Exploring Board Game Strategies

A Recreational Application of GUIKit

Yves Papegay

The programming paradigms available in *Mathematica* together with the level of generality that can be obtained through its symbolic capabilities can be applied to a wide range of programming situations, particularly the rapid prototyping of applications. This article shows our use of this outstanding computational environment to develop playable prototypes of board games and explore game strategies. *Mathematica* features allow some general ad hoc design patterns for such games to be expressed, applied, and further refined. We show the benefits of such an approach—lots of code reuse, clearer design and programming, and ease of experimentation—by exhibiting our work on three different board games. We further show how the statistical and mathematical techniques available with *Mathematica* can be used to generate reports on the playability of a particular game as well as to develop winning strategies. We also show how *J/Link* and *GUIKit* can help refine and improve the interface of a game.

■ Introduction

Chess, backgammon, Abalone, Go, Tic-Tac-Toe, Mine Sweeper, Tetris, and many others are well-known examples in the large collection of so-called board games [1]. Board games provide many interesting topics of study for several academic disciplines, which have their own scientific societies and journals [2]. Board games are also very popular among freeware and shareware users. There are many ways to play against a machine, running the code online as an applet or servlet, or on a personal computer, a PDA, or even a mobile phone.

Developing a computerized version of a board game is an exciting challenge and a very good exercise for computer science students. From the theoretical point of view, the implementation of the evolution of a game usually involves a wide range of classical algorithms as well as implementation of the strategies played by the computer. It is also important to keep in mind that the fun derived from playing a game often critically relies on a nicely designed and efficient user interface.

Incorporating different kinds of programming paradigms, *Mathematica* [3] embeds a highly flexible and intuitive language for prototyping applications.

Using a computational environment such as a development platform allows for a generic and high-level representation of the board and its evolution. It also makes development and testing of board game strategies easier.

The notebook environment is powerful enough to set up the interface between the game and the player and offers a good visualization of the board. However, even with careful and sophisticated development the graphics are not fancy enough to be addictive to players. Fortunately, with the help of Java toolkits, namely *J/Link* and *GUIKit*, the functionalities of Abstract Window Toolkit (AWT) and Swing [4] are within reach and permit the dream of an interface being developed with only a small effort.

In the first part of this article, we describe a generic design pattern for the implementation of board games. Actually such a game is uniquely and completely defined by a quite small set of information concerning the representation of the board, the playing rules, the initialization phase, and the termination phase. From this set of information and with the help of the generic design pattern, we can quickly develop a playable and smart implementation of the game.

Examples of such developments are given in the second part of the article where we describe in detail the full playable implementation of

1. HMaki—a Japanese puzzle game (a.k.a. Same Gnome by Linux users)
2. a classical lines game
3. Mancala—a well-known African game (a.k.a. Awale)

For each of these examples, we also provide a tutorial approach to the development of a fancy graphical user interface using Java technology with the help of *J/Link* and *GUIKit*.

The last part of the article is devoted to the exploration of strategies. One benefit of developing our own implementation of a board game with *Mathematica* is that it yields perfect control over the different steps of each game, the ability to unplay or to replay some or all of those steps, to add or remove randomness, and to add or remove interactivity during the game. *Mathematica* also makes it easier to implement and apply strategies and perform, on a large set of examples, statistical analysis on the resulting games.

■ A Design Pattern for Board Games

"So, in a broad sense, a board game is any that can be played on a flat surface such as a table or floor." David Parlett, *Oxford History of Board Games*

As quoted, the most important point when designing a board game implementation is to have a proper idea of the *board* itself, which represents the complete status of the *game* at each *play* and the *rules* that determine whether or not a play is legal.

In our design, we should keep in mind that we want the computer to analyze several games, to replay some of them, and to test some strategies.

□ Some Semantics

Before going further, let us explain in detail what is exactly meant by game, board, play, and rules.

Game

The game denotes a complete set of interactions between the program—or the physical support of the game, whatever it is—and the player. Hence, it is a succession of three phases:

- an *initialization* phase when the game starts or restarts
- a *playing* phase (i.e., when the user is playing). This is the most common behavior of the game and consists in a sequence of successive plays.
- a *termination* phase at the end of the game

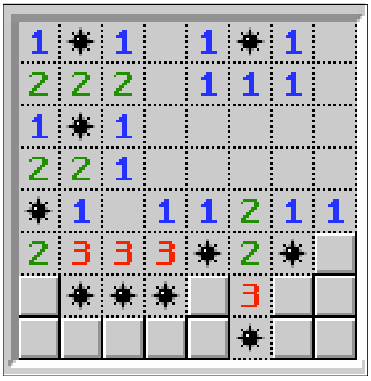


Figure 1. Windows' famous Minesweeper board.

