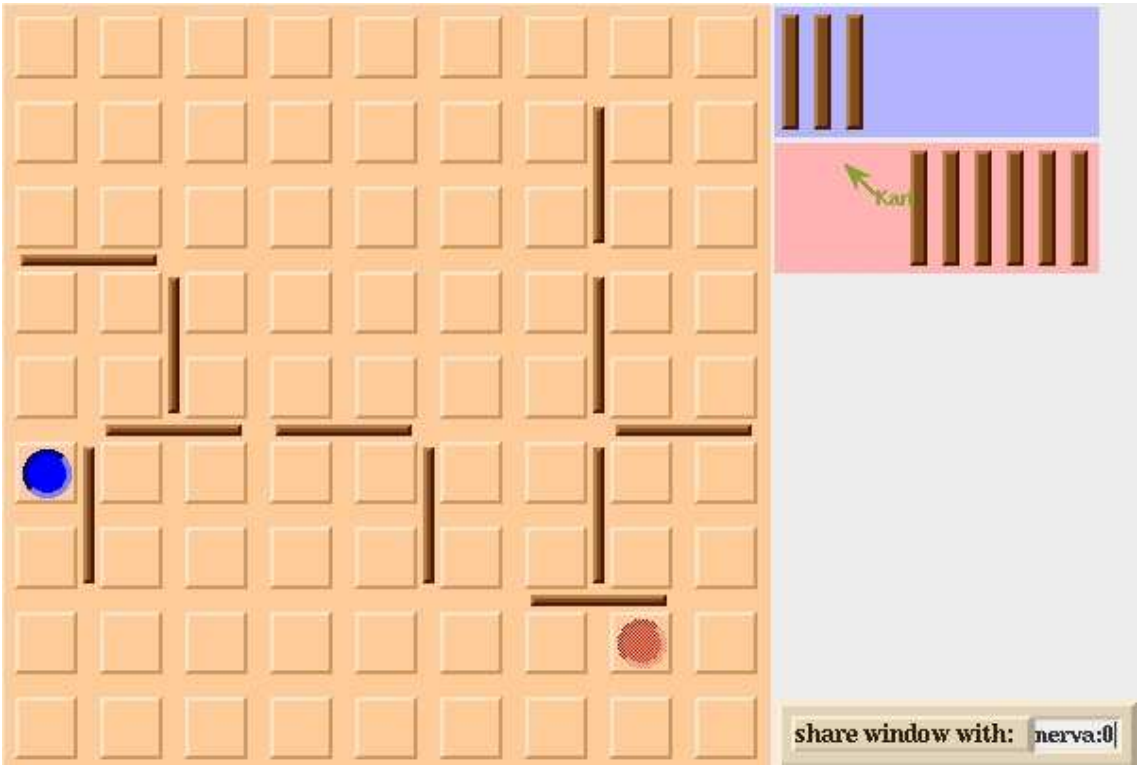
University of Aarhus
Department of Computer Science
Ny Munkegade
8000 Århus C

December 3, 2002

Aspects of Object Oriented Programming

Second Project: Quoridor



Kasper Jensen
Christian Plesner
Kari Schougaard

Contents

1 Introduction

This is our report of the second assigment in Self in the course dAOOP. In the following sections we will explain what we have done, how we did it and our experiences with the prototypical programming language/environment Self. The snapshot which is the result of the project can be found as ~`kari/AOOP/Self/objects/quoridor.snap`.

1.1 Goals

We set out to implement a board game called Quoridor. We figured there would be three subtasks besides getting used to and learning Self.

- Implementing the rules
- Implementing a graphical user interface using morphs
- Implementing support for a two-player game via LAN

We did succeed in implementing all three subtasks. We should mention, however, that support for a two-player game turned out to be a feature in Self, not something we made ourselves.

1.2 Quoridor

Quoridor is a board game developed by gigamic

`http://www.gigamic.com/jeuxanglais/quoridore.htm`

We have implemented an electronic version of the game. A java applet of the game already exists, and some inspiration for the layout has been taken from that.

Quoridor is a two player game with a red player and a blue player. The game consists of a board with $n \times n$ fields, where n defaults to nine. Each player has a piece and ten walls. The blue player starts on the left side of the board and red player starts on the right side. In each turn a player can either move to an adjacent field or set a wall. Only ten walls can be put on the board by each player and they cannot be moved. A wall has the length of two fields and neither player can move through walls. Both players shall at all times have a route to the other side of the board. If the two players are at adjacent fields they may jump over each other. If a player cannot jump over the other because the field on the other side of the other player is blocked by a wall, the player is allowed to jump diagonally to the fields on either side of the other player. When a player reaches the back row on the other side, that player has won.

2 Implementation

In the implementation we have focused on getting the Morphs for the graphics to work. We also used some time figuring out how to implement the rules in an efficient way, but as we got to the implementation little time was left, and we opted for the easiest solution instead of the most efficient.

Throughout this report we have used the convention that `courier` is used when referring to slots, helvetica is used for objects in general, and traits objects are written in *italic helvetica*.

2.1 Structure

We started out by making a namespace for our objects: quoridor. That was done by placing an object in the `quoridor` slot of the lobby and letting all objects we made have a `root*` slot referring to that object.

