Bachelor Thesis

Ultra-strongly Solving the Board Game Quarto and an Adaptation of It

Author
Christian Rahstorfer

Matriculation Number
12005184

Institute
Institute of Algorithms and Theory

Supervisor
Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin Aichholzer

Courses
700.200 24W 1SSt SE Verfassen wissenschaftlicher Arbeiten
700.101 24W 2SSt SP Bachelorarbeit Informatik
700.104 24W 1SSt PT Bachelorprojekt

Study Programme
521 Informatik

Date
March 2025

Abstract

The goal of this bachelor thesis is to ultra-strongly solve the game Quarto. Additionally an adaptation of this game, which we will refer to as simple Quarto, is also ultra strongly solved. What does it mean to solve a game ultra-strongly? It is defined as knowing, for any given legal position, for all possible moves, what moves lead to a win, draw or lose and in how many half-moves this happens. So after implementing the program, it will be able to play perfectly, that is, making the moves that lead to a win in the shortest number of moves, a draw or a loose in the longest possible number of moves, depending on the initial position. By definition, it is not possible to play any better than this, since all possible continuations were already considered beforehand in a complete game-graph, saving positions to a certain depth.

Contents

1 Introduction - explaining the rules **3**
- 1.1 What is Quarto? . . . 3
- 1.2 What is simple Quarto? . . . 3

2 Motivation for compact game states **3**
- 2.1 Game state representation . . . 4
- 2.2 Naive approach of representing game states . . . 4

3 Basic principles of encoding a game state **5**
- 3.1 Transforming board positions to non-equivalent states . . . 5
  - 3.1.1 Possible transformations of the game field . . . 6
  - 3.1.2 Possible transformations of the pieces on the game field 8
- 3.2 Encoding methods to achieve an efficient encoding . . . 9
  - 3.2.1 Encoding of bit-sorted pieces . . . 9
  - 3.2.2 Encoding permutations . . . 9
  - 3.2.3 Encoding two positions . . . 11
  - 3.2.4 Encoding Combinations . . . 12
  - 3.2.5 Encoding the position of one empty field and the 1111 (15) piece . . . 12
  - 3.2.6 Encoding $z + 2$ positions . . . 14
  - 3.2.7 Combine encodings . . . 16

4 Compact game state representation **16**
- 4.1 Calculating upper bound for the number of game states . . . 17
- 4.2 Encoding different game states . . . 17
  - 4.2.1 Encoding for 16 pieces on the board . . . 18
  - 4.2.2 Encoding for 15 pieces on the board . . . 19
  - 4.2.3 Encoding for 14 to 5 pieces on the board . . . 20
  - 4.2.4 Encoding for 4 to 2 pieces on the board . . . 21
  - 4.2.5 Encoding for 1 piece on board . . . 22
- 4.3 Possible improvements to the encodings . . . 23

5 Practical steps to ultra-strongly solving the games **23**
- 5.1 The rating algorithm . . . 24
- 5.2 How does rating one state work for simple Quarto? . . . 24
- 5.3 How does rating one state work for Quarto? . . . 25
- 5.4 Memory requirements for saving the ratings . . . 26

6 Playing against the machine on a website **27**
6.1 How does the machine make a move? 27

7 Acknowledgments **28**

1 Introduction - explaining the rules

1.1 What is Quarto?

Quarto is an imperial game, which means that all pieces can be played by and belong to both players. It consists of a 4×4 board and there are 16 pieces to choose from. Each piece has four properties: big/small, light/dark, square/round and with/without a hole. All 16 pieces are unique, differentiating each other in at least one of the four properties from the other pieces. One move consists of placing one piece on the board and giving the other player the piece they have to place. At the beginning the second player starts by giving the first player the piece with which they start the game. A player won, if they place a piece that creates at least one row, column or diagonal with 4 pieces having at least one feature in common. If all pieces were placed and no player won then the game is a draw.

1.2 What is simple Quarto?

Simple Quarto is a made up variant of Quarto. It is similar to Quarto, the only difference to Quarto is that a move does not consist of placing a piece that your opponent gave you and then choosing a piece for your opponent, but rather you choose a piece for yourself and make a move with that piece. So the game begins by the first player choosing a piece for themselves and placing it on the board. Then it's the second players turn.

2 Motivation for compact game states

A crucial part of this thesis is to efficiently represent the different game states, so to not count states as different if they are actually the same.
Why is this important? Since the goal is to store all the ratings, that is, the number of half moves to a win, a loose or a draw for all game states, it is important to keep the number of game states as low as possible.

2.1 Game state representation

The game state of Quarto consists of a 4×4 board and the pieces that are placed on it. Each piece has four properties. Those can be mapped to a binary digit with length four, so a decimal number between 0 and 15. From left to right this representation is defined as:

- light = 1 and dark = 0
- big = 1 and small = 0
- square = 1 and round = 0
- with = 1 and without = 0 a hole

For instance 7, which equals 0111 in binary, would represent the piece that is: dark, big, square and has a hole.

A full Quarto board, with all stones placed, can be represented as shown in Table 1. When referring to a specific piece we will refer to as the e.g. 1111 (15) piece. This is done for readability.

