The internals of my Twixt-programs

Table of contents

1 Introduction....2
2 Representation of the board....2
3 Behaviour....4
3.1 The static evaluation function....5
3.2 Move Generation....10
4 Literature....12

Copyright © 2007 Johannes Schwagereit All rights reserved.

1. Introduction

The following text gives a short overview how my Twixt-programs 'T1' and 'T1j' work internally. Of course, the details are simplified.
Any feedback is appreciated.

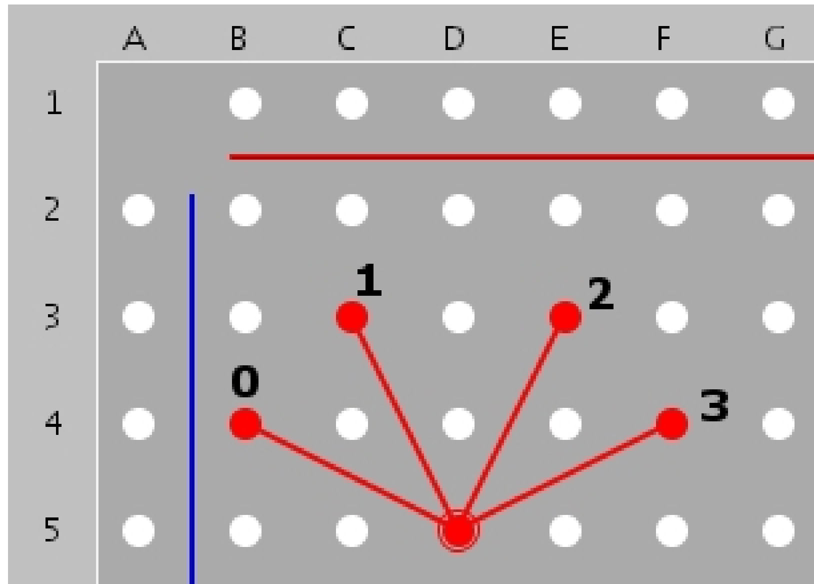
2. Representation of the board

One of the first decisions when starting to write a Twixt-program is to find a representation of the board, i.e., how to store the pegs and links on the board. The data structure should allow efficient and reliable updates (including removal of pegs and links) and queries (e.g. to check if two pins can be connected).
In my Twixt-programs the board is represented as a two-dimensional array of "Nodes".
A Node is a simple structure, in T1j (in Java) it is represented as an inner class:

```
static final class Node
{
   private int value; //0 or XPLAYER or YPLAYER
   private final int[] bridge = new int[4];
}
```

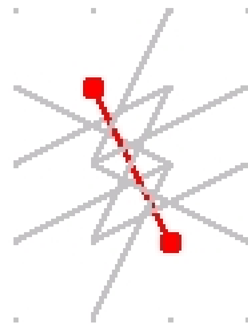
The 'value' is either 0 for an empty hole, or +1 or -1 for a pin of one colour. The player playing from left to right is called XPLAYER (value = +1), the top-down player is the YPLAYER (value = -1)
The 'bridge' is a 4-tupel, representing the four possible connections going upwards (north). So every connections is only stored once, not twice at both ends.

Copyright © 2007 Johannes Schwagereit All rights reserved.



bridge[0] connects D5 with B4, bridge[2] connects D5 with E3.
To check the connection between D5 and B6, bridge[3] of B6 is relevant.

The value of a bridge is "10", if there is an actual link between the two pegs. Otherwise the value indicates the number of *crossing links*.
In the example above, D5.bridge[1] has of course the value 10.
E5.bridge[1]has the value 2, because two other links cross the connection to D3. E5.bridge[2] has value 1.
If a link is set, the 'own' bridge is set to 10. All nine bridges with are crossed by the new link, are increased by one. And, vice versa, if a link is removed, the own bridge is set to 0 again, the crossed bridges are decreased by one.