The objects are structured as follows: The quoridor object has a slot called `traits` and a slot called `prototype`. The quoridor traits object consists of all the different traits: *traits board*, *traits field* etc, see fig. 1 . This is a very common way

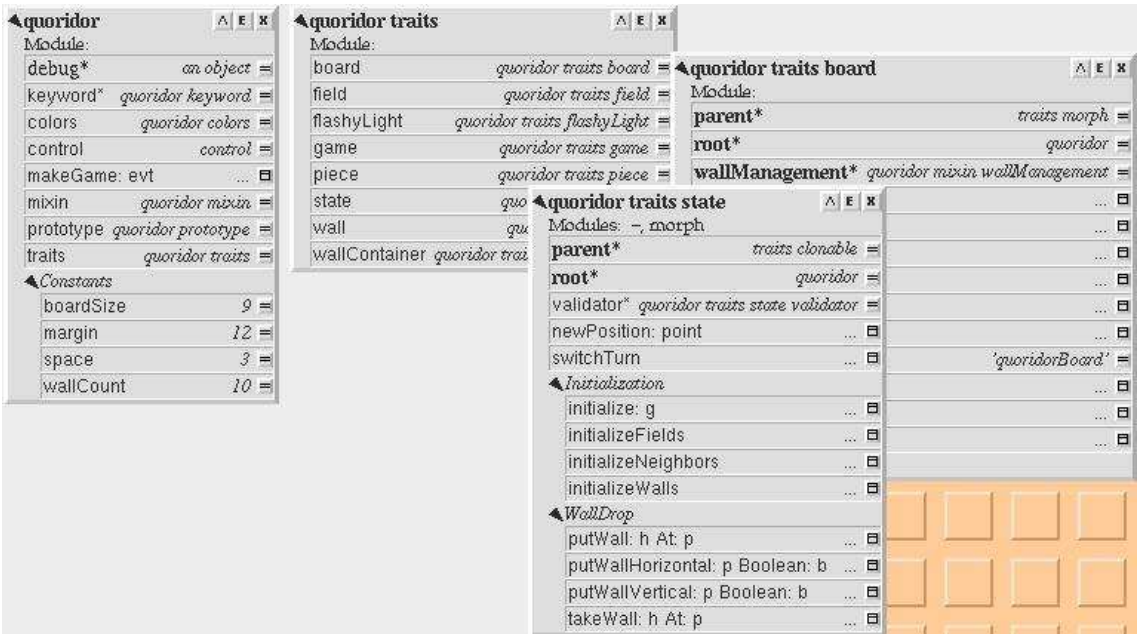


Figure 1: Trait objects in Quoridor

to structure objects in Self: the traits are parent objects and hold all the method slots. Thus traits are like classes in that several objects share the slots in them. The difference from classes is that traits do not hold any object-specific state and are not involved in the making of new objects. The new objects are copied from

the prototypes in quoridor prototype, see fig. 2 . New objects are actually not

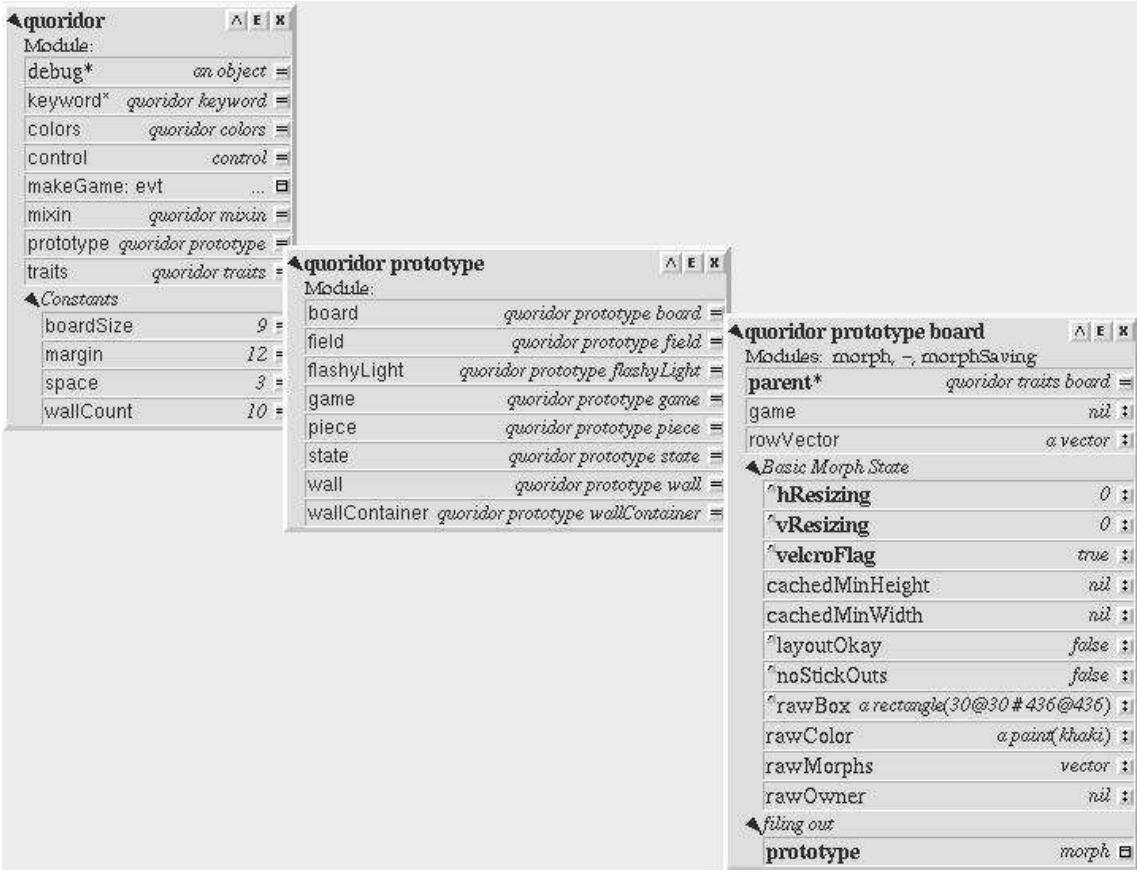


Figure 2: Prototypes in Quoridor

instantiated by copying explicitly, but rather by the custom keyword `new`.

We have specified that these objects are to be used as prototypes by their name. Having prototype objects is used in self all the time, but the use of custom annotations such as prototype seems to be very limited. We have only met traits and mixins.

