GamesCrafters 2006

The Quarto! Module

Documentation for Developers

Mario Tanev (Dev)

Amy Hsueh (Dev)

Yanpei Chen (Dev, Doc)

Revision Summary			
Author	Date	Version	Comments
Yanpei Chen	2006.12.21	0.1	Created

Table of Contents:

1. Overview

Quarto! is a two player board game invented by Blaise Muller. It is played on a 4 by 4 square board, with 16 pieces that carries 4 binary characteristics. It is a popular game, and its intrinsic symmetries has mesmerized many mathematicians and board game enthusiasts.

Quarto! work on in GamesCrafters began in January 2005. Quarto! represents one of the first "Big Games" tackled by the group. Work in Quarto! is ongoing, even though two of the initial developers have graduated from UC Berkeley.

2. Goals and Non-Goals of this Document

This documentation targets developers who maintains Quarto! code and developers who tries to understand some of the complex algorithms found in the Quarto! module.

Readers should walk away with the following:

- High level appreciation of Quarto! game module design.
- Understanding of how the Quarto! board, position, and moves are represented.
- Entry knowledge for complex Quarto! algorithms, including hash and symmetries.
- Entry knowledge for key functions in the Quarto! module.

All other issues not listed as goals would be considered non-goals.

In particular, this documentation will not try to do the following:

- Explain Gamesman architecture.
- Explain the common API used by game modules to interact with Gamesman.
- Explain the C programming language.

It is assumed that readers already have some experience in software development, and would have some familiarity with the items above.

This document is intended as a companion to Quarto! code. For low level implementation specifics, the reader should look at Quarto! code, found in ./gamesman/src/mquarto.c.

3. Game Rules

Quarto! is played between two players.

3.1. Board and Pieces

Quarto! is played on a 4 by 4 square board.

Quarto! has 16 pieces. Each pieces has four binary characteristics. Usually the characteristics are dark or light color, tall or short, square or round, with or without a hole. For example, a piece may be light, tall, round, and without a hole.

A picture of the Quarto! board with all 16 pieces is shown on the cover page.

3.2. Moves

The two players take turns choosing a piece, which the other player must place on the board. The move sequence to start a game would be:

1. A chooses a piece for B.
2. B places the piece given on any square on the board.
   B chooses a piece for A.
3. A places the piece given on any square on the board.
   A chooses a piece for B.
4. ……

And the game continues.

3.3. Winning

A player wins by placing his or her piece to make a row, column, or diagonal with all four pieces sharing a common characteristic, e.g. all tall, all round, etc.

Winning variant

In the most straightforward form of Quarto!, the first player to do this is declared the winner. There is a variant in which the winner is the first player to place such a piece and to identify it. In this variant, if the "winner" fails to identify that he or she has won, then the game goes on until either someone wins and announces that he or she has won, or until all the pieces have been exhausted.

Ties

If the pieces have been exhausted and there is no winner, the game ends in a tie.

3.4. Variants

In addition to the variants involved in winning, there are several more variants.

Board size

Some variants may alter the board size. On a 3 by 3 board, there would be $2^3 = 8$ pieces, and the game is won by having 3 in a row with same characteristics. On a 5 by 5 board, there would be $2^5 = 32$ pieces, and the game is won by having 5 in a row with same characteristics.

In the rest of this document, a N by N board has GAMEDIMENSION of N.

Diagonals

Some variants ignore diagonals that have same characteristics, such that the game is won only by having rows or columns with the same characteristics.

Variants implemented

In our implementation, the winner does not need to identify that he or she has won, and the game ends automatically when there is a row or column or diagonal with the same characteristics. We can solve a 3 by 3 board, the 4 by 4 board cannot be solved due to memory limits, and the 5 by 5 board cannot be represented due to unsigned long long bit width limitations. We also do not implement the variant without winning diagonals.

4. Global Variables for Gamesman

```
BOOLEAN  kPartizan           = TRUE ;
BOOLEAN  kTieIsPossible      = TRUE ;
BOOLEAN  kLoopy              = FALSE ;
```

Quarto is a partisan, impartial, non-loopy game with possible ties.

Partisan

Give the same board, each player would have different moves, because one player is always waiting to place onto the board the piece selected by the other player. The board cannot be used for the other player’s turn because he or she would not have any piece to place onto the board.