6	11	12	4
0	7	3	15
2	13	10	8
9	14	1	5

Table 1: Full Quarto board, example

2.2 Naive approach of representing game states

In an compact encoding each game state is mapped to a distinct number from $[0, \ldots, n)$, where n is the number of game states. So the question is how many bits would be needed, to be able to represent n and therefore also any number beneath it? The smaller n is the better of course. There are different methods. The first idea that would come to mind is to have 5 bits for each field on the 4×4 board: 1 bit to represent weather the field is occupied or not and 4 bits to know which piece is on the field. With 16 fields that would be a total of 80 bits for each game state. To put this into perspective, if all you need is one byte to store the rating of a state, you would need 2^{40} TB to store the ratings of all possible states.
This is of course not feasible. On the other hand, not all configurations of this encoding would be needed to save all legal game states of the game. For

example a field full of the piece 16 could also be encoded. So a more realistic approach on how much memory would be needed with an naive encoding is:

$$(\sum_{i=1}^{16} \frac{16!}{(16-i)!} * \binom{16}{i}) \approx 2^{12.5} \text{ TB}$$

This encoding leverages the fact that each piece is unique. There are $\frac{16!}{(16-i)!}$ possibilities to place i pieces on the board. We want to be able to pick any subset of i pieces from the set of 16 pieces and place them on the board. That is why we multiply with $\binom{16}{i}$, which represents all possible subsets of i pieces. This approach enables us to encode any board configuration with i pieces. Summing over i from 1 to 16 then yields the total memory needed to store one byte for each state. However, this encoding is still not optimal. In Section 3, we discuss several improvements to this approach.

3 Basic principles of encoding a game state

How can an encoding be more efficient?
There are two principles at play here. First, the encoding should only be able to represent game states that can be reached in the game. If a game state is not reachable it will never be played therefore there is no need that the encoding can represent it.
Second, the encoding should represent equivalent board positions as the same number. Equivalent board positions or states are positions that look different but are actually the same. For example it does not matter from which direction someone looks at the board. Boards that are turned by 90 degrees should be treated as the same position. Equivalent states also have equivalent possible moves and thus equivalent follow-up positions. With this principle a lot of optimizations are possible.
Furthermore, it should be a maximally-dense encoding, meaning all possible game states, which are represented as numbers from 0 to $2^b - 1$, where b is the number of bits needed to represent one game state as a number, should be able to be mapped to game states that could occur.

3.1 Transforming board positions to non-equivalent states

In this subsection we show different methods to reduce all possible boards to only non-equivalent states.

3.1.1 Possible transformations of the game field

The indices i, j and k of the ids $a_{i,j,k}$ in Table 2 correspond to the initial row, column, and diagonal of each field, respectively. Specifically, i denotes to the current row, j the current column and k the current diagonal of the field.

$a_{0,0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3,1}$
$a_{1,0}$	$a_{1,1,0}$	$a_{1,2,1}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1,1}$	$a_{2,2,0}$	$a_{2,3}$
$a_{3,0,1}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3,0}$

Table 2: The game field

In Table 2, all fields have distinct ids. In Table 3 the fields are changed in a way, that every field still is with the same fields in a row, column and diagonal, as before. This is possible due to the symmetric nature of the game.

$a_{0,3,1}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3,0}$
$a_{0,2}$	$a_{1,2,1}$	$a_{2,2,0}$	$a_{3,2}$
$a_{0,1}$	$a_{1,1,0}$	$a_{2,1,1}$	$a_{3,1}$
$a_{0,0,0}$	$a_{1,0}$	$a_{2,0}$	$a_{3,0,1}$

(a) 90° counterclockwise rotation

$a_{0,3,1}$	$a_{0,2}$	$a_{0,1}$	$a_{0,0,0}$
$a_{1,3}$	$a_{1,2,1}$	$a_{1,1,0}$	$a_{1,0}$
$a_{2,3}$	$a_{2,2,0}$	$a_{2,1,1}$	$a_{2,0}$
$a_{3,3,0}$	$a_{3,2}$	$a_{3,1}$	$a_{3,0,1}$

(b) Horizontal reflection

$a_{1,1,0}$	$a_{1,0}$	$a_{1,3}$	$a_{1,2,1}$
$a_{0,1}$	$a_{0,0,0}$	$a_{0,3,1}$	$a_{0,2}$
$a_{3,1}$	$a_{3,0,1}$	$a_{3,3,0}$	$a_{3,2}$
$a_{2,1,1}$	$a_{2,0}$	$a_{2,3}$	$a_{2,2,0}$

(c) Inversion

Table 3: Possible transformations for the game field

With these three transformations, given in Table 3, other possible transformations, that result in an equivalent state, can be performed. The identity is also an transformation and just means to perform no transformation. Abbreviations for transformations are given as follows:

4 Basic Operations

- the identity ... I

- 90° counterclockwise rotation ... 90° CCR
- horizontal reflection ... HR
- inversion ... INV

Composable Operations

- 90° clockwise rotation ... 90° CR
- 180° counterclockwise/clockwise rotation ... 180° CCR/CR
- vertical reflection ... VR
- left diagonal reflection ... LDR
- right diagonal reflection ... RDR
- transformation ... T

There are 16 distinct transformations that can be done. These transformations are composed of the three transformations in Table 3 in the following way:

1. I = I
2. 90° CCR = 90° CCR
3. HR = HR
4. INV = INV
5. 180° CCR/CR = 2 * 90° CCR
6. 90° CR = 3 * 90° CCR
7. RDR = HR + 90° CCR
8. VR = HR + 180° CCR
9. LDR = HR + 90° CR
10. T0 = HR + INV
11. T1 = INV + 90° CCR

12. T2 = INV + 180° CCR
13. T3 = INV + 90° CR
14. T4 = HR + INV + 90° CCR
15. T5 = HR + INV + 180° CCR
16. T6 = HR + INV + 90° CR

3.1.2 Possible transformations of the pieces on the game field

As mentioned in Subsection 2.1, the pieces on the game field have four properties, each property is either 0 or 1. An central observation is, that it does not matter in which order they are represented. Nor is it important if the one state of the property is mapped to 1 and the other state is mapped to 0, as this mapping is chosen arbitrary. It only matters that all pieces have the same notation. For example if for one piece the least significant bit represents that the piece is dark if it is 1 and light otherwise, then this notation holds for all pieces.