Now we will go through the different objects and explain their responsibilities. The objects will be presented as a couple of prototype and traits because that is the way they are designed.

quoridor, as already mentioned, works as a namespace for the objects involved in the application. As such, it also holds some constants eg. the size of the board, the space between fields and how large the margin of a field is. Margins will be explained later. It is because the view is modelled first thing that stuff like space and margin is in such a central place. You could argue that they should have been put away in a view; instead we have the

model isolated in state. Quoridor also have a `debug` and `keyword` parent, and the annotationd `colors` and `mixin`, which will all be described later.

Besides being the holder of the different objects quoridor has the responsibility of making a new game.

game consists of the board the two containers of walls and the state, brought together in the prototype game. The *traits game* only has a initialization slot.

board holds the fields in the prototype board and the information related to it being a Morph. The *traits board* is the point of interaction with the state. If anyone (that is, fields, pieces etc.) need to know something about the state of the game, they usually get that information through the board.

field takes care of stuff related to it being a graphical element and knows where in the board it's placed. The *traits field* can add a piece to the field, change the color for highlighting possible fields to move to.

wall is a graphical element, that can be moved around and either be horizontal or vertical, this accounts for most of its state. The *traits wall* has methods to use when the wall cannot be put down, where the user places it.

piece is yet a graphical element that can be moved around and perhaps has to be moved back automatically if droped in an illegal place.

state holds the generic[1] fields, positions of walls, pieces and whose turn it is. The *traits state* has methods for getting the possible moves for the current player and check if a wall can be placed in a certain spot – implementing the rules, basically.

state generic field is the lightweight fields that the state uses, the prototype consists of the neighborfields, there is no traits connected to the field

2.2 Rules

Two players take turns, where a turn consists of either moving the piece or placing a wall. When a wall or a piece is placed, the turn is changed in state.

Playing against another person is handled by inviting the opponent to the snapshot with the in Self built-in mechanism. Here, the default is that there is no private parts – everything is shared.

The state of the board concerning which fields can be reached from which, is kept in a double-vector in state (notice that whenever we call something a matrix,

[1] the field should have been named state prototype field to be consistent with the naming in qouridor.

it is really a double-vector, that is, a vector with vectors for entries). Each entry is an instance of a state generic field which contains pointers to the fields in each of the four neighbouring directions, or nil if there is no access. Using this, a method can be made that, given the position of a field, returns a boolean matrix having true entries in the places where a piece can be moved from that field, by just checking the neighbours.

If the pieces are placed next to one another the currently moving piece can jump over the other piece. If it is not possible to jump over the opponent in a straight line it is legal to jump to one of the other fields adjacent to the field of the opponent. These rules are implemented by asking if the pieces are on adjacent fields and then adding the relevant field around the opponent to the possible moves according to the values of the field's neighbors.

One thing to notice is that the rules don't have any explicit information about where the red or blue piece is. It's information is in terms of "where is the current player" and "where is the other player". That makes implementing the rules considerably easier.

Two rules govern the placement of walls. First off, walls cannot be placed on top of other walls neiter crossing them or overlapping them. Checking this is cheap, so we first perform a check for that. The state holds two matrices one holding the leftmost corner of the horizontal walls the other holding the topmost corner of the vertical walls. If there is a horizontal wall in the topmost-leftmost position, the horizontal-wall-matrix has its 0@0 entry set to true, and the vertical-wall-matrix is unchanged. These are used for the first validation af the placement of a wall by simply looking up the entries in the matrices corresponding to walls that would hinder the placement of the current wall.

The other rule governing the placement of walls is that the opponent cannot be hindered in being able to get to the goal: the row of fields all the way on the other side of the board. This should be done using a graph-coloring algorithm. There is so many ways of doing this, that the explanation has a section of it own.

2.3 Graphcoloring

Whenever a player tries to drop a wall at a specific position, we have to check that both players can reach the other side. This calls for graphcoloring. In the following paragraph we will discuss the best way to color the graph. Notice that a player may be prevented from reaching his own startline. Thus our graphcoloring algorithm has to be run twice. One search from the red player and one search from the blue player.

2.3.1 Brute force

One could ask oneself why we have to analyze which algorithm to use. The algorithms and the running times are thoroughly described in the literature. We

know that the running time for finding a way from s to t in a graph G with n nodes, will perform no worse than $O(n^2)$. In our particular program we know that each field is connected to no more than four other fields, can we exploit this fact? Might it be possible to use a heuristic to obtain an even better performance? Can we set up some starting conditions to perform even better? In the rest of this section we will analyze these questions.

For a specific game with n^2 fields, a naive algorithm will make $(9^2(9-1)^2) = 5184$ checks. So we notice that the naive algorithm will try almost boardwidth to the power of four checks. In a typical game with boardwidth 9 and 20 walls, this will mean doing 207360 checks. This number is calculated as 5184 checks per wall times 20 walls times 2 players. If we increase the boardwidth to 11 fields the number of checks will more than double. See table below. It is clear that a naive algorithm isn't good enough. Be aware that the second column in the table is the most important one. It tells how many checks are performed each time the player tries to put down the wall at a particular spot. This has to be sufficiently fast so the user won't experience any kind of lagging. The third column is merely there to tell how much effort is wasted throughout the game.

Naive		
Boardwidth	**Checks pr wall**	**Checks pr game**
1	0	0
2	8	160
3	72	1440
4	288	5760
5	800	16000
6	1800	36000
7	3528	70560
8	6272	125440
9	**10368**	**207360**
10	16200	324000
11	24200	484000
12	34848	696960
13	48672	973440
14	66248	1324960
15	88200	1764000

Using the fact that each field is connected to no more than four other fields will greatly improve this result. When a given field is reached there can at most be three edges out, see figure 3. A field which is at the border of the board can at most have two edges going out as the third edge is used to reach the field. There is no need to check out-edges from the final column, because if one field in the last column is reached, then there must be a route from the piece to the final column. So we end up in a running time $O(\text{boardheight} \cdot (\text{boardwidth} - 1) \cdot$

$3 - 3 \cdot \text{boardwidth} + 2 + 1) = \text{O}(\text{boardheight}^2)$. Notice the +1 which is because the field, the piece is on will have four out-edges. Also notice the -3·boardwidth, which are because fields at three of the borders only have two out-edges. The last +2 comes from the fact that the two corners in the final column has already been subtracted in the boardwidth-1 factor. So now we're looking at some much improved running times, See table below. The running time is now 9·83-3·9+3 = 192 checks compared to 5184 checks in the above example.