Impartial

Quarto! is impartial because both players share the same set of pieces. The partiality of a game does not affect how Gamesman searches the game tree. Thus there is no `kPartial` global to set.

Non-loopy

Pieces are never taken off the board, thus positions cannot be repeated, and Quarto! is non-loopy.

Possible ties

Ties are possible as explained in Section 3.3.

5. Key Design Decisions

To expedite code development and avoid code conflicts, we created function pointers for all the key functions, and for complex algorithms, we wrote multiple implementations. This way, developers would not need to coordinate on modifying the same piece of code, and could work independently on separate implementations. The different implementations would cross check each other for correctness, and we could set the function pointers to the most efficient implementation.

Other design decisions would be presented with the relevant sections.

6. Game Representation

This section details the data types used to present board and pieces, as well as the POSITION and MOVE semantics. Board and pieces data structures are seen only by the Quarto! module, while POSITION and MOVE would be used by Gamesman to solve Quarto!

6.1. Board and pieces

Boards are represented using the `QTBOARD` data structure.

- `short *slots[0:BOARDSIZE]`
  - Record the pieces on the board.
  - `slots[0]` = piece in hand, i.e. the piece about to be placed.
  - `slots[1:BOARDSIZE]` hold the pieces on the game board.

- `short squaresOccupied`
  - Denotes the number of board squares occupied.

- `short piecesInPlay`
  - Denotes the pieces in play, including any piece in hand, i.e. piece about to be placed on board.

- `BOOLEAN usersTurn`
    - Deprecated.

Pointer to `QTBOARD` is `QTBPtr`, and `QTBPtr` gets passed between functions as arguments and return values.

6.2. Hash – POSITION semantics

Quarto! is a large game. There are approximately 2^{63} positions for games with 4 by 4 boards. To fit all of this into 2^{64} possible `unsigned long long POSITION` values, it is crucial that our POSITION hash is tight. We aimed for a packed hash, with all possible positions iterated from 0 to the maximum number of positions. The hash is a fairly complicated algorithm based on combinatorics. It will be described in detail in Sections 7.4.

The initial empty board with no pieces in hand is encoded as 0. Overall, boards with smaller hash values would have fewer or equal number of pieces in play as boards with larger hash values.

The hash space is populated as below. There are no gaps in the hash space.

0	1 16	17	......
Empty hand Empty board	One piece in hand Empty board	One piece in hand One piece on board	

6.3. Hash – MOVE semantics

MOVE semantics is much simpler than POSITION semantics. A move consists of two parts – a slot and a piece. The slot in which to place the piece given, and the piece selected for the opponent. The first move of the game involves selecting a piece of the opponent only, and is identified by placing into a special "hand" slot.

The current MOVE representation uses the MOVE integer as a bit field.

- Piece number occupies the lower GAMEDIMENSION bits, just enough for $2^{GAMEDIMENSION}$ pieces.
- Slot number occupies the other bits.

MOVE constructor: `MOVE CreateMove   ( MOVE slot, MOVE piece )`
MOVE accessor: `MOVE GetMoveSlot  ( MOVE move )`
MOVE accessor: `MOVE GetMovePiece ( MOVE move )`

7. Key Functions

7.1. Primitive(POSITION)

Checking for primitives is relatively straight forward.

- Unhash() POSITION into local board.
- Copies the rows, columns, and diagonals into `rowColDiag = short[GAMEDIMENSION]`.
- Check for `EMPTY` slots in the array.
- If no `EMPTY` found, call `searchPrimitive(rowColDiag)`.
- Return correct results.

`searchPrimitive(short *rowColDiag)` takes advantage of the binary-bit nature of pieces to quickly search for primitives. A primitive is found if the cumulative bitwise AND of all the entries in `rowColDiag` is non-zero. A primitive is also found if the cumulative bitwise AND of the inverse of all the entries `rowColDiag` is non-zero.

e.g. `rowColDiag` = {1111,1010,1101,1100},
AND(1111,1010,1101,1100) = 1000 → Primitive found

e.g. `rowColDiag` = {0000,0101,0010,0011},
AND(NOT(0000), NOT(0101), NOT(0010), NOT(0011)) = 1000 → Primitive found

7.2. GenerateMoves(POSITION)

GenerateMoves() is also relatively straight forward.

