

> Open source is a set of principles and practices that promote access to the design and production of goods and knowledge.

This is Wikipedia's [30] definition. We are allowed to copy it and include it in this report, because it is open source.

Put simply, the main idea of open source is to let the user of a good have the right to modify and redistribute goods. The idea of open source is most commonly applied to computer software.

The Open Source Initiative [12] (OSI) which, as well as the Free Software Foundation [6] (FSF) for instance is an organization which deals with the open source software issue. The OSI gives the following definition of open source software.

> Introduction
>
> Open source doesn't just mean access to the source code. The distribution terms of open-source software must comply with the following criteria:
>
> 1. Free Redistribution
>
> The license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale.
>
> 2. Source Code
>
> The program must include source code, and must allow distribution in source code as well as compiled form. Where some form of a product is not distributed with source code, there must be a well-publicized means of obtaining the source code for no more than a reasonable reproduction cost preferably, downloading via the Internet without charge. The source code must be the preferred form in which a programmer would modify the program. Deliberately obfuscated source code is not allowed. Intermediate forms such as the output of a preprocessor or translator are not allowed.
>
> 3. Derived Works
>
> The license must allow modifications and derived works, and must allow them to be distributed under the same terms as the license of the original software.
>
> 4. Integrity of The Author's Source Code

The license may restrict source-code from being distributed in modified form only if the license allows the distribution of "patch files" with the source code for the purpose of modifying the program at build time. The license must explicitly permit distribution of software built from modified source code. The license may require derived works to carry a different name or version number from the original software.

5. No Discrimination Against Persons or Groups

The license must not discriminate against any person or group of persons.

6. No Discrimination Against Fields of Endeavor

The license must not restrict anyone from making use of the program in a specific field of endeavor. For example, it may not restrict the program from being used in a business, or from being used for genetic research.

7. Distribution of License

The rights attached to the program must apply to all to whom the program is redistributed without the need for execution of an additional license by those parties.

8. License Must Not Be Specific to a Product

The rights attached to the program must not depend on the program's being part of a particular software distribution. If the program is extracted from that distribution and used or distributed within the terms of the program's license, all parties to whom the program is redistributed should have the same rights as those that are granted in conjunction with the original software distribution.

9. License Must Not Restrict Other Software

The license must not place restrictions on other software that is distributed along with the licensed software. For example, the license must not insist that all other programs distributed on the same medium must be open-source software.

10. License Must Be Technology-Neutral

No provision of the license may be predicated on any individual technology or style of interface.

If the license used by a given program respects all of those ten points, the program could be labeled as open source software.

Open source is a lot more than legal issues. It is a whole philosophy based on freedom, sharing and intellectual non-property.

Knowledge is the only good which increases as we share it.
Marie von Ebner-Eschenbach

This is especially true in the field of computer-science. Writing and debugging a program is a lot of work. Why duplicate it ? It is much better to let people use it and modify it. It allows people to perfect a given program instead of having to write their own which does the exact same thing.

Lots of very well known, high quality programs are open source like the GNU Operating System, the Linux kernel, the OpenOffice.org office suite, the GIMP image manipulation program, the Mozilla Firefox web browser and so on.

3.2 The license

There are many different open-source licenses available. It is important to choose correctly, as it will define the behavior that the users will have towards the software. In the first place, we had selected the MIT license for our project. It is a very short text so it is included here:

Copyright (c) <year> <copyright holders>

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

As you see, the great advantages of this license is that it is very simple and short. It gives a lot of freedom to the user, the only required thing is to mention the original authors of the code. It also protects the authors from all kinds of malfunction. The users can not blame the authors if the Software does not behave in the expected way.

The MIT license satisfied us. But we then decided on a more restrictive and complex, but also more popular alternative: The GNU Public License. [8]

One of the clauses of the GPL is that only GPLed programs can use GPLed code. This is done in order to guarantee that the GPLed content stays open-source.
As some of the code we use (in the libraries) is under the GPL, we had to change the license of our program. The license is a bit more complex but we are sure that the program will stay open-source no matter who distributes it.

It has another advantage as well. As our project is also about open-source software, we wanted to be as close to the "real" GNU philosophy which is the number one reference in terms of open-source as it is possible to be. Choosing the GPL was a step forward in this direction.

It made the choice of the report's license easier. We have chosen the GNU Free Documentation License [7] (FDL) as it is the standard complement to the GNU GPL.

Chapter 4

Choosing a programming language

The C++ programming language was used for the entire code base of the software. C++ is by far not the only existing programming language, so a bit of thinking was necessary in order to make a suitable choice.
Among the parameter we took into account were the following.

1. The knowledge of the language we had at the time.
2. The speed. (Low level compiled languages are usually faster than high level interpreted ones.)

As a matter of fact, we did not want to start with a totally new language, so we had to pick up one at least one of us had already learnt. This restricted our choice to half a few languages (more details on each of them will follow).

1. PHP
2. Python
3. C
4. C++

PHP is a server-side language originally designed to generate web pages, which was not what we wanted. Moreover, it is an interpreted language, lacking the needed speed even if interpreted languages have the great advantage of being fully portable.

Python was a serious candidate. It is easier to use than C or C++ and one would be able to achieve more with less work. However, Python is an interpreted language, implying insufficient performance for a brute-force program.

Of the two remaining candidates, C++ is maybe the most productive and easiest to use, although its speed is marginally worse than C's.

C++ is a compiled language, meaning that the code is directly translated into instructions that can be read by the computer. This is done by a software called a compiler. Once the program compiled, the user can launch it without further delay, and the overall speed is better than that of an interpreted program.

The choice of C++ was a trade-off: this language was neither the quickest nor the easiest to use. Still, it was the best compromise we could find.

Part II

1st Goal: Pentominoes and Bruteforce

Chapter 5

Kinonk: the Software

Kinonk has been designed to play optimally the game of Pentomino and several of its variations (including Katamino).
We had to think a lot and made a lot of planning, trials and errors before finding a satisfactory structure for Kinonk.

We used a lot the fact that C++ is an object-oriented programming language. We wanted to make our program as easily extensible as possible. Ideally, the structure should help adding features without having to rewrite large parts of the program. This proved useful in several occasions in the process of writing Kinonk.

In this chapter some critical parts of the program are presented, but if you need a more specific, technical documentation, read the Doxygen documentation distributed with this report.

Note that it might be useful to use UML (Unified Modeling Language) diagrams in order to improve the structure of a program, but in the case of Kinonk, we considered it to be a too small project to be worth bothering with such advanced methods.

5.1 Features

The first release of Kinonk, the one this project is about, features the following:

- Solving of the games of Pentominoes and Katamino. It is possible to make Kinonk find all the winning moves, not just one.
- Solving of any variation of Pentomino and Katamino through altering:
  - The availability of the pentominoes.
  - The size of the board on which the game is played.
- Make humans play against the computer.
- For convenience, these features are implemented as well:
  - Restriction of the thinking time of the computer.
  - Definition of several critical constants using an external plain-text file (avoiding useless recompiling)
  - Output of statistics in a text file, from which one can generate graphs using a Perl script written for the occasion.

56	57	58	59	60	61	62	63
48	49	50	51	52	53	54	55
40	41	42	43	44	45	46	47
32	33	34	35	36	37	38	39
24	25	26	27	28	29	30	31
16	17	18	19	20	21	22	23
08	09	10	11	12	13	14	15
00	01	02	03	04	05	06	07

Table 5.1: int representation of the squares of a 8×8 board

5.2 Objects

The object-oriented approach is language-dependant. As C++ supports object-oriented programming, we had the opportunity to use the so convenient classes.

5.2.1 Location: Coor

The most basic thing we had to take care of is the representation of a plane.

The reason for this is simply that as the grid and the pentominoes are made of this simple unit: the square. The only thing that makes a square different from another is its location in the plane. To implement this, we have the Coor class.

This class simply represents a square in a two-dimensional space (on the board in this case). It has two integer members: the x and y coordinates in a Cartesian rectangular plane and a few functions to check if a square touches another one, or if one is out of range (not in the grid). There is also function which returns a single integer representation of the 2D Coor object (Table 5.1).

This proves very useful in case you want to use a Coor object as index to a simple c-style array. Note that this works because the size of the board is known and will not change during the game.

5.2.2 The grid: Board

A board is simply represented using a matrix of integers, with the first index representing the x position, and the second representing the y position of it squares.

For example, the integer value of the upper left corner of a 8×8 board would be given by
`board[0][7]`.

Note that the coordinates of the board's lower left corner is $< 0, 0 >$, not $< 1, 1 >$. Now for the content of this matrix: an integer value of -1 means that the square is empty (when outputted, it will simply be displayed as a period); any other value means that it is occupied by a pentomino. In that case, $value = idofthepentomino$. The Board class features, among other things, the useful `playMove` and `playMoveBack` member functions that are described in the appropriate sections and in the documentation.

5.2.3 The move: Move

There is no such thing as a "Pentomino" object in our program. Since a move is simply the position of five squares on the board, a pentomino can also be represented as a given set of Coor objects.

A Move object has two important members: the five squares it takes on the board (a simple array of 5 Coor objects) and the id of the used pentomino.

Pentomino id	0	1	2	3	4	5	6	7	8	9	10	11
Letter	I	L	Y	N	V	P	U	Z	F	T	W	X

Table 5.2: id of each pentomino

Although the latter is not strictly indispensable, it is useful to efficiently detect which pentomino is played and, for example, whether it already lies on the board or not.

The pentomino id is an integer from 0 to 11, assigned as such (the IDs are the same as those in the notice of the standard game of Katamino published by DJ Games minus one) (Table 5.2).

The Move class has other members. For example, it contains a vector with holding the positions of all squares adjacent to the move. Those squares are used when the game is played according to Katamino's rules. We can easily find all moves *adjacent* to this move by using the adjacent squares in conjunction with a special index which contains all the moves overstepping a given square. This index will be introduced in section 5.2.5. The member functions includes a function to get the "equivalent positions" (symmetry or rotation). This function is very useful to avoid analyzing several times an already analyzed moves.

5.2.4 The move: MetaMove

Move objects might have to be copied billions of times in one game. One obvious idea to make the program a bit faster was to make the Move class very light.

That's why we created the MetaMove class, a subclass of Move. In addition to the previously-mentioned data, it also keeps track of the move's status in the position the program is currently analyzing. For example, each MetaMove object holds data about the validity or invalidity of a move (whether it can be played next or not) and its heuristic evaluations given by the sorting algorithm.

Typically, there are two different evaluations the program does: a so-called "static" evaluation, done every time the moves are sorted (e.g the number of replies) and an "adaptive" evaluation, that changes throughout the game and is taken as is by the sort algorithm. This is typically used to keep track of the number of times the move generated a cutoff at each depth (killer heuristic).

5.2.5 The game: PositionData

All the previously described classes are theoretically enough to make the program work in the expected way, but we still need some sort of interface to bind everything together.

PositionData contains all the information about the game.

It has the board, a vector containing pointers to all possible moves on the given board, the list of all the previously played moves and containers of the moves sorted by pentomino identification number or by square.

It obviously contains all kinds of basic and central algorithms that make the Moves and the Board interact: play moves, take them back, sort the moves, solve a position (using backtracking) and so on. Most of the functions described in the next chapters are in PositionData.

5.3 Basic actions

To be able to analyze in detail a lot of possibilities, our program must execute the most basic actions: play a move on the board, and play it back. We also describe how the equivalent positions of a move are found.

		0					

Table 5.3: A square

5.3.1 Play a move and play it back

Note that this chapter only describes the function that plays a move *on the board* (a Board member function). The general `playMove` function includes the member function but also performs actions to eliminate candidate moves. It is described later.

One simply needs to set the value of the squares the pentomino will take on the board from 0 to the pentomino's id. The pseudocode is trivial (Algorithm 2), and playing a move back is just the exact opposite (algorithm 3).