Naive + knowledge of out-edges		
Boardwidth	**Checks pr wall**	**Checks pr game**
1	0	0
2	6	60
3	24	240
4	54	540
5	96	960
6	150	1500
7	216	2160
8	294	2940
9	**384**	**3840**
10	486	4860
11	600	6000
12	726	7260
13	864	8640
14	1014	10140
15	1176	11760

This can be accomplished quite quickly. So we might ask ourselves, should we stop here to look for further improvements in the algorithm? From an academic point of view we really need to look further into this subject. From a practitioners point of view we might as well stop. We don't really have to ask the question anymore, do we? Read on for a more thorough analysis of the algorithm.

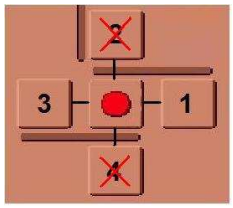


Figure 3: Only two fields are reachable.

2.3.2 Five algorithms

In the following we will discuss two common graph colouring methods, depth-first and breadth-first. Then we discuss two algorithms which exploits heuristics of the game, greedy-depth-breadth and spinning. A fifth algorithm will be presented as the one we actually implemented, reverse breadth-first.

In the first four algorithms there are worst case scenarios which are no worse than the above algorithm. We cannot do anything about these worst case scenarios and it is hard to prove that these won't come up too often. We are

contend to believe that they won't. We did manage, however, to implement reverse breadth-first such that the average case is worse than the worst case for the naive algorithm.

A typically game situation is shown in the figure 4. A worst-case (or really bad anyway) scenario is shown in figure 5. This scenario is bad because an efficient algorithm should start by going backwards. It would require a lot of practical playing with many different players to see which situations are more likely to occur. We have only been playing moderately.

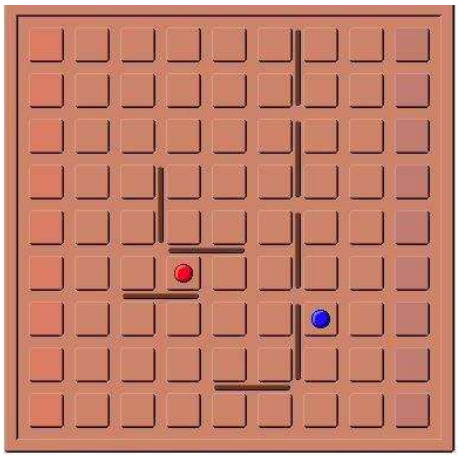


Figure 4: Typical situation

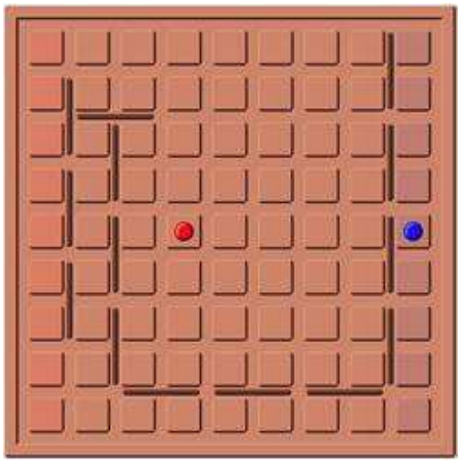


Figure 5: Bad situation. The algorithms will search through most of the fields. Except the Spinning algorithm which will find through the maze in 31 checks.

Depth-first This approach is most easily explained with some pseudo code

```
method hasRoute(field x) {
  if x in lastColumn then return true
  color(x)

  for each neighbourfield y in x which isn't colored {
    if hasRoute(y) return true
  }
  return false
}
```

How to find the neighbourfields are explained below in graph representation.

If a route is found the solution is quickly propagated to the enquiring object and it seems plausible that a lot of alternative routes doesn't need to be checked. It is hard to make an exact analysis of this, because it will depend on how the walls are distributed. It seems probable that a greedy variant of this algorithm would perform even better. By greedy we mean choosing the neighbourfield y closest to the final column. In fact we do exploit this greediness in greedy-depth-breadth.

Breadth-first Again the approach will most easily be explained by a piece of pseudo code

```
var firstField = field with piece
var queue = {firstField}

method hasRoute() {
  x = queue.pop
  if x=nil return false
  if x in lastColumn then return true
  color(x)

  for each neighbourfield y in x which isn't colored {
    queue.put(y)
  }

  hasRoute()
}
```

This solution will as always depend on how the walls are distributed. But we see that the search will fan out like rings in the water. So if the piece is far from the final column it is likely to search through as much as the naive algorithm. We mustn't neglect it's ability to overcome obstacles quickly as it searches in all directions.

Greedy-depth-breadth Pseudo-code

```
var firstField = field with piece
var queue = {firstField}

method hasRoute() {
  x = queue.pop
  if x = nil return false
  if x in lastColumn return true
  color(x)

  for each sorted neighbourfield y { // greedy part
    // There are precisely one or two fields
    // which are closer than currentField
    if there are two y's y1 and y2 {
      // make a breadth-first search around the obstacle
      queue.put(y1)
      queue.put(y2)
    } else { // continue the depth-first
      queue.put(y)
    }

    hasRoute()
  }
}
```

We'd better explain what we mean by sorted neightbourfield. Number the columns from 1 to 9 and let the final column for red player be column 9. Then we say field (i, j) is closer to the final column than field (i, k) if $j > k$. The same goes for the blue player if $j < k$. The for each statement should now look through the neighbour fields, choosing the neighbourfield closest to the final column. This will ensure a depth-first search with backtracking.