- Unhash() POSITION into a local board.
- If the board is empty, create MOVELIST from possible pieces in hand.
- If the board is about to be filled, create MOVELIST from the empty slot and the piece in hand.
- Otherwise build MOVELIST from all empty slots on board and all available pieces. A move constructed will transfer an item from hand into an EMPTY slot, and an available piece into the hand.
- Return MOVELIST.

The available pieces are marked in a helper function:

```
int FlagAvailablePieces( QTBPtr board, BOOLEAN pieces[] )
```

- `pieces` is a `short[NUMPIECES]` array.
- `board` is a locally unhashed board data structure.
- Returns the number of available pieces.
- After returning, `pieces[N]` will be set to FALSE if the Nth piece is no longer available, i.e. it has already been placed onto the board.

7.3. DoMove(POSITION, MOVE)

DoMove() is also relatively straight forward.

- Unhash POSITION to a local board data structure.
- slot and piece is decoded from MOVE using accessors described in Section 6.3.
- board->slots[] appropriately modified using slot and piece from MOVE.
- board->piecesInPlay and board->squaresOccupied modified.
- Return hash(board).

Many functionalities are abstracted away in one-line helper functions, including

- `void GetHandPiece(board)`
- `void SetHandPiece(board, piece)`

7.4. hash()/unhash() for POSITION

hash() and unhash() are inverse operations. We describe here only the hash. The unhash algorithm is the exact mirror inverse of the hash algorithm – everything hash does, unhash does the reverse in reverse order.

Functions involved

- ```
  POSITION (*hash  ) (QTBPtr);
  QTBPtr   (*unhash) (POSITION);
  ```
- ```
  POSITION hashUnsymQuarto   (QTBPtr);
  QTBPtr   unhashUnsymQuarto (POSITION);
  ```
- ```
  POSITION hashUnsymQuartoHelper (QTBPtr b, int baseSlot);
  void     unhashUnsymQuartoHelper (POSITION p, int baseSlot, QTBPtr toReturn);
  ```

hash() and unhash() function pointers are set to hashUnsymQuarto() and unhashUnsymQuarto(). We used the function pointer abstraction to pre-empt possibly multiple implementations. hashUnsymQuart() and unhashUnsymQuarto() respectively calls hashUnsymQuartoHelper() and unhashUnsymQuartoHelper() in recursive fashion.

High level approach

Our hash is based on combinatorics. The hash space is populated from 0 to max iteratively with no space. We let positions with fewer pieces in play occupy smaller hash values – see Section 6.2.

This invites a recursive breakdown of the hash problem, with combinatorial offsets added to the hash value at each recursive stage. The following example illustrates the high level idea for the hash. The hash space is not drawn to scale. Different colors denote different levels of hash recursion.
  ```

Hash recursion example

```
         HAND  SQRT1 SQRT2 SQRT3 SQRT4 SQRT5 SQRT6 SQRT7 SQRT8
Slots = [3     1     0     -     -     -     -     -     2    ]
```

All other slots are EMPTY.

piecesInPlay = 4

piecesInPlay = 0	
piecesInPlay = 1	
piecesInPlay = 2	
piecesInPlay = 3	piecesInPlay offset [3]
… zoom in to next level …	
piecesInPlay = 4 [0 …	
piecesInPlay = 4 [1 …	
piecesInPlay = 4 [2 …	First slot offset × 3
… zoom in to next level …	
piecesInPlay = 4 [3 0 …	Second slot offset × 1
… zoom in to next level …	
piecesInPlay = 4 [3 1 0 …	Third slot offset × 0
… zoom in to next level …	
piecesInPlay = 4 [3 1 0 2 - - - -]	
piecesInPlay = 4 [3 1 0 - 2 - - -]	
piecesInPlay = 4 [3 1 0 - - 2 - -]	
piecesInPlay = 4 [3 1 0 - - - 2 -]	
piecesInPlay = 4 [3 1 0 - - - - 2]	4
Final hash value	**Sum of column above**

Hash value iteration example

To further clarify the above, the following are hash value and slot content outputs for 3 by 3 game. We show hash values for piecesInPlay = 0 and 1, the transition from piecesInPlay = 1 to piecesInPlay = 2, and the transition from piecesInPlay = 2 to piecesInPlay = 3.
The first slot is the “hand” slot.