Algorithm 2 Play a move on the board

```
1: for each of the 5 Coor objects composing the move do
2:     board[x position][y position]=pentomino's id
3: end for
```

Algorithm 3 Play a move back on the board

```
1: for each of the 5 Coor objects composing the move do
2:     board[x position][y position]=-1
3: end for
```

5.3.2 Getting the equivalent positions of a pentomino

Let's take a practical example (Table 5.3). We observe the square has exactly 8 positions which are equivalent (Table 5.4)

Those equivalent position are either symmetries or rotations:

- Position 0 is the original square
- Position 1 is the horizontal symmetry of 0
- Position 2 is the vertical symmetry of 0
- Position 3 is the vertical symmetry of 1
- Position 4 is the horizontal symmetry of 5
- Position 5 is position 0 rotated 90 degrees clockwise.
- Position 6 is the vertical symmetry of 4
- Position 7 is the vertical symmetry of 5

Of course one might argue that this only holds if the grid is square. This is true, but a rectangle will only eliminate some of those possibilities and it is thus trivial for our program to handle this special case.

		0			1		
	4					5	
	6					7	
		2			3		

Table 5.4: A square and all its equivalent positions

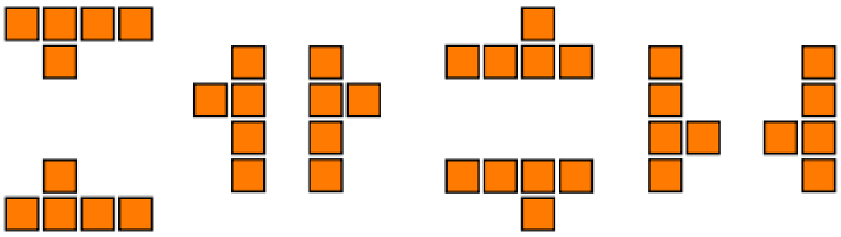

Figure 5.1: All the orientations of the "Y" pentomino

5.3.3 Move listing

Our program uses a system where all the candidate moves of the current position are listed at once, and then tried one after the other. This system may be slower than generating each move on after another, but it has the big advantage that once listed, the moves can be sorted (see Chapter 5.3.5).
The position changes a lot of times while the solving algorithm is running, and every time the position changes, the moves must be listed. This means that the method for getting all the candidate moves must imperatively be efficient.

Listing everything at once

For any game, there exist a list of all moves that could theoretically be played in at least one position. This list does not change once the game has begun. Formally, one can say there exist a set of moves A such that $\forall V_i, \subset A$ where V_i represents the set of valid moves in any given position.

The latter is a very important property; it means we can filter this list to eliminate invalid moves so that all the remaining ones will are valid.

The program generates the list at the beginning of the game. The process is composed of two main steps.

1. The program puts all orientations (see Figure 5.1) of all selected pentominoes in a queue.
2. For each orientation, register all possible positions on the board in the move list.

We have explicitly written the orientations into the program, meaning that it does not have to generate them itself.

They are represented as the most lower left move possible to play with the given orientation (call it M_0). See Figure 5.5 for an example.

Step 2 is completed in the following way: first, the program checks if M_0 fits. If it doesn't, then no positions of this orientation can fit; try the next orientation.

Else, register M_0 in the list of moves, and move it one square right (figure 5.6).

X	X	X	X				
		X					

Table 5.5: The first orientation of the “Y” pentomino, represented as a Move.

	X	X	X	X			
			X				

Table 5.6: M_0 moved one square right

Check if it fits. If it does, move it one square right and check again. If it doesn't, move M_0 one square up (figure 5.7).

Check if this position fits. If not, go on to the next orientation. If it does, move this position one square right. And so on.

Accessing the moves

It is not enough to have the list of all moves that could possibly be candidates, one must be able to find quickly which move is valid and which is not.

A quick re-consideration of the rules reveals that a move can be invalid for two kinds of reasons:

1. Positional reasons, i.e the pentomino overlaps already played figures or it does not fit the board. When using Katamino rules, a move can be invalid because it does not touch an already played pentomino.
2. Availability reasons, i.e the figure is already on the board.

If we use the no-brainer approach, each time a move is played the program will have to iterate through the whole list to look for moves stepping on the same position as the already played one or having the same pentomino id.

X	X	X	X				
		X					

Table 5.7: M_0 moved one square up

Considering there are exactly 2308 moves in the list for a 8×8 board with all pentominoes available, improvements are needed.

For example, it would be nice to get directly all the moves overstepping a given square, and all the moves with a given pentomino id. Luckily, the list of all moves does not change during the game so indexing it is easy.

Our program keeps an array containing a list of pointers on the moves stepping on a given square. The first index is the integer representation of the square. For example, a list of pointers to all moves stepping on the $< 0, 0 >$ square (the lower left corner) would be `square_index[0]`. That's one of the reasons why the function binding a unique integer with each coordinate is very useful.

The software also keeps an array containing a list of pointers on the moves with a given pentomino id, the first index being the pentomino id. For example, The address of all the moves implying the "I" pentomino (id 0) is `pento_index[0]`.

We have now all the tools for the listing of moves to be fast.

Final design

We still have a problem: When we play a move back, how do we know whether we should re-validate or keep invalid the existing moves, without looking at the other played moves ?

Can we simply valid a move because the square it stepped on is no longer occupied ? There could be other reasons on why this move should be invalid...

For this reason, our program uses an additional member in MetaMove, an integer called `reasons_invalid`. If its value is 0, then the move is valid. Otherwise the move is not. Every time a reason is found for the move not to be invalid (stepping on a played moves square, pentomino already used...), then the value of invalid is incremented by 1. If a reason to be invalid disappears for the move, then the value of invalid is decremented by 1.

The program maintains the following variables:

- The list of all moves (`all_moves`)
- The list of all played moves (`playing_order`)
- The per square index of all_moves (`square_index`)
- The per pentomino index of all_moves (`pento_index`)

For more information, you can always refer to the Doxygen documentation of the program.

The list of valid moves will be generated automatically by the sorting function (see chapter 5.3.5).

For optimization reasons, each Move object also contains its position in `all_moves` so that it can be found and modified quickly.

The pseudocode for the general `playMove` function is described in Algorithm 4 . The `playMoveBack` function is similar and described in Algorithm 5.

Of course, those algorithms only take into account the Pentominoes rules (for clarity), but the ideas are similar for the Katamino rules.

5.3.4 Finding the best move

The main issue with brute-force search is always the same: there are a lot of possibilities to look at, whereas the program must find the optimal solution in a reasonable amount of time (or else it won't suit many needs).

Algorithm 4 Play a given move M

```
1: Physically play $M$ on the board using algorithm 2
2: Register the move in the list of played moves (playing_order)
3: for each Coor object of the move as $c_i$ do
4:    for Each move $m_i$ the square index gives as stepping on $c_i$ do
5:       increment $m_i.reasons\_invalid$ of 1
6:       if $m_i.reasons\_invalid = 1$ then
7:          Delete $m_i$ from the list of valid moves
8:       end if
9:    end for
10: end for
11: for Each move $m_k$ the pentomino index gives with the same id as $M$ do
12:    increment $m_k.reasons\_invalid$ of 1
13:    if $m_k.reasons\_invalid = 1$ then
14:       Delete $m_k$ from the list of valid moves
15:    end if
16: end for
```

Algorithm 5 Play a given move M back

```
1: Physically play $M$ back on the board using algorithm 3
2: Remove $M$ from the list of played moves (playing_order)
3: for each Coor object of the move as $c_i$ do
4:    for Each move $m_i$ the square index gives as stepping on $c_i$ do
5:       decrement $m_i.reasons\_invalid$ of 1
6:       if $m_i.reasons\_invalid = 0$ then
7:          Add $m_i$ to the list of valid moves
8:       end if
9:    end for
10: end for
11: for Each move $m_k$ the pentomino index gives with the same id as $M$ do
12:    decrement $m_k.reasons\_invalid$ of 1
13:    if $m_k.invalid = 0$ then
14:       Add $m_k$ to the list of valid moves
15:    end if
16: end for
```

Hopefully, there are positions which do not provide useful information, and analyzing them is therefore a waste of time. Such positions include:

1. Positions leading to a loss, since we are only interested in positions leading to a win.
2. Positions occurring only if we make a mistake, since we have to play optimally.
3. Positions that are equivalent to other previously examined positions
4. Positions that have already been examined but appear after a different move order.

Of course, such positions are not always immediately noticeable. However, the program only needs to find one forced win and can therefore *immediately stop analyzing a position if a winning move is found.* For example, if in a given position you have to look at move a and move b and that move a proves winning, then you do not have to look at move b. This is a simplified version of what is called the *minimax algorithm* [29].

Of course, such a feature would be especially useful if *the best moves were tried first.* In a position with 1000 candidate moves, finding that the first examined move wins will avoid looking at the 999 others, whereas finding that the 1000th one wins means that you have wasted time examining 999 loosing moves.

That is why it is crucial to *sort the moves before analyzing them.*

Consider you have n_1 moves to analyze. For each of these, the opponent can answer with n_2 moves, and you can then reply with n_3 moves, and so on.

If the game lasts in average x moves, you will have to look at $n_1 \times n_2 \times n_3 ... \times n_x$ positions.(Let's say $x = 8$ Let's estimate n_1 to 2000 (as in an empty 8×8 board), n_2 to 1000, n_3 to 500 etc...

The total number of positions to look at would be about $2000 \times 1000 \times 500 \times 250 \times 125 \times 62 \times 31 \times 16 = 9.61 * 10^{17}$.

However, if the moves are optimally sorted, you will have to analyze only one move when it is your turn (although you will still have to analyze all the opponent replies to make sure none refutes your candidate). The total number of positions to look at would be: $1 \times n_2 \times 1 \times n_4 \times 1 \times n_6 \times 1 \times n_8$ and taking the same numbers as before: $1 \times 1000 \times 1 \times 250 \times 1 \times 62 \times 1 \times 16 = 2.48 * 10^8$.

This is the reason why we put a lot of efforts in sorting the moves correctly, and in taking a lot of time for it, especially at lower depths.

However, it may happen that no matter the used method, the program does not have enough computing power and/or time to find the absolute best move. In that case, the program accepts a time limit parameter. If this limit is reached, the software will interrupt its analyze and play the best move it has found so far. In that case, of course, the move Kinonk plays are not proved to be winning and hence the program could theoretically be beaten. Of course, when no time constraint is specified, Kinonk's efficiency is guaranteed.

Sorting the moves and stopping the analyze when a winning move (such a move is called a *cutoff*) is found only solves the first two issues raised above.

We first solved the remaining two by using a table of all positions that have already been analyzed. We observed the issue number 4 (repetition of the same positions with a different move order) did not occur often enough in practice to justify the time cost of using a hash table. Similarly, symmetries are only really problematic as far as the first move is concerned.

Eventually, we decided to program the sorting function to automatically remove symmetrical moves from the returned move list if we are at the root of the tree (i.e at depth 0, where the first move must be played). This does generate any significant time cost, as this happens only once during the entire search.

The possibility of Kinonk analyzing the same position more than once exists, but we decided simply not to do anything against it because we did not find any satisfactory solution that would not slow down the program instead of speeding it up.

Algorithm 6 describes the whole solving process. 1 is returned if the current position is winning for the side to play, -1 otherwise.

Algorithm 6 Pseudocode of the solving function.

```
1:  if No moves are possible in the current position then
2:      Return -1
3:  end if
4:  Get a sorted list of valid moves
5:  for Each move as $m_i$ do
6:      Play $m_i$
7:      Get the evaluation of the resulting position, using this same algorithm
8:      Play $m_i$ back
9:      if the evaluation we got = -1 then
10:         The opponent lost after the move has been played: return 1
11:     end if
12: end for
13: Return -1
```

5.3.5 Sorting the moves

In this part, we deal with the different techniques the program uses to sort the moves, explain their strengths and weaknesses, and the results they have produced. Since such methods are only approximate and do not always give good result, they are called *heuristics*.

Kinonk can be customized by the user to use a variety of sorting methods using an external plain-text file. See the manual in chapter 15 for more information.

The killer heuristic

A "killer" is a move which produced a cutoff (it proved to be winning, so that the other moves in the position did not need to be checked) at a given depth.
The program keeps track of the number of times each move generated a cutoff at each depth, and select the two of them which have the highest value. They are then put on top of the move list to be tried first.

By doing that, we assume that a move that has produced a cutoff in one branch of the tree is likely to produce another cutoff in the current position.

The killer technique is very cheap (it does not take much time), and gives good results in general.