In the case of the well-known Minesweeper game (Figure 1), the initialization phase locates the mines on the board and hides all the locations; the playing phase consists, for the player, of de-mining by selecting one location at each play and getting back the information on the adjacent locations. The termination phase is what happens when a mine explodes or when the player discovers the last mine.

Board

The board represents not only the physical board but, by extension, the complete status of the game at a given time. By definition, in a board game, this status is well defined by a mapping between a two-dimensional set of locations and additional information (usually qualitative or discrete) for each location.

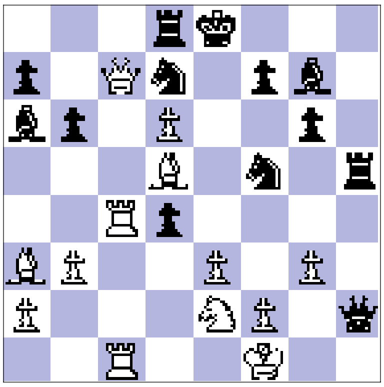


Figure 2. A chess board.

In the case of chess (Figure 2), the physical board is simply made of 64 squares, given by their line and column references. We also consider as part of the board the information, for each square, of the color and kind of piece (if any) occupying it.

Play

By definition, a play is one step of the playing phase. At each play, the player has to select which action, among the legal (*valid*) ones, to perform. In most of the games, such an action is completely defined just by selecting one or two locations on the board. In other games it may involve some random process represented by additional parameters, such as the result of dice throws in a backgammon game (Figure 3).

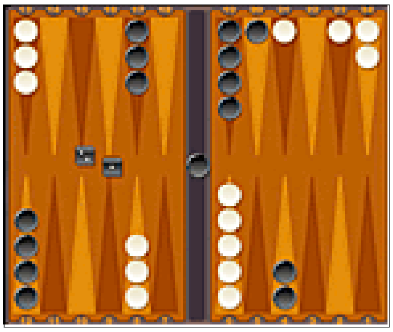


Figure 3. A backgammon board.

The information contained on the board and given by the player should be sufficient to decide whether the selected play is valid with respect to the rules of the game.

To execute the play, the computer has to ensure that the play is valid and then modify the status of the game.

If the computer is also supposed to act as a player, it must then analyze the game, select which action is the better one, and play it. Its doing so is another play, however!

Rules

The rules are the set of constraints that determine what can be played and how the status of the game should be modified by a play.

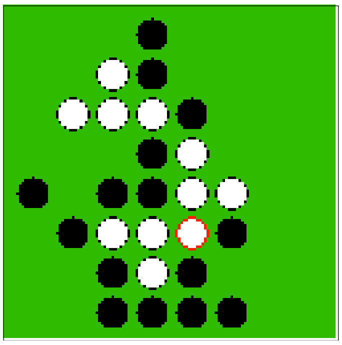


Figure 4. An Othello board before black's play.

In Othello (Figure 4), the rules state that to put a black piece somewhere it should enclose white pieces. The rules could consist of:

- A new piece should be contiguous to a piece of the other color.
- There must exist a segment (horizontal, vertical, or diagonal) starting at the new piece and ending at a piece with the same color containing only pieces of the other color.

So, if we denote the rows from top to bottom by A, B, C, … and the columns from left to right by 1, 2, 3, … , black can only play at B2, C1, C6, D1, D2, D6, D7, and E7.

The rules should also state that enclosed pieces will change color after the play. For example, if black is played on D6, then the white pieces in D5, E5, E6, and F4 will be turned to black.

□ Designing the Board

What differs from one game to another concerning the board is its size, in rows and columns, and the possible values (*patterns*) for each of the locations of the board.

For simplicity and readability of the code in this article, we decided to make extensive use of global variables, rather than providing a lot of arguments to functions. In a less experimental implementation, we would have used packages and contexts to avoid occasional shadowing of symbols. Here, we use three variables (Width, Height, and Patterns) to describe the configuration of the board and an additional one (Board) to store the board itself. To give a better picture, we use a chess game, as an example.

```
Off[General::"spell"]
Off[General::"spell1"]
NBOptions = {WindowFrame → "Palette", WindowElements → {},
  WindowFrameElements → "CloseBox", WindowClickSelect → False,
  Active → True, CellOpen → True, ShowCellBracket → False,
  Selectable → False, ShowCellLabel → False, ShowCellTags → True};
```

```
Width = 8;
Height = 8;
Patterns = Prepend[
  Flatten[Outer[List, {"B", "W"}, {"R", "Kn", "B", "Q", "Kg", "P"}], 1],
  "empty"]
```

```
{empty, {B, R}, {B, Kn}, {B, B}, {B, Q}, {B, Kg},
 {B, P}, {W, R}, {W, Kn}, {W, B}, {W, Q}, {W, Kg}, {W, P}}
```

In most of the cases, the board can just be described by a list of lists of integers and the definition of a mapping between integers and patterns—giving the patterns as a sorted list is sufficient for that. The computation time of the algorithms involved in validating plays and modifying the board is usually negligible due to its size, and it is not necessary to further optimize the representation of the board.