The picture shows the nine bridges which are crossed by a new link.
This representation has a nice advantage. It is easy to check if a link can be set: Only if the

Copyright © 2007 Johannes Schwagereit All rights reserved.

value of this bridge is zero, the link is allowed. This check is of course easy and fast. This is the (slightly modified) method in T1j ('direction' is the number of the bridge):

```
public boolean linkAllowed(final int x, final int y, final int direction)
{
   return field[x][y].bridge[direction] == 0;
}
```

Just a bit more complex is the method to check if a link is allowed between two given pins without knowing the number of the bridge:

```
public boolean isLinkAllowed(final int xa, final int ya, final int xb,
final int yb)
{
   if (Math.abs(xa - xb) >= 3 || Math.abs(ya - yb) >= 3
         || Math.abs(xa - xb) + Math.abs(ya - yb) != 3)
   {
      throw new IllegalArgumentException("Wrong distance");
   }
   //put lower first
   if (ya < yb)
   {
      return linkAllowed(xb, yb, (xa < xb) ? xa - xb + 2 : xa - xb + 1);
   } else
   {
      return linkAllowed(xa, ya, (xb < xa) ? xb - xa + 2 : xb - xa + 1);
   }
}
```

A few words about the board itself:

- The (0,0)-coordinate is the upper left corner of the board, (23,23) is the bottom right corner, (23,0) is upper right.
- The actual board is a bit larger than required, it has a margin of 3 on each side, to make the check for legal moves easier, and to prevent "array out of bounds"-errors.
- In T1j (not in T1) the board is stored twice, the second board is mirrored on the diagonal (x=y)-Axis. This is an idea of A.Braendle, it makes the implementation of the game logic easier.

3. Behaviour

T1 and T1j use the classic technique for game-playing programs: MiniMax with AlphaBeta-pruning. There might be other approaches like a rule-based or genetic algorithm, but this was never investigated by me.
My programs also uses some modifications of AlphaBeta, like 'iterative deeping' and 'killer-move heuristics'. Patterns are used to find promising moves.

Copyright © 2007 Johannes Schwagereit All rights reserved.

The two main challenges when writing TwixT-playing programs are the evaluation function, and to find promising moves to investigate. These aspects are discussed below. Both required some consideration before starting to program.

To quote [RIKN], a book on Artificial Intelligence:

> "[There are] two important knowledge-based components of a good game-playing program: a good plausible-move generator and a good static evaluation function. They must both incorporate a great deal of knowledge about the particular game being played."