The example from Table 4 applies this by XORing all pieces on the board with the first piece, including the first piece, which results in a new, but equivalent game state, where the first piece on the board is always 0. This method can be applied when the board is full, self evidently, but it can also be applied for encoding earlier parts of the game. It is useful, because that way the first field does not need to be considered in the encoding, since it is always 0 and at the same position for all boards when performing this transformation.

0110	1011	1100	0100
0000	0111	0011	1111
0010	1101	1010	1000
1001	1110	0001	0101

(a) Quarto board in binary

0000	1101	1010	0010
0110	0001	0101	1001
0100	1011	1100	1110
1111	1000	0111	0011

(b) Quarto board after XOR

Table 4: XOR with the code of the first piece on the board

The example from Table 5 further exploits the insights from before. The bits for the first three pieces after the zero piece are sorted in a way, that the 1s are the least significant bits possible, while the 0s are on the remaining bits. That way there are fewer possibilities which numbers the three pieces

can have. The bits that are changed for one piece are also changed for all other pieces on the board, in order to keep consistency. The constrained that the pieces that where already ordered can not be unordered again, has to be considered.

0000	1101	1010	0010
0110	0001	0101	1001
0100	1011	1100	1110
1111	1000	0111	0011

(a) Quarto board after XOR

0000	0111	1001	1000
1100	0010	0110	0011
0100	1011	0101	1101
1111	0001	1110	1010

(b) after sorting bits

Table 5: Sorting bits of three pieces

In Table 4, the fact that the mapping of properties is chosen arbitrarily is exploited. Similarly, in Table 5, the arbitrary ordering of properties is utilized. These methods significantly enhance the encoding of game states. In Table 5 only the first three pieces are bit-ordered.

3.2 Encoding methods to achieve an efficient encoding

3.2.1 Encoding of bit-sorted pieces

The pieces are bit-sorted as seen in Subsection 3.1.2, and have to be encoded to a number. It is done by mapping each possible state of the pieces that where bit-ordered to a number between 0 and $k-1$, where k is the number of possible states, so to make the encoding maximally dense. Since the bit-ordering is only done for maximally 3 pieces, the number of possibilities that these pieces can have are manageable, so it can be mapped directly with the use of a table, that stores each mapping. Thus there is no need for an algorithm to encode and decode the bit-ordered pieces. In a nutshell, it is just a lookup-table to know which state is mapped to which number and likewise. In Table 6 you can see for 1 to 3 bit-sorted pieces and the number of k possible states. The table only includes bit-ordering pieces in the interval $[1, \ldots, 14]$, because in the final encoding the 0000 (0) and 1111 (15) piece are not bit-ordered. In Table 6 the column "Number of states" was calculated using a python program.

3.2.2 Encoding permutations

For the remaining pieces, that are not part of the bit-ordered pieces, there are too many possibilities to just map each state to a number and save

Number of bit-sorted pieces	Number of states
1	3
2	16
3	121

Table 6: Number of states for bit-ordered pieces that are in the interval $[1, \ldots, 14]$

them in a lookup table. Therefore, an algorithm is needed to encode a permutation to a number and back. The algorithm that is used needs as input a permutation of size k of $\{0, 1, 2, ..., k-1\}$[1]. In order to use this algorithm, the remaining pieces need to be transformed. For each remaining piece, perform the following steps:

1. Sorting: Sort the piece together with the bit-ordered pieces and the 0000 (0) piece.
2. Index Adjustment: Subtract the piece's index in the sorted order from its original value.
3. Writing: Write the resulting value at the position on the board where the original piece was located.

This process transforms each remaining piece into its new value. As can be seen in Table 7, the remaining pieces are then a permutation of size $k = 12$ of $\{0, 1, 2, \ldots, k-1\}$.

0	7	9	8
12	2	6	3
4	11	5	13
15	1	14	10

(a) after sorting bits

0	7	9	8
8	1	5	2
3	7	4	9
11	0	10	6

(b) preparing remaining pieces

Table 7: Transforming remaining pieces for algorithm

This process can also be done with the pieces that are not on the board, not only for the remaining pieces that are on the board. After performing this process the algorithm can be used, which maps this permutation to a number between 0 and $k! - 1$. For decoding, the encoded number first has to be decoded to the permutation. Afterwards with the help of all the pieces that were used to transform the remaining pieces the decoded permutation

can be reversed to its original form. This is done by inverting the process described before.

3.2.3 Encoding two positions

This section is about encoding two positions of empty fields or non-empty fields. It turns out there are exactly $k = 14$ possibilities of non-equivalent positions as seen in Table 9. This is possible due to the utilization of the transformations in Subsection 3.1.1. Taking the example of 14 pieces on the board and 2 empty fields, there are only $k = 14$ positions for these 2 empty fields after the transformations are done. There are two steps, as can be seen in Table 8, the first one is to transform the matrix with the transformations from Subsection 3.1.1. There are different transformations applied depending on the position of one empty field. After the transformations are applied the empty field is on the last or second last position of the board. The second step is, that the board is transformed so the second field is at one of the non-equivalent positions given in Table 9. Doing these steps results in the encoding for the positions of the two fields which encodes them to a number in the interval $[0, \ldots, k)$. Where there is a 1:1 mapping done between the possible positions and the k possibilities. In Table 9, it can be seen what all the possibilities are for non-equivalent positions of the two fields. "A" marks the positions of the first field and "B" marks the possible positions of the second field. One thing to note is that this process of encoding the position of two fields and transforming therefore the game-field has to be done before the XORing. Because after XORing, the game-field is fixed and no more transformations are done that change the position of the pieces.

8	6	2	9
7	13	-1	0
12	1	15	11
4	14	3	-1

(a) after first step

8	7	12	4
6	13	1	14
2	-1	15	3
9	0	11	-1

(b) after second step

Table 8: Transforming the position of the two empty fields

There is one thing to note at Table 9 at board (b) where the field is at A_2. Here the third position (right of B_{10}) is not part of the second non-equivalent positions. This is the case because when applying inversion and a 90 degree clockwise rotation (see Subsection 3.1.1) the A_2 and the third position would be equivalent to the B_{14} and A_2 position respectively.