Initialization

The board is computed for the first time during the initialization phase by the generic NewGame function.

When no randomness is involved in the game (e.g., in chess or Othello), a configuration flag, NeedRandomness, is set to False, and the NewGame function only initializes the board and possibly performs an initial play.

```
NewGame[] /; Not[NeedRandomness] := (Board = InitBoard[]; InitPlay[];)
```

When some randomness is necessary, we will provide a seed as an argument to control the randomness and to be able to replay the same game. In this case, the seed of the pseudorandom generator is stored in a variable (Seed) to allow replaying the same game—when it is possible. Without an argument, the last value of Seed is used as a seed.

```
NewGame[s_: Seed] :=
 (Seed = s; SeedRandom[s]; Board = InitBoard[]; InitPlay[];)
```

With the string "new" as the argument, a randomly generated seed is used and then stored.

```
NewGame["new"] := (SeedRandom[]; Seed = Random[Integer, 31991];
  SeedRandom[Seed]; Board = InitBoard[]; InitPlay[];)
```

This ability to replay a game is a valuable feature to players who want to exercise their skills and improve their strategies.

InitBoard is another generic function that builds the board and returns the corresponding matrix as a list of lists. It is based on InitPosition, whose definition is unique and characteristic of each board game.

```
InitBoard[] := Table[InitPosition[i, j], {i, Height}, {j, Width}]
```

In the example of the chess game, the initial positions are as follows.

```
InitPosition[1, c_] := {2, 3, 4, 5, 6, 4, 3, 2}[[c]]
InitPosition[2, c_] := 7
InitPosition[7, c_] := 13
InitPosition[8, c_] := InitPosition[1, 9 - c] + 6
InitPosition[l_, c_] := 1
```

This produces the usual representation of the chess board at the beginning of the game.

```
InitBoard[] /. x_Integer :> Patterns[[x]] // MatrixForm
```

```
⎛{B, R}  {B, Kn}  {B, B}  {B, Q}   {B, Kg}  {B, B}  {B, Kn}  {B, R}⎞
⎜{B, P}  {B, P}   {B, P}  {B, P}   {B, P}   {B, P}  {B, P}   {B, P}⎟
⎜empty   empty    empty   empty    empty    empty   empty    empty ⎟
⎜empty   empty    empty   empty    empty    empty   empty    empty ⎟
⎜empty   empty    empty   empty    empty    empty   empty    empty ⎟
⎜empty   empty    empty   empty    empty    empty   empty    empty ⎟
⎜{W, P}  {W, P}   {W, P}  {W, P}   {W, P}   {W, P}  {W, P}   {W, P}⎟
⎝{W, R}  {W, Kn}  {W, B}  {W, Kg}  {W, Q}   {W, B}  {W, Kn}  {W, R}⎠
```

□ Implementing the Transition Function

For most of the board games, playing is just doing one of these three kinds of actions:

- removing a piece from the board
- adding a piece to the board
- moving a piece

So, from an implementation point of view, playing amounts simply to changing the status of one (or two) position(s) of the board and can be completely defined by giving the corresponding location(s) on the matrix as arguments to the transition function. A few games, such as backgammon, may let the player move several pieces during the same play. These cases require further refinement of the function.

Of course, only some positions are playable and, for a selected piece, only some motions are valid with respect to the rules of the game. Those constraints require us to check the arguments of the transition function before its application.

Hence, the generic transition function PlayThere relies on two nongeneric functions: IsPlayable, which performs the checking, and PlayBoard, which changes the board.

```
PlayThere[p : {__Integer}] :=
 Module[{m = IsPlayable[p]}, If[Not[FalseQ[m]], PlayBoard[p, m]]]
PlayThere[{p1_List, p2_List}] := Module[{m = IsPlayable[p1, p2]},
  If[Not[FalseQ[m]], PlayBoard[p1, p2, m]]]
```

`FalseQ` is analogous to the standard `TrueQ` function and returns `True` only when its argument is `False`.

```
FalseQ[p_] := TrueQ[Not[p]]
```

Many algorithms and computations can be invoked by `IsPlayable` depending on the complexity of the rules of the games. Usually, however, this computation will also provide information on how the board will change due to the play. For example, in chess, if you plan a play to capture a piece, checking this plan will produce the capture as a side effect. So, the results computed by `IsPlayable` usually affect `PlayBoard`. In the implementation, the value returned by the first function is given as an argument to the second.