In the above algorithm we maintain a queue of which fields to visit. If the algorithm has the possibility to go up and down it makes a width-first search around the obstacle. This can be seen like a kind of simple threading. When the search meets an obstacle it spins two threads. One going over the obstacle and one going under. Both threads can only examine one neighbour before the control is given to the other thread.

Now why would we want to make such a weird search. This is based on a heuristic after having played some games. You see, the usual strategy would be to force the other player to first go one way around an obstacle and then the other way around. This often ends up in some long aisles of

walls. It is not easy to see if one way around the wall is faster than the other so we try both at the same time. What often happens though is that once we get past an aisles of walls there is a straight line to the final column, thus giving the greedy and depth-first search a chance to show their worth.

So it seems like this last algorithm exploits the strong sides of both the width-first and the depth-first search and at the same time uses greediness to quickly find straight lines if they exists. It also gives inspiration for the algorithm we will present next.

Spinning

In figure 6 we have a very bad situation for all the algorithms mentioned so far. We're examining if we can lay down the wall two below and one to the right of red.

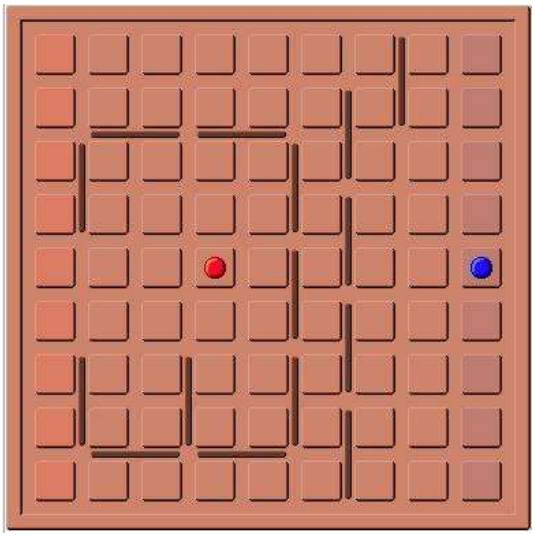


Figure 6: Bad situation for the algorithms considered so far

The depth-first algorithm might be lucky but there is no guarantee. The other algorithms will at least search all possible routes until the next to final column. In the following we will argue that we can do better. Not much, but we can give a guarantee that the algorithm will perform better than the naive algorithm. We call this algorithm the spinning algorithm.

What's so spinning about it? Well, take a look at figure 7, which is turned 90°. Now imagine the red piece as a ball falling down towards the final column. Whenever it meets a wall it will begin spinning to the right, which will make it move along the wall as shown on the figure. The ball should try to fall when the last thing it did was falling or when the last ting it did were going to the right (on the turned board).

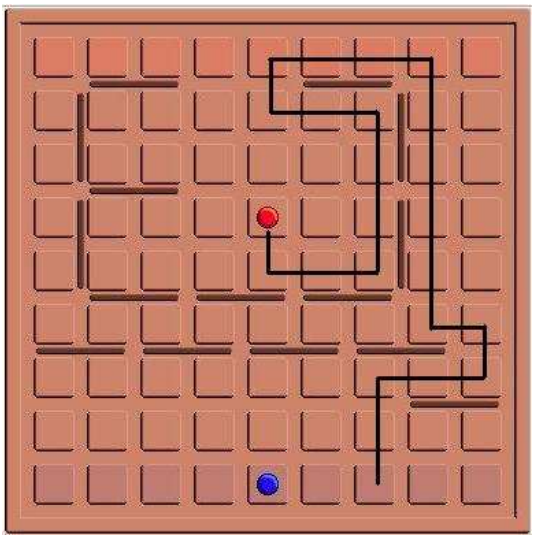


Figure 7: The power of the spinning algorithm

There is no reason why we should choose right spin in favour of left spin. In figure 8 , we fork two threads whenever it meets a wall on its way down. The average and worst case running times will be the same though. But it might be worth considering if a new heuristics is included.

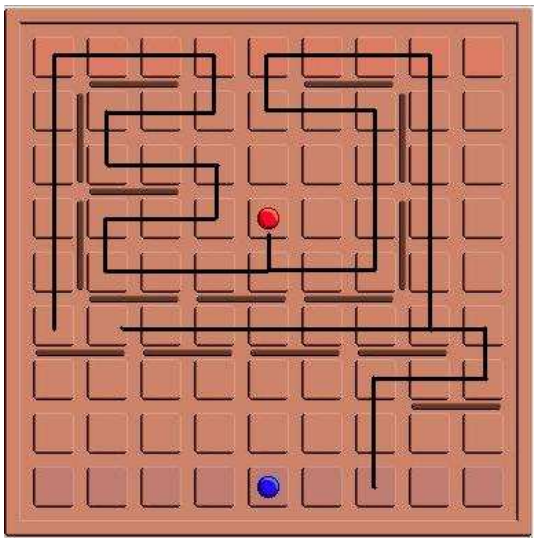


Figure 8: Forked spinning algorithm

As you can see a lot of fields are still visited, but no matter what, we will always avoid visiting them all. In fact we can give a performance guarantee. For each wall on the board we can risk visiting both sides of

the wall. That is four fields for each wall. It could also happen that we have to spin along the borders of the board. So worst case will be no worse than 3·boardwidth+4·number of walls on the board. That means a worst case scenario for a $n \times n$ board with 20 walls would be $(3 \cdot 9 + 4 \cdot 20) \cdot 2 = 214$ visited fields. Compared the table below to the above table for the Naive + knowledge of out-edges. Notice that for the small boardwidths there is not room to place 10 walls

Spinning		
Boardwidth	**Checks pr wall**	**Checks pr game**
1	86	1720
2	92	1840
3	98	1960
4	104	2080
5	110	2200
6	116	2320
7	122	2440
8	128	2560
9	**134**	**2680**
10	140	2800
11	146	2920
12	152	3040
13	158	3160
14	164	3280
15	170	3400

Reverse breadth-first From each field in the last column we make a breadth-first search to find the field, which has the relevant piece on it. So the reverse in reverse breadth-first is that we are starting from the final column instead of the field which contains the piece.