3.1. The static evaluation function

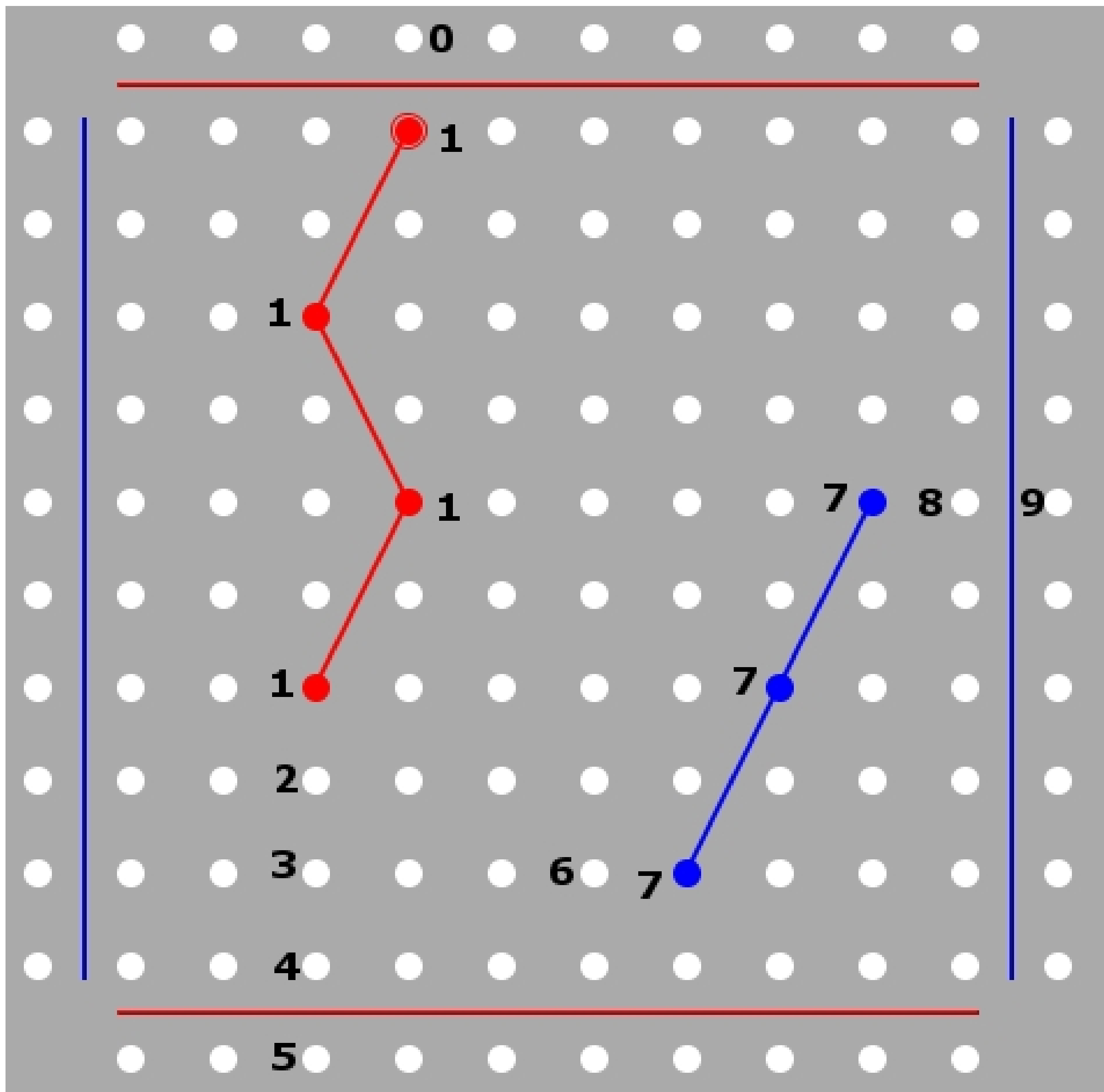
A static evaluation function is needed because the search-tree in any Twixt-program has a limited depth (perhaps 5). So at the leafs of the game-tree a function is needed to assign a value to the current situation.
Again a quote from [RIKN]:

> "... the resulting board positions must compared to discover which is most advantageous. This is done using a *static evaluation function*, which