B_1			
B_2	B_3		
B_4	B_5	B_6	
B_7	B_8	B_9	A_1

(a) First possibility for A

	B_{10}		
B_{11}			
B_{12}			B_{13}
	B_{14}	A_2	

(b) Second possibility for A

Table 9: Distinct positions of two fields

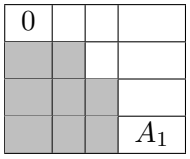
3.2.4 Encoding Combinations

The missing pieces are combinations that are encoded. This makes sense since the missing pieces, so the pieces that are not on the board, don't need to be ordered, so they can be encoded as a combination of numbers. The pieces that are already encoded, for example the bit-sorted pieces, are excluded as done for the permutations in Subsection 3.2.2. So having k missing pieces that are in the range $[0, \ldots, n)$ there are $\binom{n}{k}$ different combinations for those k numbers. So there can be a 1:1 mapping made between these different combinations and the range $[0, \ldots, \binom{n}{k})$. The algorithm that is cited and implemented, describes exactly this kind of mapping [2].
Another example where this algorithm can be used, is to encode the positions of the empty fields, or the positions of the non-empty fields.

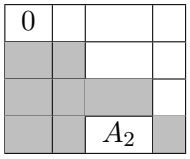
3.2.5 Encoding the position of one empty field and the 1111 (15) piece

Assuming the 1111 (15) piece is on the board and there is only one empty field on the board. We want to encode the position of the empty field and the position of the 1111 (15) piece. With the help of the transformations in Subsection 3.1.1 it is possible to reduce the number of possibilities for the two positions to $p = 25$. This is done by first aligning the empty field to the last or second last position. Then if the 1111 (15) piece is on the right upper half of the board a left diagonal flip is made of all the fields, in order to bring the 1111 (15) piece to the gray fields as seen in Table 10. This results in an encoding of the 1111 (15) piece and the empty field to the range $[0, \ldots, p)$. The 0000 (0) piece is always on the first field, so this field does not have to be considered. This is the case because this encoding is only used if 15 pieces are on the board. In this case after XORing the 0000 (0) piece is always at the first position of the board. This is further explained in Subsection 4.2.2.

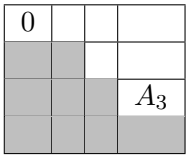
There can not be as much optimization be done as for encoding two



(a) First possibility for empty field A_1



(b) Second possibility for for empty field A_2



(c) Third possibility for empty field A_3

Table 10: Distinct positions of the 1111 (15) piece and empty field

positions of empty or non-empty pieces, see Subsection 3.2.3. This is the case because a field with the 1111 (15) piece is different from a field with no piece on it. That is also why the cases A_2 and A_3 in Table 10 are non-equivalent. Because in A_2, A_3 there is always the empty field. For example if in Table 10 in (b) the 1111 (15) piece is on one of the white pieces after aligning the empty field to A_2. Then a left diagonal reflection (see Subsection 3.1.1) is done. This results in table (c) where the empty field is at A_3 and the 1111 (15) piece is in one of the gray fields.

3.2.6 Encoding $z + 2$ positions

For encoding only two positions of the empty fields or the non-empty fields the Subsection 3.2.3 explains how to do that. But if there are $2 + z$ positions that need to be encoded, then in this case we can use a combination of encoding two positions (Subsection 3.2.3) and encoding combinations (Subsection 3.2.4) as follows.

Two positions	possibilities for remaining positions
B_1, A_1	$\binom{14}{z}$
B_2, A_1	$\binom{13}{z}$
B_3, A_1	$\binom{11}{z}$
B_4, A_1	$\binom{10}{z}$
B_5, A_1	$\binom{8}{z}$
B_6, A_1	$\binom{6}{z}$
B_7, A_1	$\binom{5}{z}$
B_8, A_1	$\binom{3}{z}$
B_9, A_1	$\binom{1}{z}$
B_{10}, A_2	$\binom{6}{z}$
B_{11}, A_2	$\binom{5}{z}$
B_{12}, A_2	$\binom{4}{z}$
B_{13}, A_2	$\binom{3}{z}$
B_{14}, A_2	$\binom{2}{z}$

Table 11: Possibilities to encode positions

In Table 12, it can be seen how many possibilities the remaining z positions for certain 2 aligned position. B_x and A_x refer to Table 9. The sum of all the possibilities are the total possible encodings for $2 + z$ positions. This can be mapped to $[0, \ldots, s)$, where s is the sum of all the possibilities for $2 + z$ positions.
Why do the possibilities for the different options make sense? Because they

are excluding each other in a way, so e.g. if any two of the $2+z$ positions of a game-field can not be mapped to B_1, A_1, but to B_2, A_1, then the position B_1 can be discarded as a position of an empty field or a non-empty field. As a reminder, when encoding positions, there are either the positions of the empty fields or the positions of the non-empty fields encoded, since one can be inferred from the other. That's why then for B_2, A_1, there are only 13 fields left where the remaining positions can be. For B_3, A_1 it's the same, but also the reflection of B_2 can be discarded as an option. This pattern is continued until B_9. Then if all A_1 options are not possible to be mapped to the used positions for the encoding of a given game-field, then there are 6 possibilities for the remaining fields in the case of B_{10}, A_2. In the same fashion, as before, the other options of different two positions are calculated.