However, it only gives us the first two moves to try, the others remaining unsorted. Some of our experiments have shown that putting more than two killers moves on top of the list also has drawbacks and generally lead to a slower solving time.

The number of replies

This is what Kinonk uses by default. The program simply sort the moves according to the number of remaining candidate moves once the move has been played, with the moves with the lowest number of replies being analyzed first.

This technique assumes the following about a move which has a low number of replies.

1. The move is more "aggressive", and therefore more likely to win
2. Less positions will have to be analyzed with this move than with other ones.

Although the method is more or less cheap in computing time, it does not always prove efficient (assumption number 1 is especially tricky).

For example, our program tried to solve a 6×6 board, all pentominoes being at the player's disposal. The winning move turned out to have only the 90th number of replies, out of 132 moves.

The "wins ratio"

The idea is to play a small number of games for each move to be sorted, and to use the games wins/defeats ratio to sort the moves.

This is done by analyzing only n replies at each depth, with n a small number. The program then calculate $wins/totalnumberofgames$ and sort the moves accordingly, in descending order.

The n moves are usually chosen randomly.

Another implementation leading to different results was suggested by our tutor, R. O'Donovan. This time, n moves are played randomly from the current position before completely solving the resulting board. This process is repeated m times and the ratio that is used is $wins/m$. The candidates are then ordered in descending order.

Both implementations take about the same time, and perform somewhat equally.

In our opinion, the wins ratio is sometimes more reliable than the number of replies but take far more time and must therefore be reserved for the very first depths.

We have observed that the wins ratio's efficiency vary from setup to setup and that's why Kinonk does not use that method by default.

5.4 The User Interface

When Kinonk has to play against human players, it expects input from the players. The system Kinonk uses is similar to the command line. When prompted, the user has to type a letter and hit the `<Return>` key. According to this letter, the program will, either ask for further input or perform an action.

For example, you are playing and it is your go, if you hit 'p' and `<Return>`, Kinonk will ask for five coordinates. If you then input the five coordinates of the move you want to play, the program will play it for you and go on with the game.
If instead of inputting 'p', it was 'c', Kinonk would have solved the position and played the best move for you.

Note that if you do not know the commands, you can input 'h' and the program will display a help.

5.5 Libraries

Libraries are pre-compiled files that contain code that can apply to a variety of situations. Using libraries, programmers can avoid re-writing the same code every program uses over and over again (e.g like a function printing something on the screen). If you choose your libraries well, you can end up with very powerful functions that do exactly what you need. It saves a lot of work.
For this project, all the libraries we used are open-source and thus perfectly compatible with Kinonk.
Here is a description of the three libraries we used.

5.5.1 Datastructures and Basic algorithms: STL

The STL (Standard Templates Libraries) is the standard library used in C++. This means that it is distributed with most C++ compilers and is developed in a centralized way. It contains several useful datastructures like vectors (extensible arrays), queues, stacks and so on. It features also all kinds of basic algorithms in the fields of mathematics, time handling, handling of the data structures, input and output streams and so on.

5.5.2 Command line parsing: tclap

We used tclap (originally written by Michael E. Smoot). The options given on the command line to Kinonk are parsed and interpreted by this library.

5.5.3 File parsing: ConfigFile

ConfigFile (originally written by Richard J. Wagner) is used for the file parsing. When you specify an external configuration file with the `-a` or `--algo` option (see the manual in 15), this library handle the parsing of that file.
This library ignores the comments, “links” the name of the variables with their value and assigns them to the right constant in Kinonk.

Chapter 6

Results

6.1 Who wins

Kinonk can be told to *solve* a board, that is to find who is winning and output one or all possible winning moves. As expected, Pentomino (and Katamino) are first player win *for most setups*. Here, we consider some boards whose solving results we found interesting. For all the boards we present in this section, all pentominoes are available and the Katamino rules do not take effect. At all depths, the moves are sorted using the number of replies heuristic (see chapter 5.3.5).

The 8×8 board was solved on a 1.6 Ghz computer while all the other boards were processed by a modern (2007) PC, with an Athlon 5000+ dual core processor.

Table 6.1 is an overview of our results, where:

- N is the number of candidates
- n is the number of candidates without considering equivalences (e.g symmetries or rotations of the board)
- w is the total number of winning moves, without considering equivalences
- p is the position between 1 and n of the first found winning move in the list of distinct candidate moves.
- t is the time the program spent before finding a winning move.
- T is the total time the program spent.

What follows are some comments on each result.

Size	N	n	w	p	t	T
3×3	49	10	10	1	less than 1 second	less than 1 second
5×2	44	12	12	1	less than 1 second	less than 1 second
4×4	244	33	0	-	-	less than 1 second
5×5	571	82	1	16	less than 1 second	less than 1 second
6×6	1024	133	1	90	7 seconds	12 seconds
7×7	1603	217	32	13	17 minutes	10 hours
7×8	1924	492	0	-	-	45 hours
8×8	2308	296	at least 1	2	18 days	not tested
11×11	5179	676	probably 0	-	not tested	not tested

Table 6.1: Overview of the solving results for different configurations

X	X	X
		X
		X

Table 6.2: One of the 3 × 3 board winning moves

X	X
	X
	X
	X

Table 6.3: One of the 5 × 2 board winning moves.

6.1.1 Three by three: the smallest one

This is the smallest possible square board one can play with. There is enough room just for one pentomino, so of course all the moves win.

This is the only reasonably sized square-shaped board where all moves are winning.

6.1.2 Five by two: the mirror

We saw that in boards that have only room for two pentominoes, the second player wins. The boards that have *exactly* room for two pentominoes thus 10 squares have an interesting property. The first player can force a win.

He has two ways to do so:

- Use a strategy consisting of splitting the board in half so that the opponent does not have any room left.
- Use the fact that pentominoes are available only once.

The first way is quite straightforward. The second is a bit more interesting. If first players play so that the five squares that are left are contiguous, they will win as well. This is because the five remaining squares would mirror the played pentomino's shape (see table 6.3). In other words, only the pentomino that has been already played would fit this space and as pentominoes are only available once, the second player loses.

6.1.3 Four by four: the exception

This is the only square shaped board that we have proved to be a second player win.

As for the 3 × 3 board, the result depends a lot on the available space. The second player wins because player 1 cannot take enough space to prevent player 2 from playing. The reverse strategy also fails: If player 1 tries to leave a lot of space for player 2, the space left after player 2's move will be too small to play a move (Table 6.4).

6.1.4 Five by five: quickly solved

Even if it is small and thus instantly solved, this board is the first one with which we have a real need of analysis. There is only one winning move (shown on table 6.5). This move literally cuts off the board in half, so we deduced a strategy might be to try taking as much space as possible in order to reduce the opponent's choice.

X	X		
X	O	O	O
X	O		
X	O		

Table 6.4: One possible final position for a game on a 4×4 board

		X		
		X		
		X		
		X		
		X		

Table 6.5: The only winning move for a 5×5 board

6.1.5 Six by six: on the edge

This setup is that on which we did most of our testings. It does not take much time to be solved but is still an interesting test to our algorithms.

This board appeared interesting at the time, because it contradicted much of what we had previously observed. This setup only has one winning move, and it is placed on the board's edge (table 6.6). This makes this move appear among the last ones if we sort the moves using the number of replies. It is a good example to show that this simple heuristic is not always the best way to go.

It might also be worth mentioning that while the 5×5 board's winning move literally tries to remove as much space as possible, this move is almost its opposite. Our conclusion is that it seems to be another strategy, which consists in leaving as much room as possible to your opponent, hoping that he will get stuck by himself.

6.1.6 Seven by seven: the limit

The time needed to solve a board increases exponentially when increasing the size of the board. When considering only square-shaped boards, the 7 by 7 is the largest one that can be run in a "reasonable" amount of time on today's hardware.

There are numerous winning moves. Some of them are on the side of the board, but others are also centered so it is difficult to make any observations here.

6.1.7 Eight by eight: first player win

This is the classical board on which pentomino is normally played. It was first solved by Hilarie Orman [11]. Our program echoes Orman's results: it found the same answers to the move which allows the minimum number of replies (see table 6.7) and found a winning move similar to the one which is suggested in Orman's paper (see table 6.8).

				X	X
				X	
				X	
				X	

Table 6.6: The only winning move for a 6×6 board

O			X	X			
O				X			
O				X			
O				X			
O							

Table 6.7: The first player move that allows the minimum number of replies, and its refutation

				X	X		
					X		
					X		
					X		

Table 6.8: The first player moves that Kinonk proved to be winning

Kinonk spent what one would call an unreasonable time on this board's solving, and still only managed to analyze two moves, with one being winning. This setup clearly shows us that our brute-force algorithms are limited by *the time the user is willing to wait* before getting a result. We think most users would rather have the program play a move quickly, so by default Kinonk plays a move after 60 seconds of searching, whether it solved the board or not.

When we ran our first solving in April; the program spent 48 days without finishing. We then managed to improve the program's speed and the second solving lasted "only" 18 days. Even though the time taken has been divided by more than 2.5, it remains impossible to wait that long for a human player. That is why it would be unreasonable to spend a lot of time trying to improve the brute-force's speed. Even a solving in one day would be too much of a burden for a normal user.

If we do some geometrical considerations, we see that the winning move is a translation of the first analyzed move (shown in table 6.7) of 1 square to the right. This must certainly reduce the effects of the annoying replies the program found to the previously analyzed move.

6.1.8 Eight by seven: a surprising exception

This board has the same property as the 4×4 one but obviously not for the same reasons. Perfect play on this board from both sides leads to a second player win. There is *no* winning move. It is worth mentioning that this applies even if the Katamino rules are used.

It is difficult to find a convincing explanation. If we try to relate these results to those of the 8×8 board, we see that placing the 'I' pentomino in the edge in reply to some centered move (table 6.9) must certainly have a more devastating on an 7 by 8 board than on an 8 by 8 one (table 6.7). Perhaps this simple geometrical consideration allows just one pentomino less to be put on the board.

6.1.9 Eleven by Eleven: over the top

Every time before beginning to run the brute-force search, our program does a few checks to ensure the users will not have to wait too long for their setup to be solved. They include a check where 100 random games are played, and if all of them end up with all pentominoes on the board, then the program deduces there should be

O		X	X			
O			X			
O			X			
O			X			
O						

Table 6.9: Comparing the the 8×8 board (figure 6.7) with the 7×8 one

something wrong and output a warning. It means the board's size is too large to have any practical interest, not considering the huge amount of time it will take to be solved.

Kinonk first begin to issue such a warning when we reach the 11×11 size. This is a board containing 121 squares. Thus we decided to make the program automatically refuse to consider any board with more than 120 squares.

6.2 Statistics

When enabled, the “stats” option of the program creates an additional file just before exiting. The detailed description of what is to be found in this file is in the software's man page (see chapter 15). We used the content of this file in order to get accurate facts about our algorithm's efficiency and also on the game of Pentominoes as well.

If you try to make Kinonk produce statistics for you, you might observe some small differences between yours and ours. This is because the algorithm which takes care of the moves sorting uses the qsort algorithm from the standard C++ library. If some elements in the list qsort has to sort compares equally, then their final arrangement is undefined. Since, on some machines, qsort randomly determine the value of a variable called the pivot, then some move with the same number of replies might end up being sorted in different orders if you run the program several times.

As always the more information we have, the more accurate interpretations can be. That's why we have chosen to analyze in details the situation where you play Pentomino with all pieces on a 7×7 board and the case where you play Katamino with all pieces on an 8×8 board. We unfortunately could not get statistics on the solving of an 8×8 board using the rules of Pentominoes in time for this report.