uses whatever information is has to evaluate individual board position by estimating how likely they are to lead eventually to a win."

In my Twixt-programs, the basic idea is quite simple: The value of a situation is the number of rows or columns which are missing to complete a path between the two sides. To be more precise: The value is the difference between these values for both players.

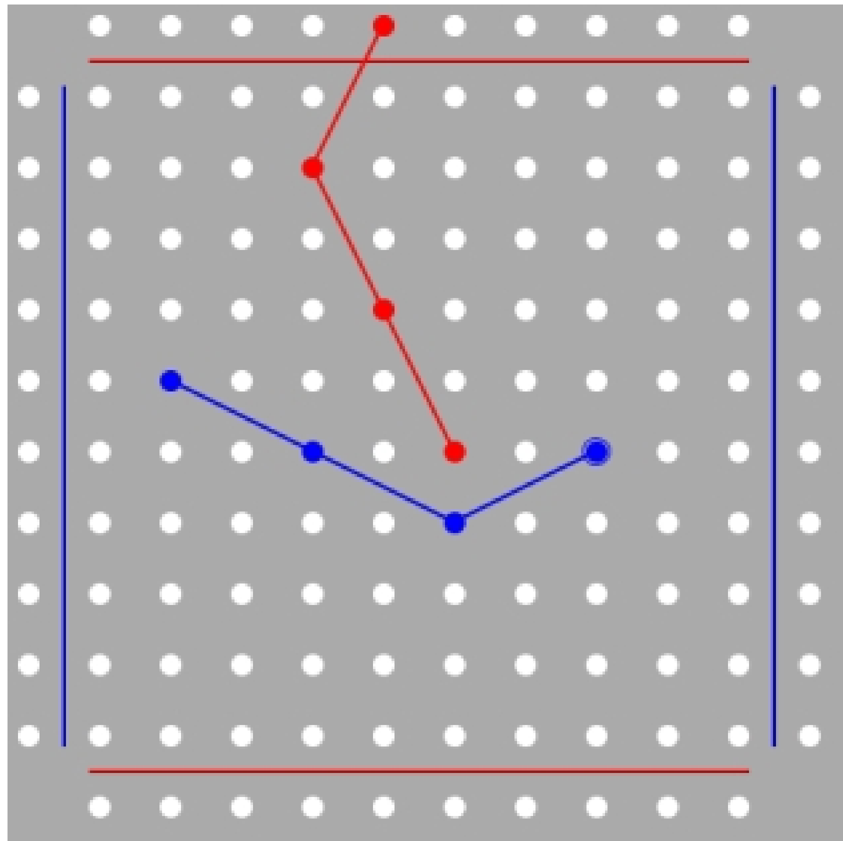
An example to make this clearer:

Copyright © 2007 Johannes Schwagereit All rights reserved.



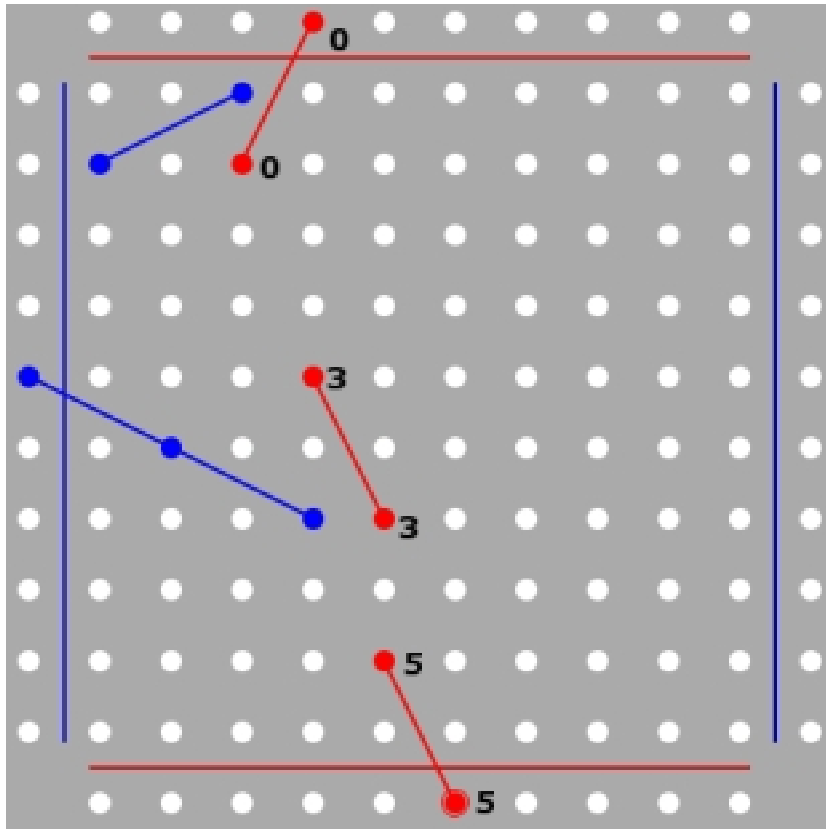
The red player has 5 rows to cross to complete his path from north to south. For the blue player this distance (from east to west) is 9. So the value of this situation is 9 - 5 = 4 (the higher the better for the red player).
This simple example does not show the difficulties in more complex situations. Above all, the distance to cross is not always a straight line.

Copyright © 2007 Johannes Schwagereit All rights reserved.