```
Testing full blown hash/unhash ...

  hash value   0; slots:   -  -  -  -  -  -  -  -  -  -

  hash value   1; slots:   0  -  -  -  -  -  -  -  -  -
  hash value   2; slots:   1  -  -  -  -  -  -  -  -  -
  hash value   3; slots:   2  -  -  -  -  -  -  -  -  -
  hash value   4; slots:   3  -  -  -  -  -  -  -  -  -
  hash value   5; slots:   4  -  -  -  -  -  -  -  -  -
  hash value   6; slots:   5  -  -  -  -  -  -  -  -  -
  hash value   7; slots:   6  -  -  -  -  -  -  -  -  -
  hash value   8; slots:   7  -  -  -  -  -  -  -  -  -

  hash value   9; slots:   0  1  -  -  -  -  -  -  -  -
  hash value  10; slots:   0  -  1  -  -  -  -  -  -  -
  hash value  11; slots:   0  -  -  1  -  -  -  -  -  -
  hash value  12; slots:   0  -  -  -  1  -  -  -  -  -
  hash value  13; slots:   0  -  -  -  -  1  -  -  -  -
  hash value  14; slots:   0  -  -  -  -  -  1  -  -  -
  hash value  15; slots:   0  -  -  -  -  -  -  1  -  -
  hash value  16; slots:   0  -  -  -  -  -  -  -  1  -
  hash value  17; slots:   0  -  -  -  -  -  -  -  -  1
  hash value  18; slots:   0  2  -  -  -  -  -  -  -  -
  hash value  19; slots:   0  -  2  -  -  -  -  -  -  -
  hash value  20; slots:   0  -  -  2  -  -  -  -  -  -
  ……
  hash value 503; slots:   7  -  -  -  -  -  -  -  -  5
  hash value 504; slots:   7  6  -  -  -  -  -  -  -  -
  hash value 505; slots:   7  -  6  -  -  -  -  -  -  -
  hash value 506; slots:   7  -  -  6  -  -  -  -  -  -
  hash value 507; slots:   7  -  -  -  6  -  -  -  -  -
  hash value 508; slots:   7  -  -  -  -  6  -  -  -  -
  hash value 509; slots:   7  -  -  -  -  -  6  -  -  -
  hash value 510; slots:   7  -  -  -  -  -  -  6  -  -
  hash value 511; slots:   7  -  -  -  -  -  -  -  6  -
  hash value 512; slots:   7  -  -  -  -  -  -  -  -  6