P_0	P_2	P_5	P_{10}
P_1	P_3	P_7	P_{12}
P_4	P_6	P_8	P_{14}
P_9	P_{11}	P_{13}	A_1

(a) First possibility for A

	P_{15}	P_{20}	
P_{16}			P_{21}
P_{17}			P_{18}
	P_{19}	A_2	

(b) Second possibility for A

Table 12: The order in which the positions are checked given the first aligned field

In Table 12 we can see the order in which the second aligned position and then the remaining positions are searched for. The algorithm starts the search from P_0 to P_{14} if the first aligned field is A_1. For A_2 the search for the second field and remaining fields starts from P_{15} to P_{21}.
In order to also fit the two aligned fields in the encoding the sum is taken for all the two positions that came before the current two positions. This is then added to the encoded of the remaining positions. As an example, if the two aligned positions are B_4 and A_1, the sum is $\binom{14}{z} + \binom{13}{z} + \binom{11}{z}$. This sum is then added to the encoded of the remaining positions. The remaining positions are calculated by encoding the indexes of them with the encoding combinations algorithm described in Subsection 3.2.4. In this example if $z = 2$ and the two remaining positions are at P_5, P_8, the indices that would be encoded are $0, 3$, since the indexes are counted in the order of the P_x.

This results in a dense encoding which is in the range: $[0, \ldots, s)$.

$$s = \binom{14}{z} + \binom{13}{z} + \binom{11}{z} + \binom{10}{z} + \binom{8}{z} + \binom{6}{z} + \binom{5}{z} + \binom{3}{z} + \binom{1}{z} + \binom{6}{z} + \binom{5}{z} + \binom{4}{z} + \binom{3}{z} + \binom{2}{z}$$

3.2.7 Combine encodings

To encode the hole game-board there are a few different encodings needed. All these encodings give a number as output. So there are a few different numbers that represent the encoded game-field. These numbers should be combined into one number compactly.
Assume we have three encodings, where a, b, c are their number of different possibilities. Let a^*, b^*, c^* be the encoded values, which are strictly smaller than a, b, c respectively. The single encoding e that is the result of combining the three encoded values into a single value, is done in the following way:

$$e = a^*(bc) + b^*(c) + c^*$$

and decoded into the three values again by

$$\begin{aligned} e \text{ DIV } (bc) &= a^* \\ e \mod (bc) &= e_1 \\ e_1 \text{ DIV } (c) &= b^* \\ e_1 \mod (c) &= e_2 \\ e_2 &= c^* \end{aligned}$$

The division (DIV) here only outputs integers, for this to work. As can be seen this can be done with arbitrary many encodings. e is in the range $[0, \ldots, abc)$, which is maximally compact.

4 Compact game state representation

For maximum memory efficiency, game states with different numbers of empty fields are encoded differently, as can be see in the following subsections. This does make sense, since fields with a different number of pieces on the board may have different properties, which can be considered to create more efficient encodings.

4.1 Calculating upper bound for the number of game states

With the Equation 1 the number of possible game states with n fields, n pieces to choose from and k pieces on the board can be calculated. But why does this formula work for calculating the upper bound? When thinking about placing each piece one after the other on the board there are at the beginning 16 possibilities to place the first piece. This is not the 0000 (0) piece, but just any piece that is placed as the first piece on the board. Then there are 15 remaining possibilities to place the second piece and so on. So intuitively there are $16 * 15 * 14 * ... * 2 * 1 = 16!$ possibilities the board is arranged, when the board is full. Since each time we place a piece there is one less option where we can place it. Because this place was taken by the piece that was placed before.

$$\frac{n!}{(n-k)!} * \binom{n}{k} \tag{1}$$

$$n = 16 \tag{2}$$

$$\frac{16!}{(16-k)!} * \binom{16}{k} \tag{3}$$

Since the board consists of 16 fields and has 16 pieces to choose from, $n = 16$ is always the case for the game Quarto. Equation 3 gives the same upper bound for the number of possible game states, when setting $k = 16$, as the intuitive solution we discussed before.
The first term, $\frac{16!}{(16-k)!}$, is calculating the number of game states when choosing k pieces once and placing them on the board in all possible permutations. The second term, $\binom{16}{k}$, calculates the number of combinations of choosing k pieces out of 16, so all possibilities of combinations of pieces that are not on the board. Since all possible configurations of the board should be considered when calculating the upper bound, these two terms have to be multiplied.

4.2 Encoding different game states

In this section the encoding for the different number of pieces on the board is explained. The only assumption for the different encodings that is made, is that we know how many pieces are on the board. In the following subsections e is used to denote the number of possible states on the board. Therefore $\log_2(e)$ equals the number of bits needed to represent a number that is mapped to a state in the range $[0, \ldots, e)$.

There are $b = 121$ possibilities to bitorder (Subsection 3.2.1) a permutation of 3 numbers that are distinctly picked from the range $[1, \ldots, 14]$.

4.2.1 Encoding for 16 pieces on the board

$$e_{16} = b * 9 * 11!$$

To encode 16 pieces on the board, first XORing is done with the first piece on the board (Subsection 3.1.2). After that there is a 0000 (0) piece on the first position of the board. Since the 0000 (0) piece should stay on the first position, the only further transformation that can be done, is a left-diagonal reflection. This is done only if the 1111 (15) piece is in the upper right corner (the white fields in Table 13). The first three fields after the 0000 (0) piece (marked with "*" in Table 13) are bit-ordered (Subsection 3.2.1).
They have b distinct possibilities to be ordered, when only considering ordering pieces in the range $[1, 14]$. This condition is met, since the 0000 (0) piece is on the first position and the 1111 (15) piece is not in the upper right corner.
The 1111 (15) piece is encoded separately. It can only be on one of the gray fields (see Table 13), since the 0000 (0) piece is on the first position, there are only 9 possible other fields the 1111 (15) piece can be.
At last, the permutation of the remaining 11 fields are encoded (Subsection 3.2.2), which gives 11! possibilities for them.
The game-board is completely encoded, the different encodings for the different parts of the board are combined to a single number (Subsection 3.2.7). This number can be converted back to the game-field, when the way it was encoded is known. To note is, that it can only be decoded to the point after all the transformations are done. So not to its original form, but that does not matter, since this decoded game-field is still equivalent to the original game-field it was encoded from. The transformations are done for exactly the reason to reduce the number of distinct game states that are possible, so there is a $1 : M$ mapping done between 1 distinct state and M non-distinct boards.