This is the algorithm we actually implemented. It's not a very good algorithm. In fact the average number of checks is 3·number of fields. Which for the 9×9 board example gives 243 checks compared to 192 in the worst case for the naive algorithm. It also has some extra overload because each time a field is visited you have to ask if it has the relevant piece on it.

2.3.3 Graph representation

Our board has the fields as a row vector with each entry a column vector consisting of fields. In the following we will discuss how to keep track of which fields are neighbours to which.

Computed edges

One way to keep track of a fields neighbours is to compute which fields are the neighbours each time we need to visit a neighbour. In our program we have chosen to represent a list of all fields as a row vector with column vectors in each entry. Each field (i, j) would have neighbours $(i-1, j)$, $(i+1, j)$, $(i, j-1)$ and $(i, j+1)$. Thinking about how many times during a game we would have to compute these neighbours it seems quite a waste. We can do better and next we will describe how to do better at the cost of some extra space.

Neighbourlists

A question quickly arises when we analyze neighbourlists. Would we want each field to know which fields it is connected to or do we want some other object(s) to keep track of these relationships. In any case the problem with neighbourlists is that it takes up some extra spaces ($4 \cdot \text{boardwidth}^2$). On the other hand you can get to each neighbour in a constant amount of time.

We wanted the state to be responsible for keeping track of the current neighbors of a field. state thus has a matrix for the state generic fields, where each field has a pointer to its (up to four) neighbors. When it is illegal to go to a neigbor because the field is on the edge of the board or there is a wall between the fields the slot is nil. The implementation of the possible moves from a field is thus conferrred to the state and the field (morph) is stripped of this knowledge. Whenever a wall is put on the board the fields in the affected area has their neighbors updated to reflect the new state of the board.

2.4 Morphs

A central part of implementing the game was making the graphical interface. The interface of the java applet version consists of several components: The board itself with the two pieces, two containers, with the walls and a turn indicator, and a control box with undo, redo and new game. This interface was based mainly on explicitly calculating which components were located at a given point on the applet's canvas. This seemed like quite a bad idea for Self – here, we could use morphs for the components.

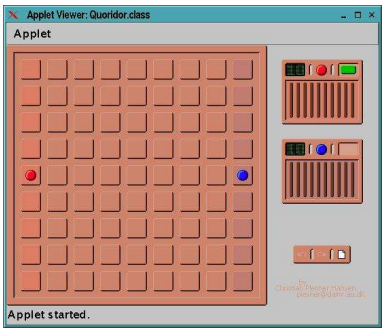


Figure 9: Quoridor applet

2.4.1 Fields

A clear candidate for being a morph is the field. Our fields have three responsibilities: handling stuff being dropped on them, positioning themselves on the board and drawing themselves.

Whenever something is being dragged over a morph, the morph has its `wantsMorph:Event` method invoked. The thing being dragged can only be dropped if this method returns true. All morphs have a `morphTypeName` slot that tells which type of morph it is, so if a morph is being dragged over a field, it can use that to say if it is interested in having that morph dropped onto it – which is only when the morph is a wall or a piece. When a piece is dropped onto a legal field, it is added to the field. The piece itself handles positioning and resizing. When a field is resized, it tells all submorphs (that is, the piece on it if there is one) to resize itself.

A field knows which board it is located on, and its own position on the board (the top-left field has position 0@0). Using this information, it can calculate which part of the board is has to occupy. This is calculated in the `update` method:

```
update = (| b. w |
  w: 2 * space.
  b: (board baseBounds size) / boardSize.
  position: (b * spot) + w.
  setWidth: (b x) - (2 * w) Height: (b y) - (2 * w).
  morphsDo: [|:m| m update].
  self
)
```

Here, `spot` is the field's position and `space` and `boardSize` are global slots defined in quoridor.

Drawing a field is pretty straightforward – it's just a rectangle with two light and two dark borders. Since we need bevels other places than in fields, this functionality is split out into a mixin bevelled. Drawing bevels require a plain, dark and light color. We need such colors so we chose to make a set of quoridor colors that were objects with `plain`, `light` and `dark` slots. The slot quoridor colors red, for instance, contained the colors needed to draw the red piece and the fields used `quoridor colors skin`.

2.4.2 Pieces

A piece is just a circle with a light and a dark arc as shadows. It is responsible for not letting itself be dropped where is shouldn't and for resizing and drawing itself.

A piece can only be dropped on a field. When a piece is picked up by the hand, it remembers which field it was on. When it is dropped somewhere, it has

its `justDroppedInto:Event:` method invoked. If it turns out that it hasn't been dropped on a legal field, it will slip back onto the field it came from. In fact we made this so strict that it is not even possible to drop the piece in the trashcan. If it is not rejected, it will tell the game's state that it has been moved.

Self has built-in operations that draw circles and arcs, so doing the actual drawing was just a matter of calling those. The piece belonging to the current player is painted solid and the other player's piece is half-transparent. That is accomplished by using `withPattern:Do:` which lets you specify an overlay the drawing is done through. One such overlay is grayMask which defines half-transparency. If someone tries to pick the piece up, it has its `leftMouseDown:` method called. If it is the piece's turn, it lets itself be picked up as usual. If not, the method returns a special dropThroughMarker that signifies that the morph is not interested in handling the event and won't let itself be picked up.

2.4.3 Board

The board is just a large flat rectangle with no special graphical characteristics. Its main responsibility is creating the fields and handling wall placement.

When a board is created, its `initialize` method will create boardSize×boardSize fields which are all assigned positions within the board and are asked to place and paint themselves. Each time the board is resized (using `setWidth:Height:`), the fields are notified and told to relocate to fit the new format. That means that the board can be resized freely and still look reasonable.