  hash value 513; slots:   0  1  2  -  -  -  -  -  -  -
  hash value 514; slots:   0  1  -  2  -  -  -  -  -  -
  hash value 515; slots:   0  1  -  -  2  -  -  -  -  -
  hash value 516; slots:   0  1  -  -  -  2  -  -  -  -
  hash value 517; slots:   0  1  -  -  -  -  2  -  -  -
  hash value 518; slots:   0  1  -  -  -  -  -  2  -  -
  hash value 519; slots:   0  1  -  -  -  -  -  -  2  -
  hash value 520; slots:   0  1  -  -  -  -  -  -  -  2
  hash value 521; slots:   0  1  3  -  -  -  -  -  -  -
  hash value 522; slots:   0  1  -  3  -  -  -  -  -  -
  ……
```

Hash algorithm

The hash/unhash functions and all necessary support functions are several hundred lines long altogether. So our presentation here is necessarily high level and terse.

`POSITION hashUnsymQuarto(QTBPtr b)`

- `b` is allocated and freed by function calling hashUnsymQuarto().
  `b` should not be modified upon return of hashUnsymQuarto().
- If piecesInPlay = 0, return 0.
- If piecesInPlay = 1, return offset[0] + slot[HAND]
- Else recursion needed:
  - Create a helperBoard copy of `b`
  - `squaresOccupiedOffset = offset[b->squaresOccupied]`
  - `firstSlotOffset = permutation(NUMPIECES-1,b->squaresOccupied) *combination(BOARDSIZE, b->squaresOccupied)`
    This is the number of ordered ways we can put all pieces except the piece in hand into all the occupied squares, multiplied by the number of unordered ways we can fill up squaresOccupied square on the board.
  - Iterate through helperBoard->slots[], normalize the board slots such that pieces larger than the piece in hand is decremented by 1.
    This is to make the slots other than the hand passed to the next recursion call contain the set of pieces {0, 1, 2, …… }
  - If squaresOccupied < NUMPIECES, board is still not filled,
    Return squaresOccupiedOffset +
    b->slots[HAND] × firstSlotOffset +
    hashUnsymQuartoHelper(helperBoard, baseSlot =1)
  - Else, board is about to be filled,
    Return squaresOccupiedOffset +
    hashUnsymQuartoHelper(helperBoard, baseSlot =1)

`POSITION hashUnsymQuartoHelper (QTBPtr b, int baseSlot);`

- `b` is allocated and freed by function calling hashUnsymQuartoHelper().
  `b` is modified upon return of hashUnsymQuartoHelper() and should not be reused.
- Setup necessary local variables:
  - `slotsSubset` = number of slots starting from `baseSlot`
  - `slotsOccupiedSubset` = number of occupied slots starting from `baseSlot`
  - `firstSlot` = first occupied slot in the slots after `baseSlot`
  - `firstPiece` = piece in `firstSlot`
  - `piecesBeforeBase` = number of pieces before `baseSlot`
- If slotsOccupiedSubset = 0, return 0.

- If slotsOccupiedSubset = 1, return firstPiece × slotsSubset + (firstSlot – baseSlot).
- Else recursion needed:
    - Calculate
      `firstSlotOffset` = number of hash values before `firstSlot` and
      `firstPieceOffset` = number of hash values for each piece in `firstSlot`.
      This calculation is done using cumulative sums. In particular, calculating `firstPieceOffset` is analogous but not identical to calculating `firstSlotOffset` in `hashUnsymQuarto()`.
    - Iterate through b->slots[] starting from firstSlot+1, normalize the board slots such that pieces larger than the piece in hand is decremented by 1.
      This is to make the slots other than the hand passed to the next recursion call contain the set of pieces {0, 1, 2, ... }
    - Return firstSlotOffset +
      firstPiece × firstPieceOffset +
      hashUnsymQuartoHelper(b, baseSlot = firstSlot+1)

Unhash algorithm

The unhash() algorithm is the exact reverse of hash().

The recursion on unhash() deconstructs the POSITION into QTBOARD in the exact opposite manner as hash() constructs POSITION from QTBOARD. The naming of local variables and the use of pHelper (helper position) is parallel to that in hash().

8. Game Initialization

This section describe the `InitializeGame()` function, which calls `initGame()`, with the `initGame()` function pointer set to `yanpeiInitializeGame()`. There is a deprecated `marioInitializeGame()` which does not contain initializations necessary to facilitate the `hash()/unhash()`.

```
void yanpeiInitializeGame()
```

- Set global variables.
- Set `factorialTable`, used by hash and canonicals to lookup factorial values.
- Set `offsetTable`, used by hash.
- Set `lookupTable`, used by canonicals. This requires hardcode setting `trivialTable`.
- Initialize an empty board, and set `gInitialPosition` to its hashed value.
- Set other global variables and function pointers.
- Call any test functions for hash or canonicals.

9. Options Supported

- Symmetries
  - getCanonical(), yanpeiGetCanonical(), marioGetCanonical().
  - Verified for 3 by 3 and 4 by 4 games.

- Move to string
  - Not implemented.
  - Should be implemented to facilitate move history – Section 10.2.

- GPS
  - Implemented.
  - Need additional verification.

- Tier
  - Not implemented.
  - Should be implemented to facilitate strong solution for 4 by 4 – Section 11.2.

9.1. Comment on Canonicals/Symmetries

Functions involved