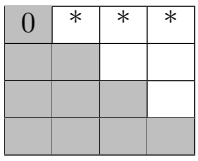


Table 13: Encoding of 16 pieces

4.2.2 Encoding for 15 pieces on the board

$$e_{15} = b * 25 * \binom{11}{1} * 10! + b * 2 * 11!$$

This formula, is different from the one, when encoding 16 pieces, because it considers the two cases, when the 1111 (15) piece is on the board and when it is not, separately.

Lets first look at the first case where the 1111 (15) piece is on the board, which is represented with this part of the formula: $b * 25 * \binom{11}{1} * 10!$. In this case, first the empty field is aligned to the last or second last position. Then if the 1111 (15) piece is on the right upper half a diagonal flip is done. After this, XORing all pieces with the first piece and bit ordering the 3 pieces that come after it as in Subsection 3.1.2 is done. This results in a game-state represented by Table 14. The 0000 (0) piece is on the first position, the "-" mark where the empty field can be and the gray fields mark where the 1111 (15) piece can be. Furthermore the "*" mark where the bit-ordered pieces are.
There are b distinct possibilities to order the 3 bit-ordered pieces, as can be seen in Subsection 3.2.1. There are 25 possibilities the position of the empty field and the position of the 1111 (15) piece can be arranged, as seen in Subsection 3.2.5. The $\binom{11}{1}$ stands for the amount of values the piece that is not on the board can have, since the 0000 (0) piece, the 1111 (15) piece, and the bit-ordered pieces are on the board, there only remain 11 pieces to choose from, that are not on the board. The remaining 10 pieces that are on the board are encoded to one of 10! possibilities (Subsection 3.2.2).

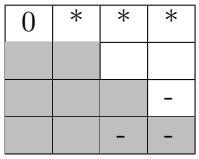
In the second case where the 1111 (15) piece is not on the board the encoding looks a bit different, represented by the other part of the formula: $b * 2 * 11!$. First the transformation of the game-field is done, which is the same as in the first part. The only exception is that we don't have to care about the 1111 (15) piece, since we know it is not on the board. This results in a state represented by Table 14. The empty field only has one of 2 positions. The bit-ordered pieces are again represented by a number in the range $[0, \ldots, b)$. There remain the 11 pieces on the board that are represented together by only one number between $[0, \ldots, 11!)$ (Subsection 3.2.2). The missing field does of course not need to be encoded, since we know it is the 1111 (15) piece in this case.

Combining the different parts for the encoding into one number is done

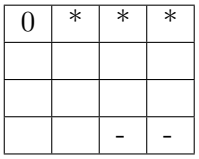
for the two cases as seen in Subsection 3.2.7. This results in a number c. If the 1111 (15) piece is one board $c_{final} = c$ and we are done. Otherwise c_{final} is calculated with Equation 4.

$$c_{final} = b * 25 * \binom{11}{1} * 10! + c \tag{4}$$

To distinguish the cases if the 1111 (15) piece is on the board or not, c_{final} is calculated. Every number from $[0, \ldots, e_{15})$ can be distinctly mapped to a game-state.



(a) the 1111 (15) piece is on the board



(b) the 1111 (15) piece isn't on the board

Table 14: Encoding of 15 pieces

4.2.3 Encoding for 14 to 5 pieces on the board

$$e_{14 \text{ to } 5} = p * b * \left(n * \binom{11}{k} * \alpha! + \binom{11}{k-1} * (\alpha + 1)!\right)$$

If x is the number of pieces on the board, then:

$$n = x - 1$$
$$\alpha = x - 5$$
$$k = 16 - x$$

Let us first define what the different letters mean. First, n is the number of positions on which the 1111 (15) piece can be. Second, α is the number of arrangements/permutations the remaining pieces can have. If $\alpha = 0$, then the definition of $0! = 1$. Furthermore, k is the number of pieces that are not on the board, excluding the 1111 (15) piece. Moreover, p is the number of possible configurations of the positions of non-empty fields or empty fields. For $x \geq 8$ the positions of non-empty fields are encoded. For $x < 8$ the encoding changes to encode the positions of the empty fields instead of the

positions of the non-empty fields. Only either the empty fields or the non-empty fields have to be encoded, since from one the other can be derived. This is done, because p is smaller for $x < 8$ if you encode the positions of the empty fields instead of the non-empty fields (see Subsection 3.2.6).

There are again two cases, where the 1111 (15) piece is on the board and where it is not. But let us first look at the things that are done in the same way for the two cases.
First, the positions are translated to a number in the range $[0, \ldots, p)$ as stated in Subsection 3.2.6. Then the bit-ordering is done as in Subsection 3.2.1. The difference here is that the algorithm first searches row after row and picks the first piece that it finds. With this piece XORing is done. The next three pieces it finds, that are not the 1111 (15) piece, are bit-ordered and encoded to a number in the range $[0, \ldots, b)$.
After these two steps, it again depends on if the 1111 (15) piece is on the board or not. For the first case, if it is on the board, the position of the 1111 (15) piece is represented as a number in the range $[0, \ldots, n)$. Its position is relative to the other numbers and since after XORing it is always after the 0000 (0) piece $n = x - 1$. Then the α remaining pieces, excluding the bit-ordered ones, are encoded as in Subsection 3.2.2 to a number in the range $[0, \ldots, \alpha!)$. Also the k missing pieces are encoded as seen in Subsection 3.2.4 to a number in the range $[0, \ldots, \binom{11}{k})$.
In the second case, if the 1111 (15) piece is not on the board, the α+1 remaining pieces on the board are encoded to a number in the range $[0, \ldots, (\alpha+1)!)$ (see Subsection 3.2.2). The $k - 1$ missing pieces, are encoded to a number in the range $[0, \ldots, \binom{11}{k-1})$ (see Subsection 3.2.4). Here again the 1111 (15) piece is not part of the missing pieces that are encoded, because we already know it is not on the board in this case.
The resulting encoded number c_{final} that lies in the range $[0, \ldots, e_{14 \text{ to } 5})$ is calculated the same way as in Subsection 4.2.2 to distinguish between the two cases.