Handling walls are definitely the hardest part of the GUI, since we are excluded from using morphs here the same way as with fields and pieces. Walls are not allowed to overlap, but the places where walls could potentially be placed can. This means that we couldn't really place morphs to accept the walls the same way as with fields and pieces, since those morphs would overlap. What we chose to do was to fall back on doing what the java applet does: take raw points from the board and calculate how a wall should be placed if it was dropped there.

The functionality used in doing these calculations are used more than one place, so we chose to split them out into a mixin wallManagement. Another reason is that we were, at the time, working on two different parts of *traits board* at the same time, and splitting out one of the things made it easier to share the outliner.

The method that calculates the direction and position of a wall wasn't that complex – what was complex was constructing it and making sure that it worked. The code for determining if the wall is not in a space between two fields is this:

```
overField: p = (| fieldSize. mrg. vfield |
  mrg: margin + space.
  fieldSize: baseBounds size / boardSize.
  vfield: (((p / fieldSize) * fieldSize) + mrg) ##
          (fieldSize - (2 * mrg)).
  (vfield include: p)
```

Notice that even though it looks like ordinary integer arithmetic, this is mostly arithmetic done on points and rectangles. We vere very impressed by Self's ability to do exactly what we wanted when we used overloaded operators in arithmetic on various combinations of integers, points and rectangles.

2.4.4 Walls

Even though potential wall positions can overlap, actual walls can't, so they are just morphs. They are actually very similar to fields and include mixin bevelled. A wall has the responsibility of relocating itself to the right position and drawing itself.

The board has some functionality for deducing the position of a wall given a raw point. The wall has to go the other way: it knows where it is and has to calculate its size and coordinates on the board. This isn't much harder than the first problem, but just as uninteresting.

When a wall is being dragged, it changes orientation corresponding to the direction it would have if it were put down. This is done by the wall itself. Each time it is moved, it calculates whether it is in a space or not. If it is both in a vertical and horizontal space, it doesn't change. If it is over just one space, it changes direction by resizing and repositioning itself to make the hand point to the center of it.

When a wall is dropped, it sees if it has been dropped in a legal space. If not, it knows which container it came from and can slip back into it. If it can be placed, it is added to the board and has its `movable` slot set to false, which means that it can never be moved again.

2.4.5 Wall containers

The wall containers are just empty canvases. When they are created, they make wallCount walls and assign indexes to them. An index is just an integer from 0 to wallCount − 1 which tells the wall where to locate itself within the container. When the hand tries to pick up a wall (that is, when `leftMouseDown:Event:` is invoked), the wall sees if it belongs to the current player. If it does, it lets itself be picked up. Otherwise, it rejects by returning a dropThroughMarker which means that the container is picked up instead. This is actually a major nuisance, and given more time, we would definitely have made the walls that can't be picked up half-transparent.

3 Working with Self

As it is a new experience to work with Self we will explain about our experiences with Self in this section.

Self is an untyped language, which means that only when a method is called does the type errors show up. This gave us some work especially when writing the calculation of legal moves of pieces. The prototype State has one type of prototype Field, a point is another way of denoting a field and finally there is the prototype Field of the view. Working without types in these circumstances is frustrating, as the work usually done in the type checker has to be done by hand, and mistakes are frequent.

When programming in a typed language you will catch some of the errors when compiling; with untyped languages the finding of these errors are pushed until the relevant slots are called and even then it isn't certain that the errors show up right away. But there is a way of working around this in Self. It is possible to send a message from a given context by writing the message in an objects evaluator and pressing Do it or Get it. Thus you can invoke a message even though the other parts of the application - through which the message eventually will be send - is not coded yet. You have direct access to the objects and can modify their state by manipulation thus it is easy to set up a simple test. The work on a test is not reusable though.

Now you just have to make a test each time you would compile in a language like Java or Beta. Of course this will not be done, but the possibility for debugging is still an asset. In a way the information you have access to in Self is more or less the information that would be displayed in a debugger for another language.

3.1 Custom "keywords"

In Self, you can easily make new objects that have slots taking care of specific behavior. Of course this will not really be keywords as their use in other contexts is not prohibited. In two cases we have made new "keywords":

`if:Then:Else:` When you are used to the `if (...) {...} else {...}` control structure of eg. Java and C it is awkward to use the `... ifTrue:[...] False:[...]` of Self. We therefore made an `if:Then:Else` slot which was put in *control*, a parent of the *lobby*, see fig 10. We later included `if:Then:`, `if:Then:ElseIf:Then:Else:` and `unless:Do:` slots.

One disadvantage of self's conditional syntax is, that you have to read past the condition of the expression to the `ifTrue:` in order to discover that what you're reading is a conditional expression. Having the first thing in the expression be `if:` makes the code easier to read.

`new` It is a bother to copy the relevant prototype and initializing it with the relevant parameteres every time you want a new object. That's why the object new was inserted in the parent *keyword* of quoridor namespace, see fig. 11. A copy of one of the prototypes in quoridor can now be made by typing `new` *name of prototype*, followed by an argument if this is necessary.

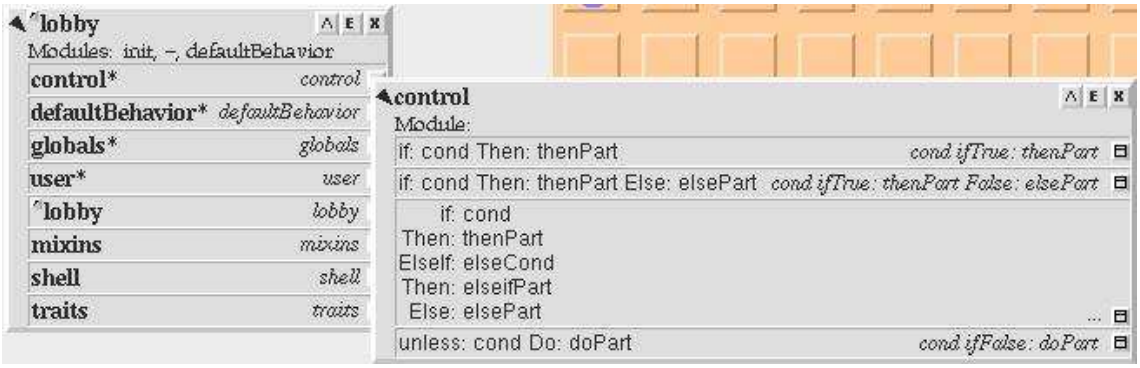


Figure 10: New control structure

The structure has to be maintained manually, by adding new slots to the new object.