In short, we will consider two games:

- Board of size 7×7
- All pentominoes are available
- Pentomino rules
- find *all* winning moves

called game A in the next pages

and

- Board of size 8×8
- All pentominoes are available
- Katamino rules

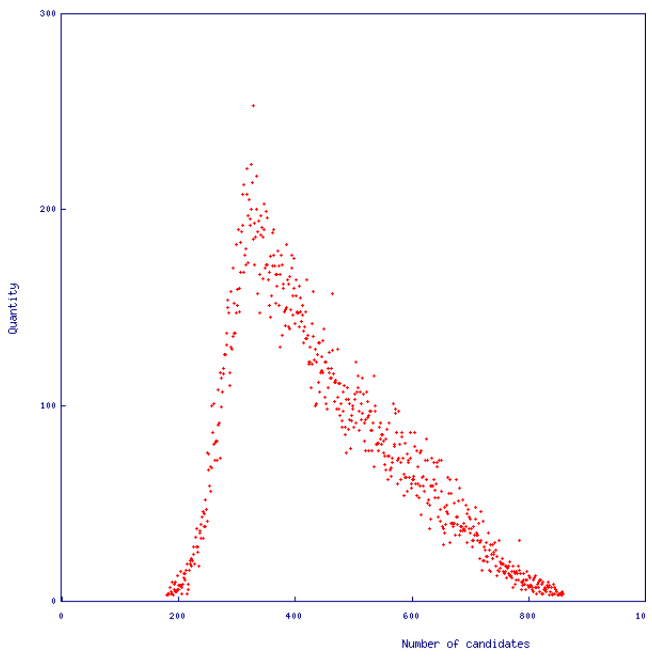

Figure 6.1: Distribution of the number of candidate moves at depth 2 in A

- find *only one* winning move

called game B in the next pages

Both of those games have been proved to be first player win by our program.

The term "depth" is recurrent in this chapter. It is the same as the number of moves that have been played so far. Thus the number of positions analyzed at depth 0 would be the number of positions the program considered when no moves have been played yet (equal to 1, of course).

6.2.1 Number of candidates

In figures 6.1 to 6.7, we observe that the graph takes more and more the shape of a curve as the game goes on (the deeper in the tree we are). We think it is partly due to the fact that we have more data (i.e. more positions are analyzed at depth 6 than at depth 1 because of combinatorial explosion). Still, it is remarkable such a curve does not have any irregularities. It shows the pentomino is, after all, truly a logical game.

Another, logical, phenomenon is that the maximum of the curve moves towards 0 as the game goes on. This is because there is progressively less space and fewer available pentominoes.

6.2.2 Cutoffs

An interesting thing to look at is the shape of the pentominoes the most likely to produce a cutoff in the tree (figure 6.8 and 6.9). This could help us to better understand the game and thus improve our heuristics for sorting the moves.

In figure 6.9, we see that pentomino "F" has produced only one cutoff. This is is not because it is particularly inefficient, but because the only first move the program considered during the search used the F pentomino. Therefore it could not use it for its analysis at higher depths.

We see that the results are not similar for each boards.

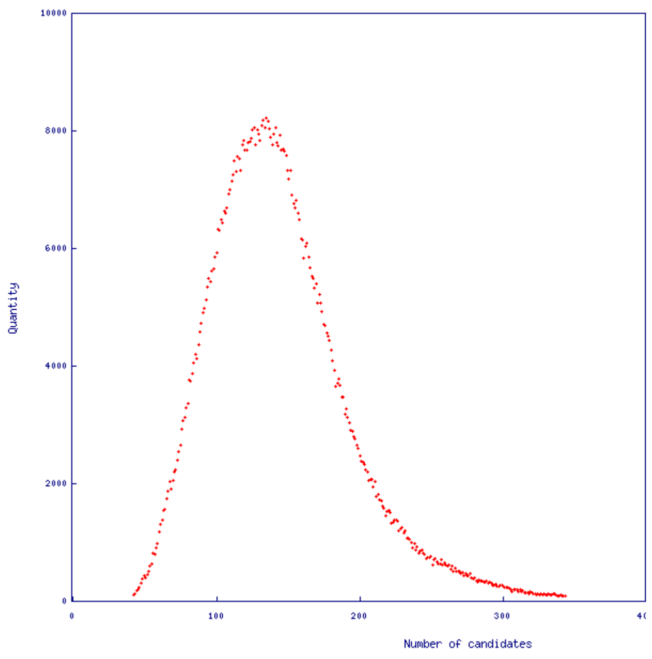

Figure 6.2: Distribution of the number of candidate moves at depth 3 in A

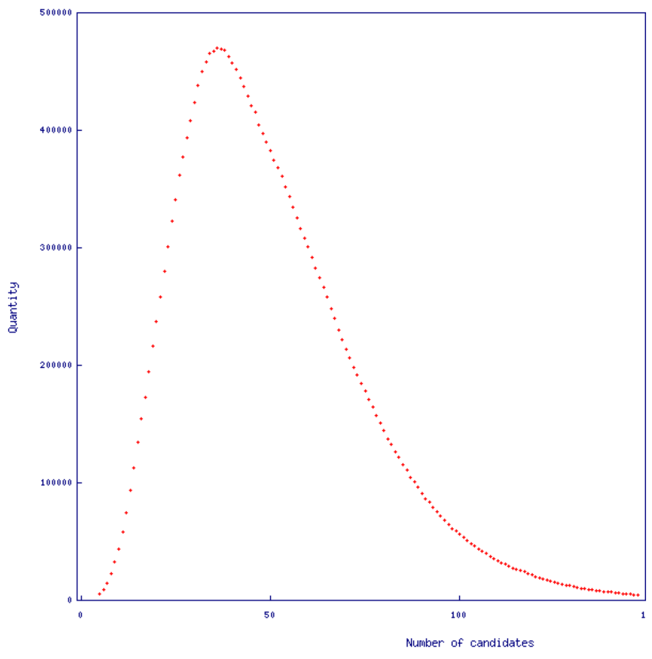

Figure 6.3: Distribution of the number of candidate moves at depth 4 A

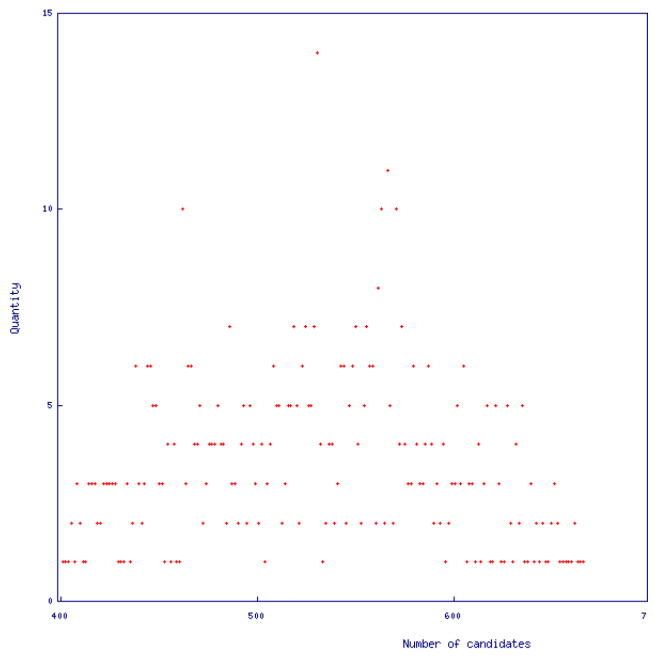

Figure 6.4: Distribution of the number of candidate moves at depth 2 in B

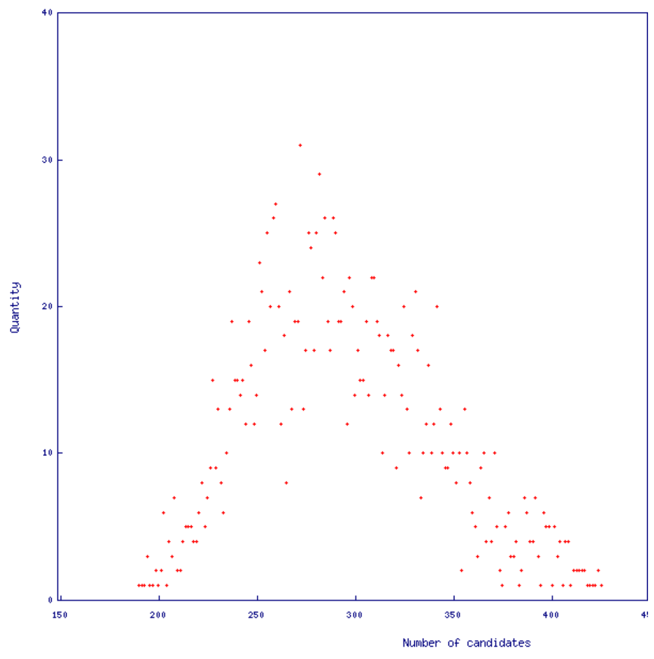

Figure 6.5: Distribution of the number of candidate moves at depth 3 in B

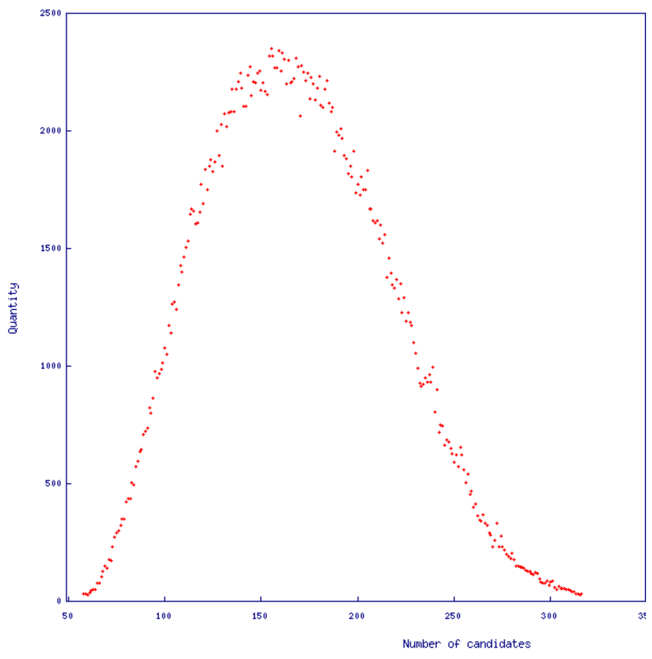

Figure 6.6: Distribution of the number of candidate moves at depth 4 in B

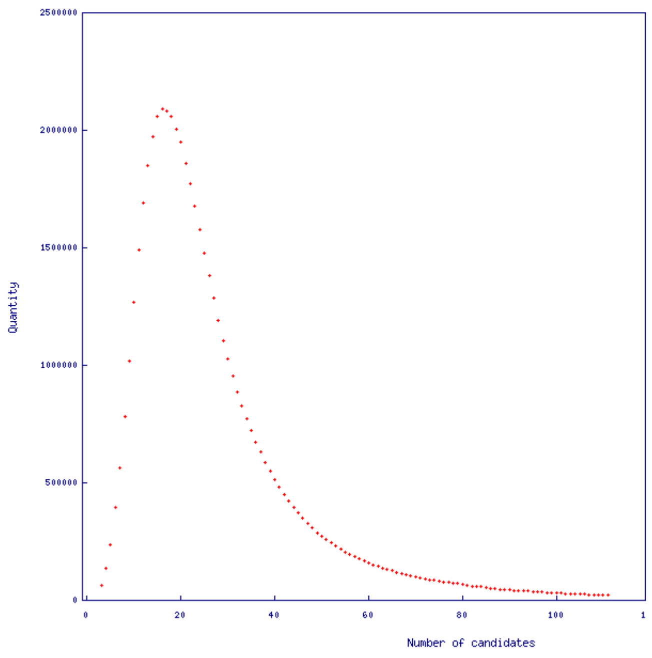

Figure 6.7: Distribution of the number of candidate moves at depth 6 in B

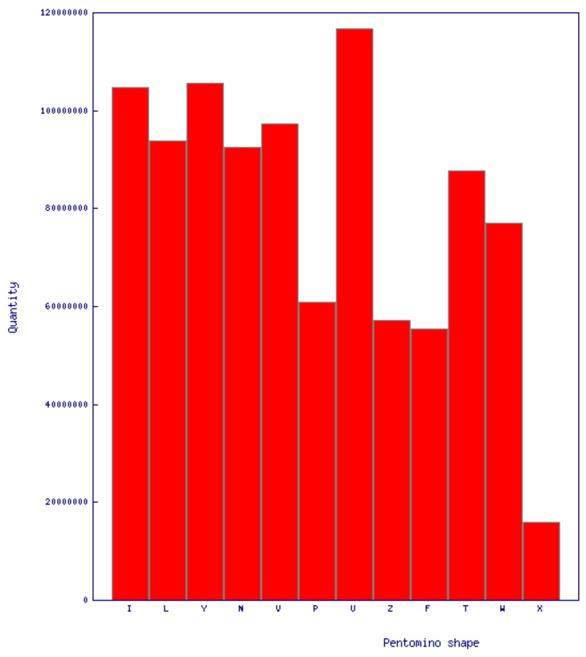

Figure 6.8: The number of times a given pentomino has produced a cutoff in the tree in (A) all depths combined