4.2.4 Encoding for 4 to 2 pieces on the board

$$e_{4-2} = a * \binom{14}{z^*} * \left(b_0 * n * \binom{11}{k} + b_1 * \binom{11}{k-1}\right)$$

If x is the number of pieces on the board, then:

$$
\begin{aligned}
n &= x - 1 \\
k &= 16 - x \\
a &= 14 \text{ (does not depend on } x) \\
z^* &= x - 2 \\
b_0 &= \begin{cases} \text{if } x == 4 \text{ then } 16 \\ \text{if } x == 3 \text{ then } 3 \\ \text{if } x == 2 \text{ then } 1 \end{cases} \\
b_1 &= \begin{cases} \text{if } x == 4 \text{ then } b \\ \text{if } x == 3 \text{ then } 16 \\ \text{if } x == 2 \text{ then } 3 \end{cases}
\end{aligned}
$$

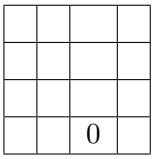
For encoding the board if only 4 to 2 pieces are on the board, everything stays the same as when encoding 14 to 5 pieces. The only difference is that encoding the positions of the non-empty fields is done differently. When encoding the positions, first two positions of non-empty fields are taken, for which the game-field is transformed to one of $a = 14$ distinct positions of the two non-empty fields, as can be seen in Subsection 3.2.3. Then the remaining positions of non-empty fields are encoded to a number between $[0, \ldots, \binom{14}{z^*})$, since there are only 14 fields left where the other empty fields could be, also see Subsection 3.2.4 for this. As can be seen from the formula, b_0 and b_1 are also different from the encoding of 14 to 5 pieces and vary, depending on the number of pieces. The reason for this is, that there are no longer enough pieces on the board to bit-order. So for $x = 4$ pieces on the board and no 1111 (15) piece is on the board $b_1 = b$, so 3 pieces are bit-ordered. But if the 1111 (15) piece is on the board only 2 pieces are left to be bit-ordered, since the first piece is always XORed to be zero, which results in $b_0 = 16$ possibilities to order the two pieces. Furthermore for $x = 2$, $b_0 = 1$ and $b_1 = 3$, since for b_0 no pieces are bit-ordered and for b_1 only one piece is bitorded, for which there are 3 possibilities.

4.2.5 Encoding for 1 piece on board

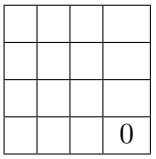
$$e_1 = 2$$

For encoding the board with only one piece on the board, it turns out that the are only 2 non-equivalent states, to which all boards can be mapped

to. First the piece is aligned to the second last or last position of the the board, then it is XORed with itself. With this, it does not matter with which piece you start and in terms of distinguished states it only matters if you place the piece on a diagonal or not. See the two different states in Table 15.



(a) the state $e_1 = 0$



(b) the state $e_1 = 1$

Table 15: Encoding of board with one piece on it

4.3 Possible improvements to the encodings

There are several small improvements to the encodings, that I don't want to keep unmentioned.
First, in the encodings it is still possible to encode illegal positions, so for example positions with two winning rows. This is unnecessary, but on the other hand not obvious how to avoid it. Second, maybe some symmetries can be used even better, but it would probably make the encoding even more complicated. Third, the bit-ordering could be done even for more pieces on the board, but this would only gain minor improvements, since the ordering of bits are restricted by the other pieces and therefore the method becomes more ineffective the more pieces you bit-order. Choosing 3 pieces to bit-order was a conscious choice to not make the decoding unnecessary more difficult for only a small performance gain. Since there are only a few more than a hundred possibilities to order three pieces, they can still just be mapped to a number and back. But if this number of possibilities gets too large you need an algorithm to do the mapping, which would mean more complexity.

In Table 16 it can be seen how many bits are used for the encodings.

5 Practical steps to ultra-strongly solving the games

To realize the goal of ultra-strongly solving this game, one approach is to rate all non-equivalent states and save those ratings in a file. Here rating means how many half moves it takes that either player wins or loses or if

the position is currently a draw. This is done via an algorithm described in Subsection 5.1 in depth.

5.1 The rating algorithm

Since Quarto and simple Quarto are two player games, there exists a player A and a player B. Player A starts and tries to maximize their winning chances, then it is player B's turn and they try to maximize their winning chances of course, therefore trying to minimize the winning chance of player A. So the thing that is maximized and minimized in this algorithm are the winning chances of player A. In each half move the current player tries to make the best move therefore, to do this they have to look at all the next possible moves and pick the best one. To be able to pick the best one they have to again go one depth further and pick the best one for the other player. As can be seen, this is a recursive function call until at some depth the game ends. So the idea is to create non-equivalent continuations of all possible playable moves, and to evaluate the positions and determine the rating for each non-equivalent configuration. If two configurations are equivalent they don't have to be rated twice, since they have to have the same rating. This is the case because all their continuations are equivalent.