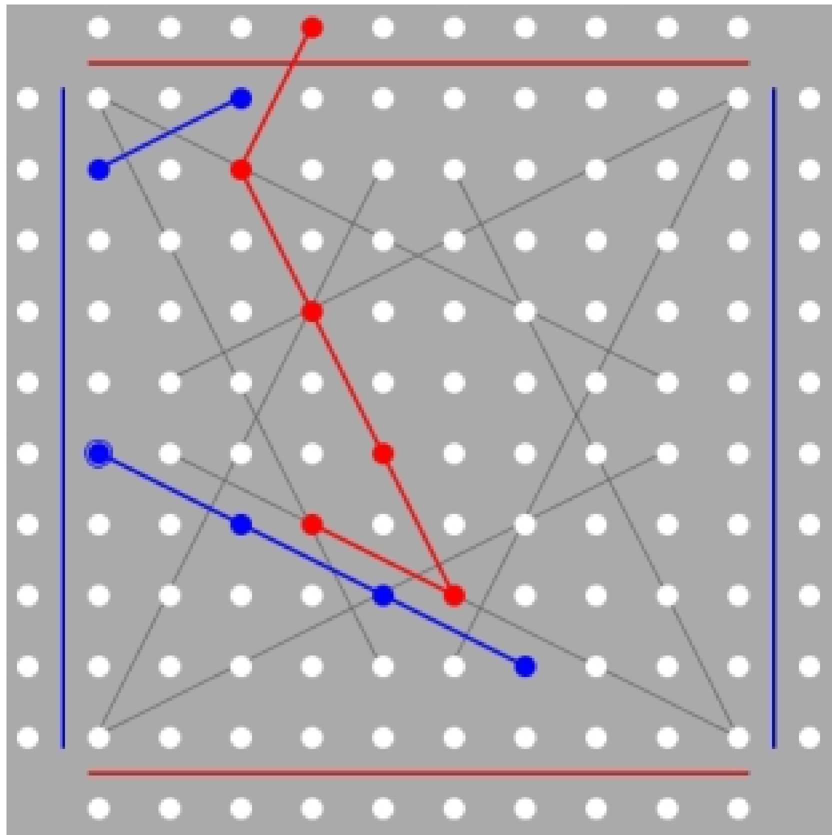
In this example the red player is blocked - his graph of connected pins is worthless.

Copyright © 2007 Johannes Schwagereit All rights reserved.



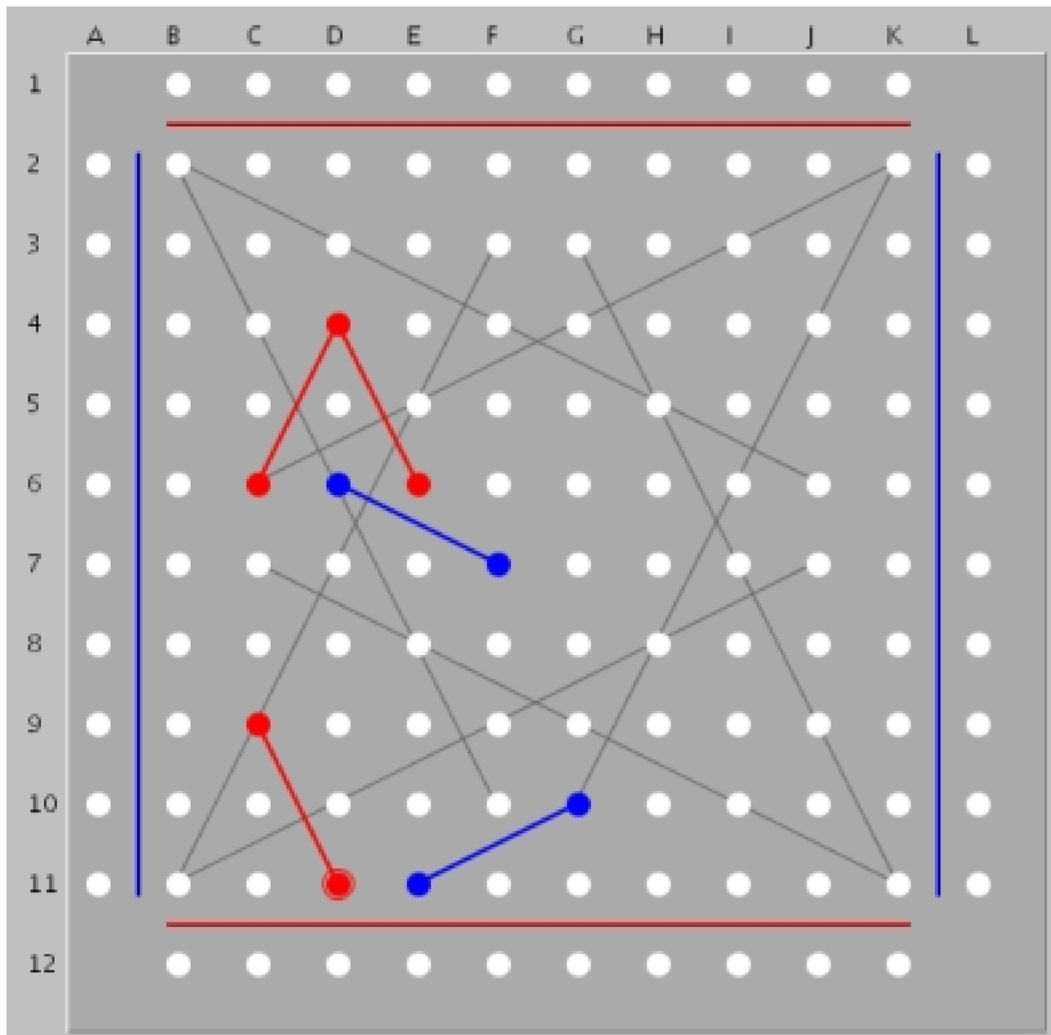
The "path" of the red player consists of three seperate graphs, the evaluation counts the gaps between them.