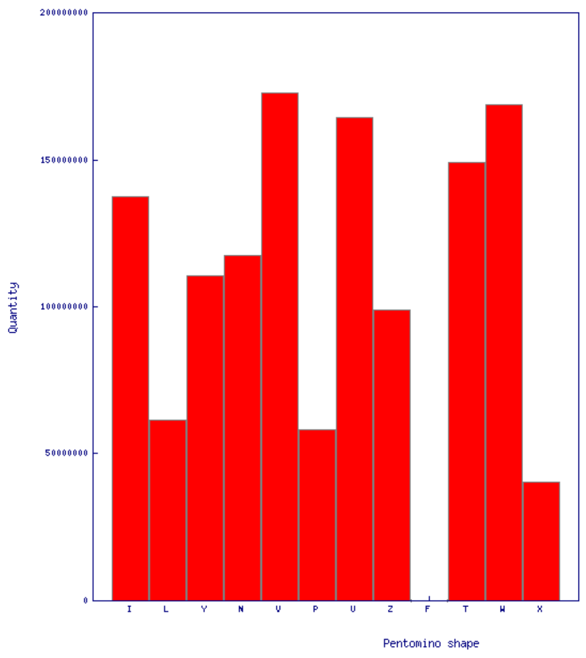

Figure 6.9: The number of times a given pentomino has produced a cutoff in the tree in (B) all depths combined

Setup	Minimum number of leaves	Analyzed positions	Maximum number of leaves
A	$2.1 \cdot 10^6$	$1.6 \cdot 10^9$	$5 \cdot 10^{14}$
B	$1.2 \cdot 10^8$	$2.2 \cdot 10^{10}$	$8 \cdot 10^{16}$

Table 6.10: Minimum, analyzed and maximum number of positions for each setup

We think this is because this properties of the various pentominoes are dependant to the size of the board. For instance, on a 3 by 3 board, all fitting pentominoes would generate a cutoff (all of them are winning).

The only similarity is the “X” pentomino: it seems to be one of the least efficient in most cases.

Our first guess was that this might have something to do with the fact that this pentomino has a relatively high number of axes of symmetry. This does not appear to be an explanation because other symmetrical pentominoes seem to give pretty good results (at least in the two cases above) such as “I”, “V”, “W” and “U”.

Another guess was that the “X” pentomino was one of the most “squarish” pentominoes. It thus may leave more room for the other pentominoes, while being itself hard to find room for.

In the same way as some geometrical property of the 8 by 7 board makes it a second player win (as explained in 6.1.8), it seems that some of a board’s quirks make some pentominoes *efficient* and some others not.

6.2.3 Number of analyzed positions

Figure 6.10 and 6.11 shows the number of positions Kinonk effectively did analyze compared to the total number of positions it could have had analyzed. The latter is *not* the total number of leaves of the search tree, shown in table 6.10. The maximum number of positions the program could have had analyzed at depth d is simply the number of positions that would have had analyzed had there been no cutoff found at $d-1$. The maximum and minimum number of leaves are calculated as follow.

$max = n_1 \cdot n_2 \cdot n_3 ...$ while $n_i \neq 0$
$min = 1 \cdot n_2 \cdot 1 \cdot n_4 ...$ while $n_i \neq 0$

n_i is the average number of candidates at depth i

$min \approx \sqrt{max}$

The number of analyzed positions given in table 6.10 is the total for all depths. It is obtained by adding the number of analyzed positions at each depth.

While the total number of analyzed positions seems pretty large, it actually is only a fraction of what the program could have had to analyze, had it not been able to cut off the search tree. Those results show that Kinonk finds cutoffs in the tree among the first moves, especially at the high depths.

It is however difficult to make something out of this observation: does it mean our sorting algorithms are very efficient, or that there is a lot of possible winning moves at high depths ? We think it might be a mixture of both...

In figure 6.12, we can clearly see most cutoffs are between the first and twentieth position in the move list, but we cannot deduce it is because our sorting algorithm is efficient, for example 90% of the moves could already lead to a forced win at that point. The only way to know for sure would be to analyze all moves at all depths. As figure 6.10 shows, the search space would then become 10^5 to 10^7 larger, so this solution isn’t really worth considering.

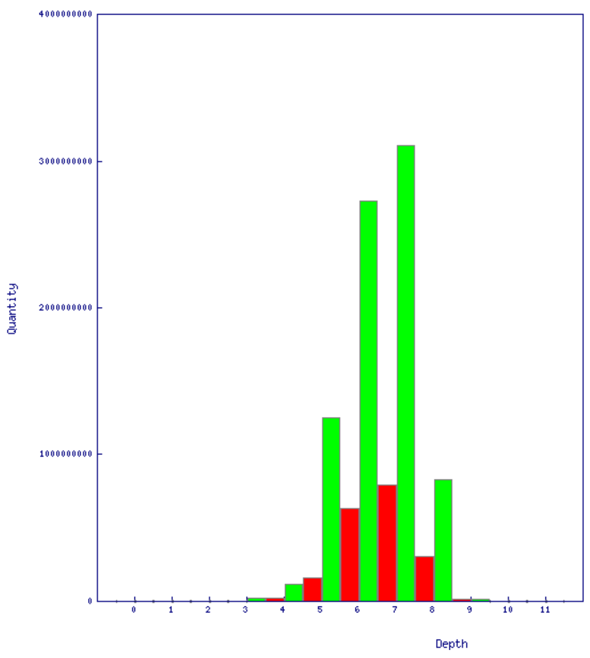

Figure 6.10: Total number of analyzed positions at each depth compared to the maximum number of positions that could have been reached from the previous depth. (A)

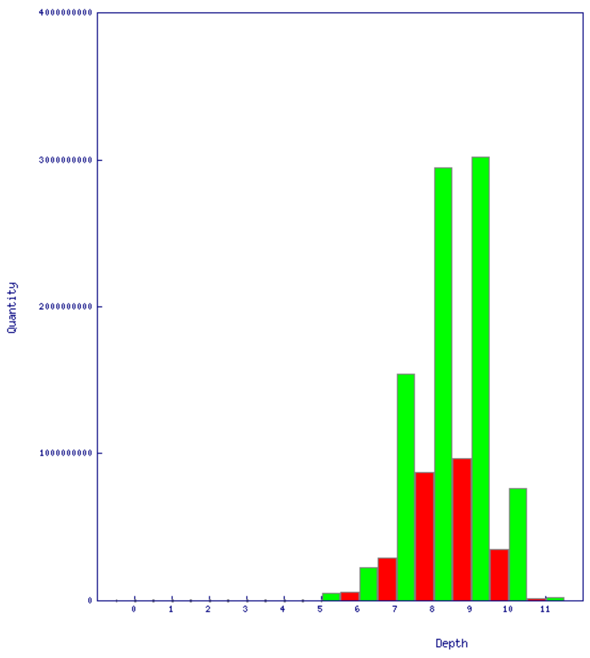

Figure 6.11: Same as figure 6.10 but for game (B).

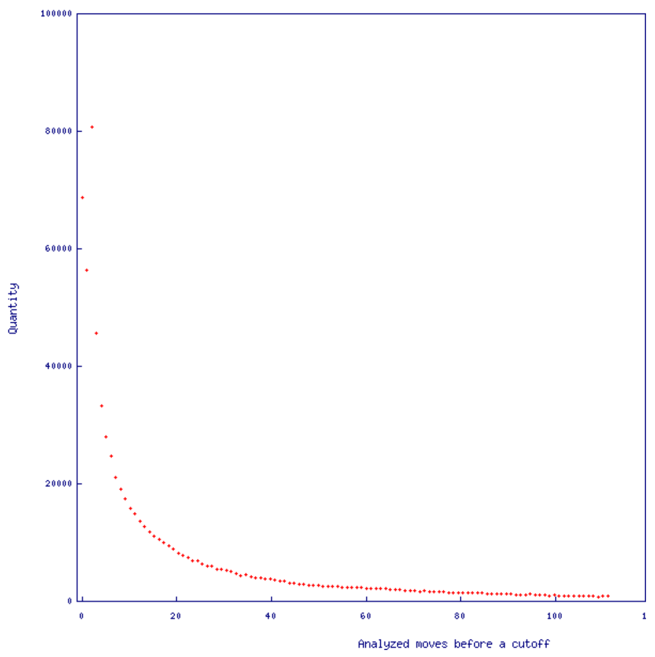

Figure 6.12: number of moves analyzed before finding a move producing a cutoff at depth 3 for (A)

6.2.4 Game length

Referring to figure 6.10 and 6.11 again, one can deduce the average length of a game. For setup A, it seems to be between 6 and 9, probably averaging at 7 . We can see it by finding at which depth the maximum number of analyzed positions was reached.

As far as setup B is concerned, a game seems to last between 8 and 11 moves, the average being 9. This is strange, because the moves in Katamino are somewhat more restricted than in Pentominoes and thus a Katamino game should last slightly longer than a Pentominoes one. Hilarie Orman said in [11] that the average length of one Pentominoes game has been "empirically" deducted to be 10. We unfortunately do not have any information on how that value was computed. Maybe this number is valid for games between humans, but it seems that most games Kinonk considered ended one move before.

6.3 Improving the sorting algorithms

Back in April 2007, we had our first relatively fully-featured stable version of Kinonk (at the time, the moves were sorted using their number of replies at each depth. This default has not changed since).

The next goals were set as follows.

1. Add new features, such as statistics-keeping and Katamino rules support.
2. Increase the program's speed.

Although goal 1 was fulfilled, the program's speed has not really increased (in fact, it has even slightly slowed down, due to the addition of time-consuming new features such as support for time limits).

We first tried to optimize the general organization inside the program, using a profiler to know where to concentrate our efforts.
The performance gain was negligible. In a brute-force implementation, the only thing you can really improve is the way the moves are sorted. This section describes some of the strategies we (unsuccessfully) tried.

6.3.1 Time-consuming methods for the first depths, fast ones for the rest

This consists of using the wins ratio, or some variant of it, for sorting the moves at depth 0 or 1 and using the number of candidates for sorting higher depths. While this may work fine in some settings (especially for large sizes), this can take a large amount of time for sorting the moves. This amount of time would be better used for searching the tree, especially if you consider low board sizes. Another problem is that although the position of the winning move might be improved, moves coming before it are going to be tougher and hence take a lot of time to be analyzed. For example, consider a position with four moves, *a*, *b*, *c*, and *d*. Assume *a* is a winning move, *b* a tough, but losing move and *c*,*d* losing moves which can be countered by many different replies. The wins ratio method might sort the moves in the following order: *b*,*a*,*c*,*d* while a faster method might return *c*,*d*,*a*,*b*. While the first possibility is objectively the best one, the second will probably end up taking less time to be analyzed.

6.3.2 No (or random) sorting at all depths

In most case, this is the slowest method of all. While the program takes almost no time at sorting, it takes a long time at analyzing worthless moves it would not have considered if using other methods. It is worth noting that using the number of replies does not take much sorting time, and is much more efficient than not sorting the moves at all.

6.3.3 Alternating descending and ascending order using the number of replies

In our opinion, this is one of the most interesting idea we had. While observing the logs of Kinonk, we realized that the program was either playing a move leaving a lot of room for other pentominoes or on the contrary a very central one.

We deduced that they were two strategies for playing well:

- Let the opponent put his pieces on the board and take space himself by leaving him as much space as possible and hoping he will get stuck.
- Take as much space as possible with your own moves, restricting the opponent's space as much as possible.

It seems that both players cannot use the same strategy (or at least that's what we observed). Some examples were the only winning move on a 6x6 board (one the side) and the winning reply to the most central 8x8 board move (also on the side).

Although we were very exited about this method, it gave only mixed results. It seems that it only applies to low depth, and that it does not work well with some apparently unrelated configurations.