5.2 How does rating one state work for simple Quarto?

We are given a board with a certain depth x, so there are x pieces on the board. We want to determine the rating for this board, that is, how many half-moves it takes to win, draw or lose assuming perfect play from the winning player or from both in case of a draw. One half move is defined as one turn, so taking a piece and placing it. So for instance, a rating of 0 would mean on the current board is a winning position. A rating of 1 means from the current board you can make a move that is winning. A rating of 2 means if you play the best move you will loose in 2 half-moves and so on. So all odd ratings are winning for the player whose turn it is. Likewise all even ratings are loosing. Draw ratings are just a draw. So determining the worst or best rating depends on whether a rating is even or odd.
To find the rating for one state at the current depth x we need to find the ratings for all possible next boards at depth $x + 1$, that is, try out possible moves and get the ratings for the resulting boards. Then we choose the worst rating r_{x+1} out of all the tried moves. To do this, we first look for the smallest even rating among all the resulting boards. If no even rating exists, we try to find a draw rating. If a draw rating also does not exist,

we determine the largest odd rating among the resulting boards. The worst rating r_{x+1} is used to calculate r_x the rating for the current state.

$$r_x = \begin{cases} \text{if } r_{x+1} == -1 \text{ then } -1 \\ \text{else } r_{x+1} + 1 \end{cases}$$

If $r_{x+1} = -1$ this means the worst rating for possible next boards is a draw, which means of course r_x is also a draw. Since the number of half moves increased by one from r_{x+1} to r_x, $r_x = r_{x+1} + 1$.

It makes sense to take the worst rating from the next depth, since you try to make the best move, which should result in the worst position for your opponent. In simple Quarto you can choose your piece freely, so when you try to play perfectly you just place the piece to a position that results into the worst rating for your opponent. Your opponent does the same and so on.

5.3 How does rating one state work for Quarto?

For Quarto, the rating of one state works similar, but a bit differently, since you have to take into consideration that your opponent gives you the piece. So you again try out all moves with all the pieces your opponent can give you and get the ratings from the resulting boards. For each piece you pick the worst rating of all possible positions you can place the piece. But what is different from simple Quarto is that r_{x+1} is the best rating of the worst ratings that you picked for each piece. Then you calculate r_x the same way as before.

Why does this make sense? Since your opponent tries to minimize your chance of winning they will try to give you the worst piece for you, but he can not control where you place the piece, so you place it of course at the worst position for him.
To clarify, the rating of one state says, if both players play perfectly from this state on then the game is a win, draw or loose in that many half-moves. One half move in Quarto consists of placing the piece your opponent gave you and giving your opponent a piece.

5.4 Memory requirements for saving the ratings

To generate and then save all the ratings of states we need memory to save them, as it turns out, exactly as much as seen in Table 16. Since saving one rating of one state needs exactly one byte.

x	Bit size (bits)	Total space needed (GB)
16	35.34	40.48
15	36.92	121.45
14	38.88	472.31
13	40.00	1023.35
12	39.98	1012.10
11	39.22	594.33
10	37.79	221.82
9	35.77	54.53
8	33.18	9.07
7	29.60	0.76
6	25.49	0.04
5	20.87	0.00
4	17.72	0.00
3	12.07	0.00
2	5.81	0.00
1	1.00	0.00
Total space (x = 1 to 16)		**3550.25 GB**

Table 16: Bit size of one encoding and space needed to store the ratings of all the states

The bit size of one encoding is calculated as $\log_2(e_x)$, where e_x refers to the encoding for x pieces on the board.
Our resources where limited, we started by calculating the ratings for each state of depth $x = 11$ and saved this. This process was done with parallel computing the ratings of the states. To rate one state, one has to compute all non-equivalent continuations, this can be done in parallel. At depth $x = 11$, the computational requirements remained within the feasible range for this project. Then for $x = 10$ (and below) only a single process was used, since we only need to calculate all non-equivalent next states with $x = 11$ (and below) to find the best rating of the current state. As we don't need to build all non-equivalent continuations for one state, this is less computationally

expensive of course.

6 Playing against the machine on a website

The game is implemented on a website, hosted by TU-Graz, where you can try your luck to play against the machine. The link to the website is: `https://gamesatcs.tugraz.at/quarto/`.
After depth 9 the machine may take a bit longer to compute the best move, since the ratings are not already saved on the server, but have to be calculated online by recursively building all non-equivalent continuations from the current state to rate a given position. To compute all the ratings from depth 9 and below we had to compute and save the ratings at depth 11. But as seen in Table 16 there is a lot of memory needed to save the ratings a depth 11 and 10. That's why we decided to delete the ratings of those depths, since it made little difference performance wise on how long it takes to get the rating for a board. Stating the rating process from depth 9 instead of depth 11 would have taken much longer. That is why we decided on this approach.

6.1 How does the machine make a move?

With the ratings at hand, the machine can easily make a move from any position. For simple Quarto it just has to try each move that is possible with the current board and then choose the move with the piece and the position where the rating is worst.

For Quarto it's a bit more complicated, first you try all positions with the piece you where given and place the piece on the position for which the rating is worst. Then you still have to give your opponent a piece. You do that by trying all possibilities for all the remaining pieces to place. Then you calculate the best rating for each piece and choose the piece with the worst rating.

7 Acknowledgments

I want to say thanks to my family and friends who gave me the strength and support to finish my thesis and my Bachelor program. Furthermore, I sincerely thank Assoc.Prof. Dipl.-Ing. Dr.techn. Oswin Aichholzer for his guidance and support throughout this project. I also appreciate his provision of the necessary computing resources, which were essential to parts of this work. Lastly, I want to thank my sister for reminding me that I should write this section.

References

[1] W. Myrvold and F. Ruskey, "Ranking and unranking permutations in linear time," 2000, accessed: 2024-06-03. [Online]. Available: https://web.archive.org/web/20240603092849/http://webhome.cs.uvic.ca/%7Eruskey/Publications/RankPerm/MyrvoldRuskey.pdf

[2] J. McCaffrey, "Generating the m-th lexicographical element of a mathematical combination," 2004, accessed: 2024-06-03. [Online]. Available: https://web.archive.org/web/20240603093334/https://learn.microsoft.com/en-us/previous-versions/visualstudio/aa289166(v=vs.70)