Copyright © 2007 Johannes Schwagereit All rights reserved.



Red is *not* blocked - if he has the next turn, the value of his pins is 3. But if Blue has the next turn, the red pins are useless. (T1j knows this difference, but T1 does not.)
The (simplified) algorithm to calculate the value of a position is to iterate over all holes of the board, row for row, from North to South. The holes in the first row (above the red line) get the value zero. Any hole in the following rows get the value of the hole above plus one. If this hole is filled with an own peg, all connected pegs get the same value. If a hole is filled with an opposing peg, it (and the holes below) get the value 99.
The value of a position is the minimum of all holes in the last row.

Copyright © 2007 Johannes Schwagereit All rights reserved.



The red upper graph (D4, C6, E6) has the value 3, the lower one (C9, D11) has the value 6, so the red pins have value 7. (Of course, if blue plays the next pin, B7 would block red.)

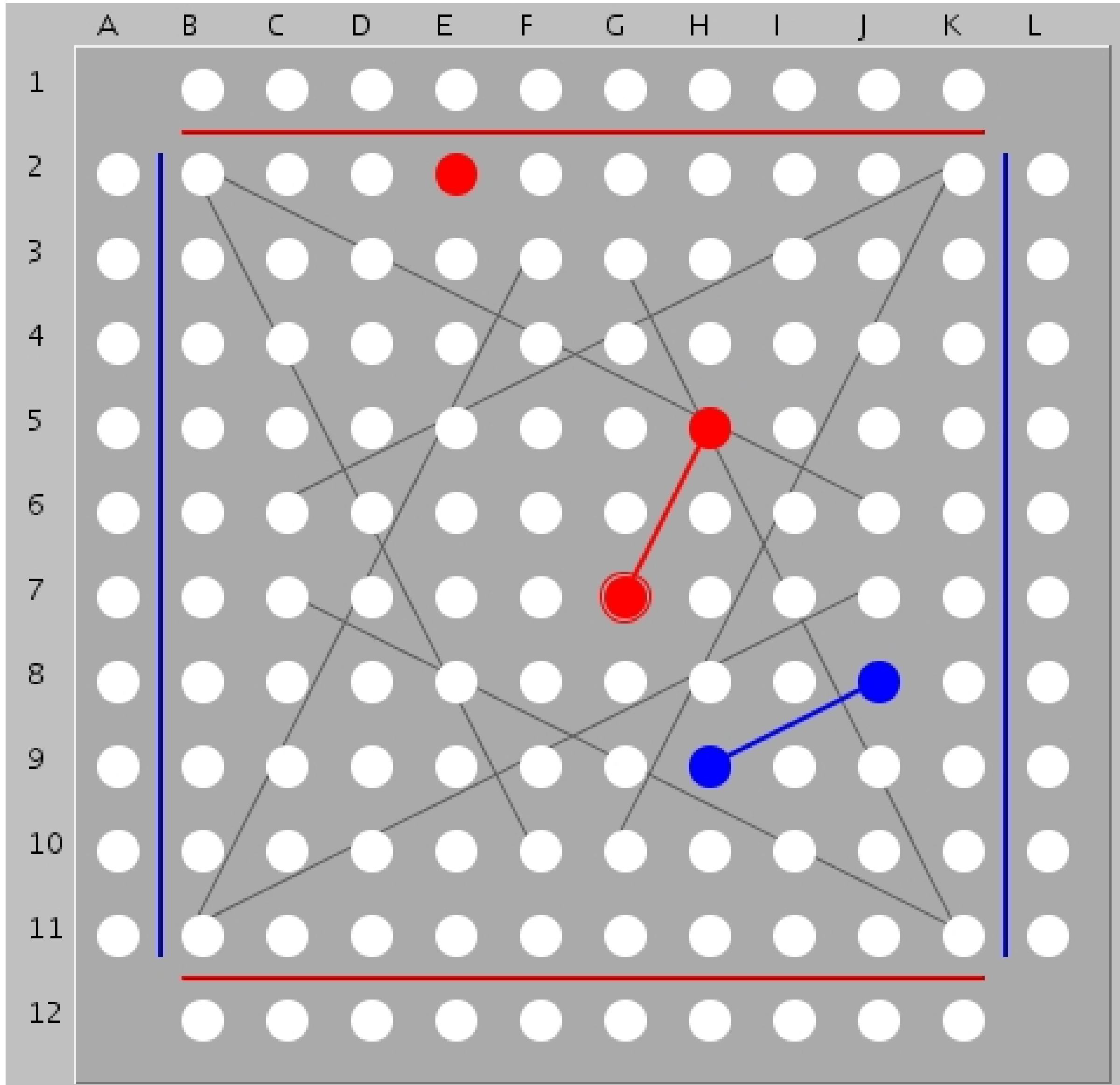
3.2. Move Generation

As said before, not all possible moves are considered for the Gametree, but only promising moves.
Moves are selected using a simple pattern matching algorithm.

The ends of the graphs which are used in the evaluation function are the Reference points for any move-selection. All moves considered are relative to these reference points.

Copyright © 2007 Johannes Schwagereit All rights reserved.

An example:



The reference points for red: H5-North, G7-South. E2 is not on the shortest path for red. For blue: H9-West and J8-East.

T1 and T1j both use 'offensive' and 'defensive' patterns. An 'offensive' pattern is used to find promising moves relative to own pegs, an 'defensive' pattern tries to stop the opponent.

Copyright © 2007 Johannes Schwagereit All rights reserved.

In our example, if red has the next move, the program would check the 'offensive' patterns for H5 playing North, and for G7 playing South.

Patterns are stored in a text-file using a simple language. An example for an 'offensive' pattern is:

```
Off 2
   Own 3 3
   Set 2 1
   BPoss 0 0 2 1
   BPoss 3 3 2 1
```

This can be translated as:

- This is offensive pattern number 2.
- An own peg is required at position (3,3) relative to the reference point. *In our example 'H5 playing North' this is E2.*
- The new tag is set at (2,1) if possible - every pattern contains exactly one 'Set'-command.*This is F4 in our example.*
- A bridge has to be possible from (0,0) to (2,1). *Our Example: H5-F4.*
- A second bridge has to be possible from (3,3) to (2,1).

Only if all conditions of the pattern are true, the peg specified by the 'set'-command is added to the list of promising moves. *Of course, other moves like G3 or I3 would be found by other patterns.*
All pattern are used twice, as described in the file, and as their mirrored counterpart.

4. Literature

[RIKN]
"Artificial Intelligence, 2nd edition" by Elaine Rich and Kevin Knight, McGraw-Hill, 1991.

Copyright © 2007 Johannes Schwagereit All rights reserved.