Chapter 7

Possible Extensions

7.1 Large lookup tables

Since we know have solved most board configurations using all pentominoes, the program should not have to do this work over again. We could store the winning move for each configuration in an external file that Kinonk would just have to read to know what to play.

For some setups, like the 8×8 using the Pentominoes rules, it takes a lot of time even to find some move at depth 1 or 2 , so those moves should be stored as well.

If a configuration the program is asked to solve a configuration that is not in its lookup tables, then the result will be appended to it after the solving.

Such tables are used in a lot of board games such as chess or checkers so that the program can play perfectly any ending with only a few pieces on the board.

Those tables can grow very large, and then there is the problem of the time cost of looking them up which cannot be neglected.

In the case of Kinonk, if we refer to the minimum number of leaves in table 6.10, we see that such a table would not take that much space, so it could be an extension idea for our program (not that it is very interesting to implement, of course).

7.2 Artificial intelligence

Kinonk can use several different methods to sort the moves. We tried different settings for each depth in order to find the best possible way (refer to 6.3). Still, it seemed that the more complex and time-taking the methods are, the more inefficient for most settings they become. In layman's term, using an exotic and time-consuming method to sort the moves might be very efficient for a precise board, but will generally give bad results when applied to other configurations.

Eventually, we chose to use the number of replies method by default, for all depths. We still give the user the possibility to choose the method he would like to use for each depth in an external configuration file.

We have come to the conclusion that it is very difficult for a human to find rules that would give us a clue of a move's value without analyzing all its replies. A lot of factors enters into consideration, and it is difficult to prioritize them correctly.

Still, we think such rules exist. The solution would be to use machine-learning to find them. Concretely, here are some of the ideas we had.

- Make Kinonk rate and rank moves in order to remember which ones work the best. Kinonk would begin by playing random moves and seeing whether it works or not. The more Kinonk plays, the more it "learns", the ranking of the moves is more accurate and therefore the better it plays.
- Make Kinonk recognize "groups" of moves and consider in each position what "kind" of move works the best. Here, the terms groups and kind are yet to be defined formally...

Such a feature is not at all easy to implement. Artificial intelligence is one of the most complex branch of computer science. It is clearly out of the scope of this project which was to create a simple brute-force implementation. When we first started the project, we thought brute-force alone would be able to solve the game. This proved to be true, but the thinking time required was tremendous, and no human has the patience to wait for so long. Hence we discovered that brute-force and artificial intelligence should be used in conjunction to minimize the time the program takes while keeping on playing perfectly. For some configurations, it might still not be enough. The user might still sometimes have to wait for a very long time before a move is proved winning.

Therefore, another idea would be to abandon brute-force altogether and rely exclusively on artificial intelligence for finding the best move. In that case, Kinonk becomes fast in all circumstances but we lose the desired perfect play.

7.3 Parallelization

One of the advantage of the brute-force method is that it can (theoretically at least) easily be parallelized. This would especially be useful for multi-processors machines, but since they are becoming mainstream nowadays this feature could really represent a significant performance gain.

Concretely, Kinonk will use only one processor if it is ran on a multi-processor machine. A parallelization would divide the time it takes to finish by the number of processor...

Parallelization would best be implemented by splitting the work (the moves to analyse) between the different processors. For example in a two-processors system, the first processor, p_1 will have to analyse the moves m_1 to $m_{\frac{n}{2}}$ while the second one would analyse $m_{\frac{n}{2}+1}$ to m_n (with n being the number of moves).

We decided not to implement parallelization, as it often is tricky (because of possible race conditions) and might have required changes in the program's internal organization. Moreover, parallelization methods often differ from one operating system to another and it was thus very difficult to parallelize the program and keep it portable at the same time.

7.4 Graphical interface

This is unrelated to the main goal of this work, but it is in our opinion a must have in order for beginners to be able to play against Kinonk. It simply isn't convenient to enter coordinates by hand, without being able to clearly now which pentominoes have already been played.

Even though a GUI has not been implemented yet, Kinonk features a command-line option that makes it display only its moves, without any board or help text. This makes it very convenient for a GUI to communicate with our program.

Chapter 8

Conclusion

By opposition to some other board game like chess or go, where it is easy for experienced players to subjectively evaluate a position without calculating several moves in advance, it is very hard to evaluate a Pentominoes or Katamino position without considering all the possible moves there is.

Furthermore, the effects combinatorial explosion that makes a game such as chess unsolvable are limited in Pentominoes because the number of candidate moves strongly lessens as the game goes on.

However, even though our program is able to solve any board, the time it takes to do so on 2007 hardware (e.g. 48 days for the solving of an 8×8 Pentominoes board) is often unreasonable for the average user. We are really at the limit of what brute-force can or cannot do.

This does not mean Kinonk is not able to play against a human: our program can be given an arbitrary time limit to play its move. Even then, it remains an extremely strong player: we think it would beat most (unprepared) humans on an 8×8 board with its default time limit of 60 seconds. In that case though, the perfect play is lost but the possibility of being able to beat the program makes it more fun for its opponent.

The first step towards implementing an efficient brute-force algorithm is to use the minimax technique: in our case, this is equivalent to simply stop the analyze once a winning move is found.

The second step, and by far the hardest one, is to be able to sort the moves optimally, so that the minimax algorithm can deliver its best performances. To do that, we need to be able to estimate a position's evaluation in a quick and reliable way, using heuristics. Example of the heuristics we used was the number of replies, the killer, or the wins ratio.

Eventually, we discovered that we would not be able to find an optimal heuristic to sort the moves, only a computer could. We therefore strongly believe the best solution is to make the program learn how to play, using machine learning.

In a world where time efficiency is a must, it sometimes is good to take risks especially against a human player who can (and will eventually) fail.

That's why we think it is worth considering completely dropping brute-force and use machine learning to do the whole decision process. This would also concur to making the program more "human" in the sense that it will also sometimes make mistakes.

Part III

2nd Goal: Managing the development

Chapter 9

The two models

In 1999, Computer Programmer Eric S. Raymond published “The Cathedral and The Bazaar” [4], an essay contrasting two different open-source development models:

1. The Cathedral model, in which only a restricted group of programmers works on the code between releases. Only the releases’ source code is available to the end users.
2. The Bazaar model, in which the code is developed in view of the public by a large number of developers from around the world. Anyone wanting to participate can do so very easily. One of this model’s main advantages is, according to Raymond, that bugs can be found very quickly.

Although Raymond exclusively argues for the bazaar model, we chose to develop Kinonk following the Cathedral model. Our reasons were as follows.

1. The requirements of this TM itself. As a matter of fact, we had to do most of the work ourselves.
2. As no other software close to what we were looking for existed, we had to start from scratch. In our opinion, confirmed by Raymond’s essay, one of the condition for a bazaar model to be possible is a software that users can run.

Concretely, we decided for the following rules of thumb:

1. We wouldn’t do any release before handing the TM back. In the meantime, the code would still be available to the public, although no advertisement would be made.
2. We wouldn’t accept any contribution before the first release was made.
3. Most of the information related to this TM were either kept along with the code, or kept private (e.g emails, notebook).
4. All the code that one of us would write had to be reviewed by the other.

This approach worked reasonably well, although maybe not as impressive as the bazaar-like approach of open-source project like the Linux kernel or the Mozilla Firefox web browser. One of Raymond’s main argument against the Cathedral model is that of Brook’s law: The greater the number of people working on a project, the harder it is to coordinate and merge their work (this effect is reduced in the Bazaar model, since everything is divided into very small sub-tasks carried on by very small groups).

However, we were only two to work on Kinonk. Thus, we were able to keep up with the source code’s evolution without wasting too much time.

Chapter 10

Software

Developers rely on several software in order to write their code, and eventually make it work. This chapter describes the main ones and those we choose.

10.1 The operating system

An operating system is a software controlling the hardware of a computer. Nowadays, Microsoft Windows along with Apple Mac OS share most of the desktop market.

Open-source developers usually like using open-source software themselves, because they are interested in how what they use work and often want to customize their tools according to their needs. Indeed, some programmers even try (mainly for ethical reasons) not to use any proprietary software whenever possible.

As a matter of fact, most open-source coders do not rely on Windows or even Mac OS because the source code of these O.S. is not disclosed to the public. The most well known open-source OS is GNU/Linux. It supports a wide variety of language, developing environments, and often comes with a wide range of open-source tools installed.

Since both of us were already using GNU/Linux at the time, there was nothing to change as far as the operating system was concerned.

We were never disappointed by GNU/Linux. Most, if not all, of the tools we used to develop Kinonk were already part of GNU/Linux and did not need any additional tweaking.

10.2 The text editor

A text editor is used to write all of the source code (this source code is then converted to machine-readable code using a software called a compiler). The most basic editor everyone knows about is Notepad, which is part of Windows.

As a developer, you would like your editor to indent (to ease readability, each code line does not begin at a single column. The deeper in the code, the further you have to indent.) and color (in order to be able to easily distinguish comments from the rest of the code for example) your code for you. Moreover, it would be great if it provided some integration with related tools, like the debugger, so that you can read your code while using your debugger for example.

Text editors offering such features are called “Integrated Development Environment”. There are a lot of IDE available for Linux. One of the oldest one is emacs. It is fully programmable and includes tight integration with

CVS, the gcc compiler and the gdb debugger. We already had had some experience with it, so we decided to use it once more for this project.

10.3 The build system

A build system is the set of tools one use to convert the software from plain text human-readable source files to a binary machine-readable executable. A compiler is used to convert a plain text file to an executable. Still, there are usually many source files and most often they must be compiled in a specific order. To take care of those issues, another software called a build tool has to be used.

As a compiler, we used GCC. This is the one the Linux project uses and overall the most popular one.

A big open-source project usually has tens of source files distributed into several directories. After a modification is done, every file must be compiled again. One cannot afford to run manually the compiler so many times.

Moreover, sometimes file `foo` needs file `bar` to be compiled in order to compile properly, so you also have to compile files in some non-random order.

Those issues are not at all simple to deal with. There are dozens of software created only for that purpose. Through configuration files, the developers inform the program on which files must be built and their dependencies on other files, and the software does the rest for you. The most popular, almost universal build tool on GNU/Linux is called “make”. We rejected using it because its syntax did not seem obvious to us at all. Moreover, it did not seem easy to install on Windows (and we wanted the program to work on Windows as well).

Eventually, we decided for a less well-known tool called “scons”. Its syntax is much less cryptic, as it is that of the Python programming language, in which we already had some background.
Furthermore, it works out of the box on a variety of operating systems so building on Windows was not an issue.

Looking back, scons was a good tool, although we came to realize that its syntax wasn’t as powerful as we thought it was. For example, it was very difficult to program scons so that executing ’scons install’ would only install, not build the files. In fact, we ended up using scons only for the build itself, leaving the installation to a shell script. This has some disadvantages as well, since some operating system like Windows do not support UNIX shell scripts out of the box.

Chapter 11

Communication

An open-source project managed by more than one developer needs constant communication. One programmer might want to criticize the code of another, for example report a bug or a outline missing feature. Moreover, one might have a question regarding how some piece of code works, in order to use it later. Finally, developers sometimes discuss together before implementing significant new features, for example on the design or simply on who will be in charge.

In this chapter, we describe the communication tools we used and contrast them with those of large open-source projects.

11.1 Instant Messenger

Instant messenger (commonly abbreviated IM) was used when quick replies were needed. Indeed, this was rarely the case in our project. Sometimes we were online at the same time and thus could avoid using email for communicating quick information. The only time when we needed IM was when what we called a “meeting” was needed. In a meeting, we would discuss the advancing of the project, and deal with “administrative” material such as deadlines and assignment of tasks.

Large open-source projects use IRC as instant messenger network, because it supports an almost unlimited number of simultaneous connection. We both already had a MSN client and were using it at the time, so MSN was chosen mainly for convenience reasons.

11.2 Email

In our opinion, this is the primary mean of communication of most open-source projects. Ours made no exception.

Emails can be read from any computer connected to the Internet and do not require the communicating programmers to be connected at the same time.

Most open-source projects use public mailing lists, where the emails are published on the web and can be read by anyone in or outside of the project. Anyone can subscribe to a mailing list and receive a copy of the emails being exchanged. One can also freely participate and send an email to this mailing list. Large projects have several mailing lists dealing with different topics, e.g bug reports, random thoughts, question about the code etc...