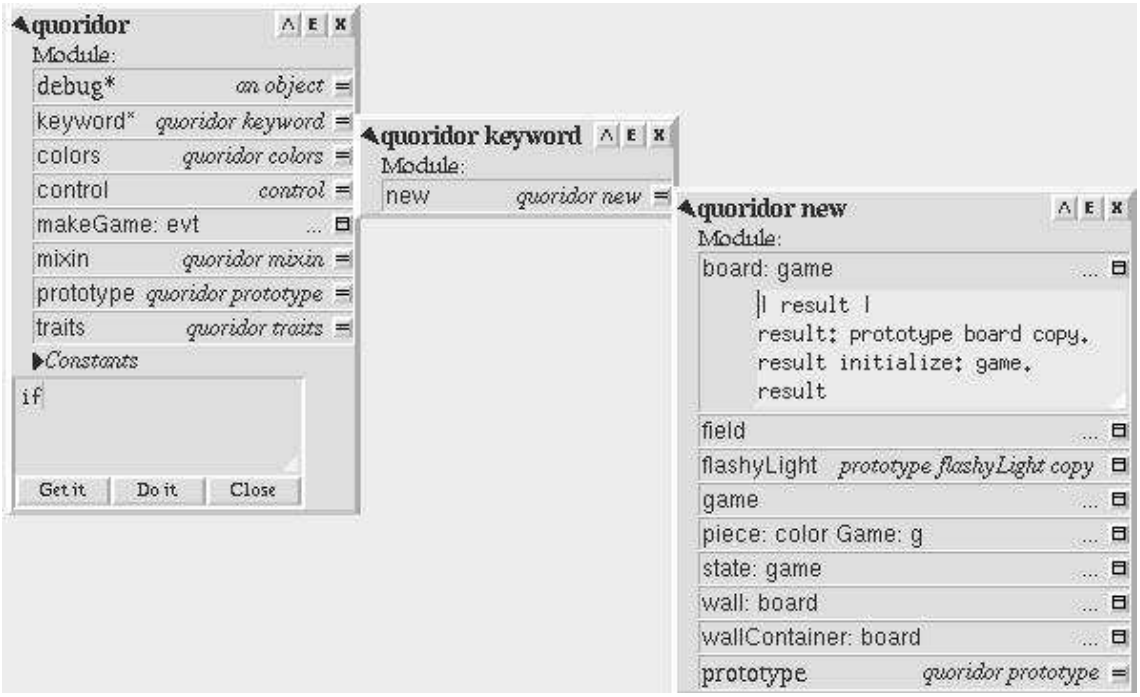


Figure 11: New "keyword"

Being able to introduce keyword-like slots gives the language great flexibility, syntaxwise. Using this feature, we were able to alter the syntax into something that we thought was considerably easier to read than "native" self.

3.2 Cooperative work

All in all it was a great experience working together on the same snapshot. The nice features and the drawbacks are pointed out in this section.

It did not seem to slow down self, that two people shared the same snapshot. There were no locks - thus you could imagine that working on the same object would lead to conflicts, but this was not the case. As you can see the interactions of the other user, you know what is being worked on. It is not possible to have two outliners for the same object and thus it is not possible for the other user to work on the same slot without your awareness.

The impossibility of having two versions of the same outliner makes it necessary for the two users to share a part of their workspace such that the common objects can be put in this part of the workspace. This can give space problems. We did not have to work all three of us, at the same time, the problem of the shared parts of the workspaces would be even greater with more participants in the cooperation.

In the beginning we placed the two workspaces side by side, but this was to confusing. When the rightmost developer requested to see a new object its outliner would fly across the screen of the leftmost developer. The same problem is there when an object is dismissed using the animated dismiss, putting the dismissed object in the paperbasket. We used Christian Plesner's solution to the dismiss problem. These annoyances is diminished but not eradicated by placing the workspaces on top of one another.

When you choose an entry in a menu using one of the mousebuttons, the last used entry will be the default next time you use the menu. That is it will be the last entry *any* user of the snapshot used. This means that you have to check the default anyway.

As Self is so easily modified the above mentioned things would be candidates for changes, if you should cooperate in Self on a regular basis.

Its it annoying for the user who invites another user to a snapshot that it is the host who gets all the errormessages and will have to place them before the guest can have a closer look on the errormessage if it is something about the stuff the guest is working on. It can also be a good thing though; when the guest works on a very hard part the host can get rid of the errormessages popping up leaving the guest to work on the problem.

Another problem is that Self crashes very often - especially when you work with Morphs. This is because the slots of Morphs are called by application external objects, this means that the slots you are working on will be called as well if they are part of the things that should be done when the morph is drawn. Thus it will crash twice as often when two persons are working at the same time, sometimes barely leaving you the time to get anything done in between the restarts. It seems as if several often occuring errors are propagated right through to the user.

4 Conclusion

The goal of this project was to implement the game of quoridor. We have accomplished that. The game can be played, enforces all the rules, and has a minimum of erratic behaviour. We did experience some unexplained problems (walls disappearing, for instance) but they seem to have gone away. The game can be played over the net, but purely through self's mechanism for cooperative working.

As mentioned in the introduction, a snapshot can be found in

`~kari/AOOP/Self/objects/quoridor.snap`.

A new game can be started by pushing the "New game" button. All the code we've written is available through the slots of the quoridor object.