```
POSITION (*getCanonical)(POSITION p);
POSITION yanpeiGetCanonical(POSITION p);
POSITION marioGetCanonical(POSITION position);
```

The second two functions represent two different implementations. Both implementations have been verified by hand and cross checked against each other. Presently `getCannonical()` is set to `marioGetCanonicals()` because that implementation is much faster.

Support functions

Both implementations call a host of support functions, too many to list and explain in detail here. Most of these function abstract away frequent operations involved in rotating, reflecting, otherwise normalizing boards and sub-board, or masking pieces and sets of pieces. We have tried to have intuitive naming and comment our code as much as possible.

Code for the support functions are located close to the code for the principle `yanpeiGetCanonical()` and `marioGetCanonical()` functions.

High level description of symmetries

`yanpeiGetCanonical()` takes advantage of the geometric symmetries in the game board, as well as the symmetries in the binary characteristics of the pieces. The symmetries in the pieces are analogous to rotations in an N-dimensional binary bit hypercube, where N is the GAMEDIMENSION. We compute rotations of an N-dimensional hypercube from rotations of an N-1 dimensional hypercube. A 2-dimensional hypercube is the 2-bit square, a 3-dimensional hypercube is the 3-bit cube. The canonical POSITION is found by fanning-out a given POSITION to its symmetric equivalents, hashing all symmetric POSITIONs, then fanning-in by setting canonical POSITION to the minimum hashed value.

`marioGetCanonical()` is a more complex algorithm that avoids the fan-out given POSITION of symmetric equivalents followed by fan-in to find the minimum canonical POSITION. It does normalization on the given POSITION to transform it step by step to the canonical. The original author for `marioGetCanonical()` has graduated from UC Berkeley. We are trying to contact him for more detailed input for this doc.

Explaining the canonicals in more detail would require comparable space to explaining the hash. To keep this document a reasonable length, we defer a detail explanation here.

10. C Code Status

10.1. Variants implemented

- Winning:
    - Automatic win – implemented.
    - Need to declare win – not implemented.
    - Reverse win objective – not implemented.

- Diagonals:
    - Wins include diagonals – implemented.
    - Wins do not include diagonals – not implemented.

- GAMEDIMENSION:
    - Dimension 3, 3 by 3 board – solved.
    - Dimension 4, 4 by 4 board – cannot be solved, memory limitation.
    - Dimension 5, 5 by 5 board – cannot be represented, bit width limitation.

10.2. Open bugs/issues

- IO polish – There continues to be lingering text input issues due to continuously updating Gamesman architecture. Quarto! code need to keep up with changes in text input conventions and updates in the reserved character and escape character sets.

- GMP verification – One of the Quarto! developers integrated GMP support in Gamesman specifically to facilitate correct representation of 5 by 5 games. The correct functionality of GMP should be verified.

- Implement MoveToString() to facilitate move history.

- Polish game specific menu.

- Polish code.

- Replace hard-coded values for EMPTY and HAND.
- Check for NUMPIECES vs. GAMEDIMENSION^2 confusion. Test thoroughly for GAMEDIMENSION = 3, where NUMPIECE = $2^3 = 8 \neq$ GAMEDIMENSION$^2 = 9$.

10.3. Help strings

Implemented help strings include

- kHelpTextInterface
- kHelpOnYourTurn
- kHelpStandardObjective
- kHelpTieOccursWhen
- kHelpExample

Unimplemented help strings include

- kHelpGraphicInterface
- kHelpReverseObjective

11. Looking Forward

11.1. Quarto! GUI

A first-pass GUI module is implemented in ./gamesman/tcl/mquarto.tcl. It is an incomplete module, and almost certain to contain critical bugs. Work on mquarto.tcl has not progressed since its chief developer graduated from UC Berkeley. Completing and polishing mquarto.tcl can elevate one of the most advanced games in Gamesman to a highly presentable level.

11.2. Solving Quarto! Using Tiers

Tierifying Quarto! would likely bypass the memory and bit width limitations for solving 4 by 4 and 5 by 5 Quarto!.

Tier semantics is intuitive – tiers are defined by the number of pieces in play.

11.3. Continuously Optimized Architecture

Quarto! energized quite a few architectural optimizations in Gamesman. Since work on Quarto! began, there has been a continuous stream of new options and features added to Gamesman. All these architectural improvements, including new databases, tiers, and others, make it more and more likely that a strong solution for 4 by 4 and even 5 by 5 Quarto! would be found.

Offering a strong solution for Quarto! would be a significant step up from present cutting edge Quarto! research by Kerner (http://zoo.cs.yale.edu/classes/cs490/00-01b/kerner.matthew.mmk29/quarto/quarto.html) and the original Quarto! solution by Goosens (link no longer working), both of whom used Mini-max pruning. It would gain wide reception among the legions of Quarto! enthusiasts, as well as verify extensive work on Quarto! combinatorial mathematics. A strong solution for Quarto! would tremendously raise the public profile of the Gamescrafters group.

It is the sincere hope of the original Quarto! developers that the Quarto! project would be sustained, because a strong solution for 4 by 4 is just beyond the horizon. And as far as we know, Gamescrafters would be the first to strongly solve Quarto!