Our emails were not made public. We did not feel the need to do so, as we had decided to adopt the “Cathedral model”, where we maintain full control on our code between release.

11.3 The Notebook

We kept a notebook, accessible only by both of us, in which we wrote information that did not need any action from any developer.

That included logs of what we did everyday, in order for the partner to keep up with his collaborator's work. Moreover, we could also write our thoughts on the project, for example a brainstorming on some new ideas.

The purpose of the notebook was to give us something on which to fall back in case we got stuck, i.e we could always go through the notebook if we didn't know how the project should continue.

For most larger open-source project, employing often more than ten developers, keeping a notebook would be more of a burden: it would grow very quickly, mixing logs and ideas. Soon, it would get difficult to go through it.

Every time a change is made to the source, developers are expected to write a small message describing their changes. Those messages might then be appended to a `ChangeLog` file, or kept along with the project's revision history (see chapter 12.2).

We also used that feature but we thought it would be clearer to maintain a centralized notebook. Nevertheless, most projects rely on the previously-mentioned feature for keeping logs while ideas are generally exchanged using email.

Chapter 12

Hosting

12.1 The Hosting Provider

An open-source project requires some physical medium where the sources can be stored. In order for anyone to be able to participate, the sources must also be accessible at anytime of the day from any location.

The solution is to host the sources on a computer connected to the Internet. This computer must be on all the time and have access to a large bandwidth, in order to be able to quickly upload data to the Internet. Such a computer is very different from our desktop computers and is called a *server*.

It was of course possible to buy our own server and manage it ourselves, but it would have been costly. However, some companies rent a tiny part of the server they manage for free. In layman's terms, the company provides a limited amount of space and some service (e.g. a bug tracker, a forum...) on a server they manage. Several projects can then share the same server at the same time. Such a company is called a "hosting provider".

There are several hosting providers around. Without entering into details, we chose Sourceforge (`http://www.sourceforge.net`) because it was popular and easy to set up. Other popular hosting providers include GNU Savannah or Berlios.

12.2 The Version Control System

It is not enough to have a place where to store data, there must still be a convenient way to read and write them. In the case of a multi-developers open-source project, we would like anyone to be able to write their one code without any risk of conflict within the rest of the code.

For example, one would like to avoid the following situations.

- Alice accidentally deletes a file Bob was working on.
- Alice changes a file Bob was working on, and Bob has to start all over again.
- Alice finds a bug but the code was changed by Bob in the meantime.

What you would like to do is to go back in time whenever you need it. For that purpose, most open-source project use what is called a versioned filesystem.

In a versioned filesystem, every change made is kept in memory, making it possible for anyone to revert any file back to any point in time. Even deletion is versioned, meaning that the file can still be recovered at any moment.

A versioned filesystem is managed by a special software, called a version control tool. It is through this software that all the programmers read the sources and write their changes.

Most versioned filesystem work by recording only the difference between successive file version. This makes it possible to merge one's work one some version of some file with a higher version of some file.

For example, consider Alice is working on the 1.4 version of a file. Meanwhile, Bob writes his changes. The most recent version of the file is now 1.5. In that case, Alice can still write his changes to the file if Bob hasn't changed the part Alice was working on.

A practical illustration of this is when two groups of developer work on two different projects: one works on the version to be released to the public, and the other works on experimental feature. At some point, both work can be merged as one, with very little hassle.

There are at least two main version control systems projects use. The first one, called CVS, had been around for a long time and was beginning to show its limitation in 2000 when work on the second one, SVN, begun.

While CVS only supports plain text files, SVN can also work with any kind of binary files. Moreover, removing and renaming of files are easier to accomplish and record in SVN than in CVS. Those are only the main improvements SVN offers compared to CVS.

Although SVN is recognized as more powerful, it is also in our opinion harder to use. Furthermore, the limitations of CVS weren't that obvious to us on the first place. A project like ours didn't need all the complicated features of SVN. Finally, our hosting provider made it much easier to use CVS because of the very large documentation available. As a rule of thumb, beginners should always choose the tool which has the largest amount of documentation available, and this is what we did.

Chapter 13

Conclusion

We have covered the main tools and techniques open-source developers are using for their projects. It appears to us that most of them are surprisingly simple.

Thousands of programmers contribute to the Linux kernel project. Still, most of the communication is done by plain-text emails visible to the public.

All the tools you need to successfully develop an open-source project can be downloaded for free. You only need a text editor and a compiler to successfully create a program. The choice of text editor and compiler is huge. A lot of them are themselves open-source.

Once you want to publish your code, several websites offer you some space for free. The versioned filesystem ensures no code will ever be lost.

Most of the tools we used are easy to use, even for non-computer-savvy users. This is true for only one reason: *the amount of documentation available.* Open-source code is so transparent that it can easily be documented. The large number of people able to contribute also is an important factor.

Part IV

Appendixes

Chapter 14

Kinonk's homepage (how to get Kinonk)

The Kinonk project has a homepage: `http://kinonk.sourceforge.net`

There you have an extensive presentation of the program as well as instructions on how to get the program, its sources and so on.

14.1 Released package

From the webpage, anyone can access the project's download page on SourceForge.net.

On that page there is a list of all available packages. You just have to select the one you need according to your platform and the content you want (executable, code, report, scripts etc.) and download it. Then what you need to do is described in the README file.

14.2 Alternative

If you use Concurrent Versions System (CVS) [25], you can download all the latest files running

To access KinonK's repository, you have to go to the directory where you want KinonK to be built. Then, run

```
cvs -d:pserver:anonymous@kinonk.cvs.sourceforge.net:/cvsroot/kinonk checkout -P .
```

This will download everything there is on the repository. If you want just a specific set of files you can replace the period at the end of the command by the relative path of the folder(s) containing what you need.

If, later, you want to update to the latest versions of the files, you do not have to do all this again. The only thing you need is to run the following command.

```
cvs -d:pserver:anonymous@kinonk.cvs.sourceforge.net:/cvsroot/kinonk update -P
```

14.3 Dependencies

In order to handle the sources and compile the program or the report, you need the following:

- CVS in order to be able to check out the files.
- Python as it is the language in which the building utility is written. (`http://www.python.org` for more information)
- SCons, the building utility used for the project (downloadable from `http://www.scons.org`).
- A C++ compiler. We used `g++`, from the Gnu Compiler Collection (GCC).

Chapter 15

The Kinonk manual

Please be aware that the following section has been automatically generated from the kinonk man page, which means that it is best viewed using the UNIX man program... Thus there might be some minor formatting mistakes, that generally are inevitable when we convert from one document format to another.

15.1 Kinonk man page

```
KINONK(6)                                                         KINONK(6)



NAME
       Kinonk - Pentominoes & Katamino player

SYNOPSIS
       Kinonk [ options ...  ]

DESCRIPTION
       Kinonk is a pentominoes playing program.

       Pentominoes  is  a  multi-player board game. The goal is to be the last
       player to put a pentomino (a piece composed of five congruent  squares)
       on the board.  Kinonk can also play Katamino, a variant of Pentominoes.

       For more information on Pentominoes:

       <http://en.wikipedia.org/wiki/Pentomino>
       <http://www.katamino.co.uk>

       Kinonk uses a brute-force search algorithm capable of playing the  game
       perfectly  if  given enough time. It is also possible to force the pro-
       gram to play within a given amount of time.

       Among the parameters that can be customized are the size of  the  board
       and the pentominoes that will be used.

       If  you  think Kinonk is not fast enough, you are given the possibility
       to change some global variables using a plain text configuration  file.

       To  measure  its  performance,  Kinonk can output statistics in a plain
       text file.

       Note that Kinonk is an engine featuring only a  very  basic  text-based
       interface. However, any graphical interface can easily communicate with
       the software.
```

```
OPTIONS
       -S SIZE,--size SIZE
              Set the size of the board to SIZE given as  widthxheight  (where
              width*height  must  be  greater than 4 and smaller than 121).  A
              warning will be printed to stderr if Kinonk thinks the board  is
              too large.  Defaults to 8x8.

       -e PENTO, --exclude PENTO
              Add  a pentomino to be excluded from the game. For example: -e I
              would exclude the I pentomino. By default,  no  pentominoes  are
              excluded.

       -k,--katamino
              Use  the  rules of Katamino, which are a little more restrictive
              than the classical ones.

       -s,--solve
              Find one winning move and exit. This flag implies --notime

       -A,--solveall
              Find all winning moves and exit. This flag implies --notime

       -p,--humans N
              Give the number of humans going to play. Currently, only  values
              of 0 or 1 are supported. Defaults to 1.

       -c,--compturn N
              Set  the place in which the computer will play, 1 is first, 2 is
              second etc...  Currently, only values of 1 and 2 are  supported.
              This  option  only makes sense if the number of human players is
              greater than 0. Defaults to 2.

       -t SECONDS, --time SECONDS
              Set a time limit for each move.  After  the  limit  is  reached,
              Kinonk  will  be forced to play even if it has not finished ana-
              lyzing the position. This option  may  reduce  Kinonk's  playing
              strength. Using --time 0 is equivalent to --notime.  Defaults to
              60.

       -T,--notime
              Do not use any time limit. Be aware that the program may take  a
              lot of time to answer if this option is set.

       -f FILE, --stats FILE
              Make Kinonk output some statistics on the search algorithm effi-
              ciency in FILE.  For more information on  which  statistics  are
              kept, see the STATISTICS section below.

       -a FILE, --algo FILE
              Set  the  customization  FILE  from  where  Kinonk will read the
              global variables controlling the main algorithms. If no file  is
              specified, the defaults will be used.  For more information, see
              the ALGORITHM VARIABLES section below.

       -d DEPTH, --details DEPTH
              Set the maximum absolute DEPTH at which Kinonk outputs  informa-
              tion  on  what  it  is  analyzing. A depth of 0 is the root.  By
              default, no information will be outputted.

       -q, --engine
              Output only errors and other necessary information. Useful  when
              other  programs  (e.g graphical interface) are using Kinonk. See
              also the INTERACTION section below.
       -h, --help
              Display this help and exit.

       -v, --version
              Display the version number and exit.


INTERACTION
       When set up to play against a human, Kinonk will  output  its  move  to
       stdout  and  read stdin to get the human's reply. If --engine is given,
       the output format of Kinonk's moves will be as follow:
       SQUARE1 SQUARE2 SQUARE3 SQUARE4 SQUARE5
       Where each SQUARE is X Y or XxY

       Kinonk expects the input to be  the  same  format  as  its  output  is.
       SQUARE  arguments  can be given in any order, and the program will find
       the move if it exists. If the move does not exist, or the input is  not
       formatted as expected, Kinonk will print an error message and exit with
       non-zero status.

       It is possible to use the following commands in order  to  make  Kinonk
       perform  various actions. The only thing you have to do is when you are
       prompted for input, just enter the command and hit the <RETURN> key.

       p      This command will just make Kinonk ask for five  coordinates  (a
              move).  It produces exactly the same result as the standard move
              input system described above. It is kept only for backward  com-
              patibility reasons.

       c      Makes  the program solve the position and play the best move for
              you.

       r      Plays a random move.

       u      Undoes all moves up to the last one you played (included).

       a      You resign. The game is interrupted and the program  exits  nor-
              mally (return value is 0)

       h      If you forget about these commands while playing, you can always
              use h to display a short description of  all  commands  you  can
              use.

STATISTICS
       If  you run Kinonk with --stats on, the program will, just before exit-
       ing, fill in a file with all sorts of  statistics  formatted  in  JSON.
       Using  this  file,  you will be able to generate various graphs using a
       script in scripts/ . To invoke it, just type
       scripts/parsestats.pl STATS_FILE PATH
       Replacing PATH by where you want the graphs to be created. About 30 png
       images will be generated.
       Here  is  a  description of ALL the data you could ever get. Sometimes,
       according to some parameters, some data is irrelevant and is  therefore
       not outputted.

       All the indexes of the arrays start at 0, else the data they hold would
       not make sense (or could be wrongly interpreted)

       All the data is represented as follow:

       Name_of_the_object
              Type of the object Description of the object


       statsfile
              string The name of the file the stats have been written to.
```

game_info
: structure Contains the parameters Kinonk has been run with so the variations in the data can be associated with the different parameters.

return_stat
: integer The return status of the program.

board
: array of integers The length and height of the board in squares.

katamino
: boolean Shows if the Katamino Rules have been used or not.

excluded
: array of booleans The boolean indicates if the ith pentomino is excluded from the game. i is the index at which the value is.

time_limit
: integer The time, in seconds, the computer has been given to solve a position.

action
: string one_win means that only one winning move has been searched and found. all_win means that all winning moves have been searched and found. comp_human is a duel between a computer (player 1) and a human (player 2). human_comp is a duel between a human (player 1) and a computer (player 2). comp_comp means that the computer played against itself.

killeristics
: integer Number of killer moves kept during the game.

method
: array of strings The method used to sort the list of moves at the ith depth i is the index at which the value is.

winsratio_nmoves
: array of integers The number of moves tested at each relative depth when partially solving the game.

iwr_nrand
: integer The number of moves the program has to play randomly before solving the position.

iwr_ngames
: integer The number of games Kinonk has to play for each candidate move.

total_time
: integer The time, in seconds, the computer needed to complete the task.

game_info
: end of structure All the data described below is not part of game_info anymore.

analyzed_before_cutoff
: structure of arrays of integers Twelve arrays called depth_ a where a represents the depth. A value is the number of times i moves have been analyzed before finding a cutoff in the tree of moves. i is the index at which the value is found.

number_candidates
: structure of arrays of integers Twelve arrays called depth_ a where a represents the depth. A value represents the number of times a move has i candidates. i is the index at which the value is found.

pento_cutoff
: structure of arrays of integers Twelve arrays called depth_ a where a represents the depth. A value represents the number of times a pento of id i has lead to a cut off in the tree of moves. i is the index at which the value is found.

average_ncand
: array of integers The average number of candidates the program had to solve at depth i i is the index at which the value is found.

analyzed_pos
: array of integers The total number of positions solved at the ith depth. i is the index at which the value is found.

max_pos
: array of integers The maximum number of positions the program could have had to solve at the depth. i is the index at which the value is found.

max_leaf_pos
: integer The maximum number of positions the program would have had analyzed if it had to do a full brute-force search.

total_analyzed_pos
: integer The total number of positions that have been analyzed by Kinonk.

min_leaf_pos
: integer The minimum number of positions the program could have solved in order to get the same results. Put otherwise, the number of positions that have to be solved in order to make a proof by exhaustion of the known result.

time_pos_sec
: integer The average number of moves the computer managed to solve each second.

ALGORITHM VARIABLES

If you are not satisfied with Kinonk's playing strength but are not willing to alter the code, you are given the possibility to customize some variables used by the program's search algorithm.

Even though the default are acceptable, they may not be optimal for all kinds of situations and that is why we leave this opportunity. Be aware though that if you want an overall performance enhancement you have no alternative but to dive into the source code. Also, Kinonk now uses special functions if the defaults sorting methods are going to be used, and that makes it slightly faster in those cases. A message such as "an optimized version of the solving function will be used" will be displayed when Kinonk starts if this is the case.

For a deeper insight on how the program works, you may also want to read the report.pdf or main.pdf file installed in your documentation directory and distributed with Kinonk.

The configuration file is a plain text UNIX configuration file that can be edited with any text editor like emacs, vi, pico, ed or notepad.

Blank lines and every line beginning with a '#' are ignored. The general format is KEY=value. Where the KEY is one of the following:

USE_KILLERS
: If true, Kinonk will use the killer heuristic (

```
       <http://en.wikipedia.org/wiki/Killer_heuristic> ) in conjunction
       with  the  usual  sorting  methods.  A value of 0 means false, a
       value of 1 means true. Defaults to 1.

N_KILLERS
       Give the number of killers that are tried  before  other  moves.
       The default is 2. Only makes sense if USE_KILLERS is true.

DEPTH_METHOD = depth0 depth1 depth2 ... depth11
       The method used to sort the moves at each depth, where depthn is
       an integer between 0 and 5. Each integer corresponds to a method
       as follows:

       0: none
       The  program  just takes the move as they are generated in Posi-
       tionData's  constructor  (see  Doxygen  documentation  for  more
       information or look at the source code)

       1: random
       The moves are sorted in random order.

       2: number of replies (ascending)
       Kinonk  gets  the  number  of replies to each move, and sort the
       moves from the lowest number of replies to the greatest.

       3: wins ratio
       The program plays each candidate, then tries only a small number
       of moves at each level and divide the number of wins by the num-
       ber of games. The candidates are sorted from the greatest  ratio
       to the lowest.

       4: improved wins ratio
       The  program  plays  each  candidate,  then plays n random moves
       before solving the position completely. The process is  repeated
       m  times. The ratio of the number of wins by the number of games
       is calculated. The candidates are sorted from the greatest ratio
       to the lowest.

       5: number of replies (descending)
       Kinonk gets the number of replies (2) to each move, and sort the
       moves from the greatest number of  replies  to  the  lowest.  It
       might  be  useful to swap this setting with the ascending number
       of replies (see the documentation for more information).

       Note that the methods numbering  has  no  special  meaning.   We
       advise  you not to use the wins ratio or improved wins ratio for
       anything else than the very first depth for large (> 7x7)  board
       size.  Consider  trying  the  improved wins ratio instead of the
       wins ratio.

       By default, the number of replies is used for each depth.

WINSRATIO_MAX
       The wins ratio will be between 0 and WINSRATIO_MAX. This is  not
       very  useful  unless  you really want more precision. Default to
       1000.

WINSRATIO_SORTING_METHOD
       The method used to sort the moves before selecting some of  them
       when  using the wins ratio method. The value must be an integer,
       see the DEPTH_METHOD documentation to  know  what  the  possible
       values are. Default to 1 (random).

WINSRATIO_NMOVES = depth0 depth1 depth2 ... depth11
       The  number  of  moves  to be tested at each relative depth when
       using the wins ratio method.
```

```
              The default is 10 5 5 5 2 2 2 2 2 2 2 2 2

       IMPROVED_WINSRATIO_NRANDOM
              If the method used is the improved wins  ratio,  the  number  of
              moves  the  program  must play randomly before solving the posi-
              tion. Default to 2.

       IMPROVED_WINSRATIO_NGAMES
              If the method used is the improved wins  ratio,  the  number  of
              games the program will play for each candidate. Default to 5.


UNRESTRICTIONS
       This program is free software distributed under an open-source license.
       You can redistribute it and/or modify it under the  terms  of  the  GNU
       General  Public  License  as published by the Free Software Foundation;
       either version 2 of the License, or (at your option) any later version.

       This  program  is  distributed  in the hope that it will be useful, but
       WITHOUT ANY  WARRANTY;  without  even  the  implied  warranty  of  MER-
       CHANTABILITY  or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General
       Public License for more details.

       You should have received a copy of the GNU General Public License along
       with this program; if not, write to the Free Software Foundation, Inc.,
       51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA.


AUTHOR
       Yann Schoenenberger <waiess at users.sourceforge.net>
       Sebastien Vasey <svasey at users.sourceforge.net>


COPYING
       Copyright (c) 2007 Sebastien Vasey
       Copyright (c) 2007 Yann Schoenberger

       Copying and distribution of this file, with  or  without  modification,
       are  permitted  in  any  medium  without royalty provided the copyright
       notice and this notice are preserved.



                                  User Manual                        KINONK(6)
```

Bibliography

[1] Stroustrup Bjarne. *Le Language C++*. Pearson Education, 2003. ISBN: 2-7440-7003-3 , in French.

[2] Overland Brian. *C++ Without Fear*. Prentice Hall, 2005. ISBN: 0-321-24695-0.

[3] Adams Douglas N. *The Hitchhiker's Guide to the Galaxy*. Megadodo Publications. Useful and mostly harmless.

[4] Raymond Eric S. *The Cathedral and The Bazaar*. O'Reilly, 2001. Can also be accessed at:
`http://catb.org/~esr/writings/cathedral-bazaar/`.

[5] Lessenger Ernest W. Pentominoes (a software solving the pentomino puzzle), 2004. [Online; accessed 13-September-2007]
`http://www.lessenger.net/ernest/projects/pent/`.

[6] Free Software Foundation. Fsf - the free software foundation, 2007. [Online; accessed 13-September-2007]
`http://www.fsf.org/`.

[7] Free Software Foundation. Gnu free documentation license - gnu project, 2007. [Online; accessed 13-September-2007]
`http://www.gnu.org/licenses/fdl.html`.

[8] Free Software Foundation. Gnu general public license - gnu project, 2007. [Online; accessed 13-September-2007]
`http://www.gnu.org/copyleft/gpl.html`.

[9] Nivasch Gabriel. Solving Pentomino Puzzles with Backtracking. [Online; accessed 13-September-2007]
`http://yucs.org/~gnivasch/pentomino/`.

[10] DJ Games. Katamino :: The fantastic educational shapes puzzle, 2007. [Online; accessed 13-September-2007]
`http://www.katamino.co.uk`.

[11] Orman Hilarie. *Games of No Chance*, chapter Pentominoes: A First Player Win, pages 339–344. MSRI Publications, 1996. Can also be accessed at:
`www.msri.org/publications/books/Book29/files/orman.pdf`.

[12] Opensource Initiative. Open source initiative, 2007. [Online; accessed 13-September-2007]
`http://www.opensource.org/`.

[13] Smoot Michael E. Templatized C++ Command Line Parser Library. [Online; accessed 13-September-2007]
`http://tclap.sourceforge.net/`.

[14] Wagner Richard J. Configuration file reader for c++, 2007. [Online; accessed 13-September-2007]
`http://www-personal.umich.edu/~wagnerr/ConfigFile.html`.

[15] SGI. Standard Template Library Programmer's Guide, 1994. [Online; accessed 13-September-2007]
`http://www.sgi.com/tech/stl/index.html`.

[16] Chesnutt Stan. Pentominoes in Java, using Knuth's Dancing Links Method (puzzle solver). [Online; accessed 13-September-2007]
`http://www.bluechromis.com/stan/pentos_dancing_links.html`.

[17] Knight Steven. *Scons User Guide*, 2007. [Online; accessed 13-September-2007]
`http://www.scons.org/doc/production/HTML/scons-user.html`.

[18] Wikipedia. Negamax —Wikipedia, The Free Encyclopedia, 2006. [Online; accessed 13-September-2007]
`http://en.wikipedia.org/w/index.php?title=Negamax&oldid=70389434`.

[19] Wikipedia. Algorithm —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007]
`http://en.wikipedia.org/w/index.php?title=Algorithm&oldid=156613112`.

[20] Wikipedia. Alpha-beta pruning —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Alpha-beta_pruning&oldid=152680188.

[21] Wikipedia. Best-first search —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Best-first_search&oldid=125508188.

[22] Wikipedia. Brute-force search —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Brute-force_search&oldid=156237457.

[23] Wikipedia. Case analysis —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Case_analysis.

[24] Wikipedia. Computer-assisted proof —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Computer-assisted_proof.

[25] Wikipedia. Concurrent versions system —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Concurrent_Versions_System&oldid=157343213.

[26] Wikipedia. Game complexity —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Game_complexity.

[27] Wikipedia. Iterative deepening depth-first search —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Iterative_deepening_depth-first_search&oldid=154701164.

[28] Wikipedia. Killer heuristic —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Killer_heuristic&oldid=110313714.

[29] Wikipedia. Minimax —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Minimax&oldid=153705705.

[30] Wikipedia. Opensource —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Opensource.

[31] Wikipedia. Pentomino —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Pentomino&oldid=153939310.

[32] Wikipedia. Polyomino —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Polyomino.

[33] Wikipedia. Proof by exhaustion —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/wiki/Proof_by_exhaustion.

[34] Wikipedia. Solved game —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Concurrent_Versions_System&oldid=157343213.

[35] Wikipedia. Standard template library —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Standard_Template_Library&oldid=155173607.

[36] Wikipedia. Transposition table —Wikipedia, The Free Encyclopedia, 2007. [Online; accessed 13-September-2007] http://en.wikipedia.org/w/index.php?title=Transposition_table&oldid=155805544.