□ Main Loop and Interaction with the Player

The main loop defines the game as a sequence of plays. It may be used for an interactive game and for batch games for analysis purposes. The generic function `PlayGame` implements this simple loop. We will show how to implement interaction through a notebook graphical interface.

Interaction with the player is made through `View`—for visualization—and through `GetPlay`. `GetPlay` may be used to apply a strategy in a batch game or to prompt the user for the selected play in an interactive game.

```
PlayGame[x___] := (NewGame[x];
  While[Not[GameOver[]], (View[]; PlayThere[GetPlay[]])]; EndGame[])
```

The termination phase of the game is performed by `EndGame`, which can be refined for a particular game but has this simple default value.

```
EndGame[] := View[]
```

□ Notebook Visualization

A *Mathematica* notebook, or a palette, can be used for a first approach to graphical visualization and a graphical user interface for a game. Here, we represent the matrix of the board with an array of buttons with text (or a color) for the pattern corresponding to the location. Each of the buttons invokes a transition function—defined in terms of `PlayThere`—with its location as the argument. Control of the game is made directly through the calls and no loop is necessary.

The generic function for playing a new game is `NBPlayGame`, which is analogous to `PlayGame`.

```
NBPlayGame[x___] := (NewGame[x]; Nbview = NotebookPut[NBView[]];)
```

`NBPlayGame` displays the notebook created by `NBView`.

```
NBView[] :=
 Notebook[{Cell[BoxData[NBBoard[Board]], CellTags → {"board"}]},
  Apply[Sequence, NBOptions]]
```

Inside `NBView`, `NBBoard` generically creates the array of buttons.

```
NBBoard[l_] := GridBox[MapIndexed[NBMakeButton, l, {2}],
  RowSpacings → 0, ColumnSpacings → 0]
```

The only nongeneric function to develop when implementing a game is `NBMake`⋱ `Button`. It has to take in account the transition of the board at each play, manage the end of the game, and refresh the display. For those purposes, `NBMakeButton` can use the generic `NBPlayThere` and `NBRefresh`. See the following examples for details of implementation.

```
NBPlayThere[p_] := (PlayThere[p]; NBRefresh[];)
```

```
NBRefresh[] := (NotebookPut[NBView[], Nbview];)
```

■ The HMaki Example

"If you start this game, you see a board full of tiles. Your task is to remove as many tiles as possible from the board. You cannot remove single tiles, instead you have to remove them in groups of adjacent tiles filled with the same color." Holger Klawitter, *HMaki 3.9.1 Information* (www.klawitter.de/palm/hmaki.html)

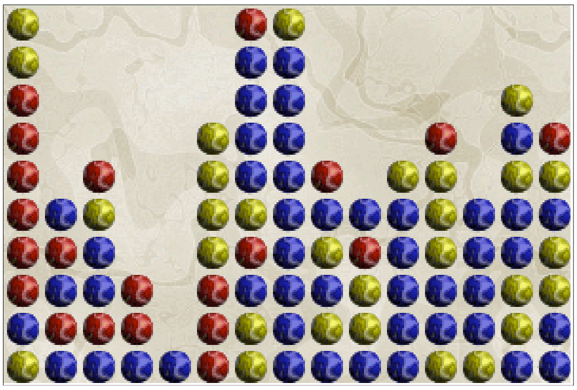


Figure 5. The board of SameGame, the Gnome/KDE version of HMaki.

Let us try our design pattern for implementing this board game (Figure 5).

□ Configuration of the Board

We give some values for the size of the board and the number of colors. We also set the flag for randomness.

```
Width = 12;
Height = 8;
Patterns = Range[5];
NeedRandomness = True;
```

Positions of colored tiles are given by a succession of calls to the pseudorandom number generator.

```
Clear[InitPosition];
InitPosition[l_, c_] := Random[Integer, Length[Patterns] - 1] + 1
```

```
Clear[InitPlay]; InitPlay[] := Null
```

Visualizing the Board

We are now able to initialize the game.

```
NewGame["new"]
```

```
Board // MatrixForm
```

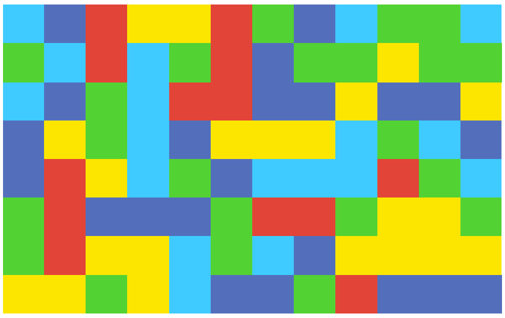
$$\begin{pmatrix} 1 & 4 & 5 & 2 & 2 & 5 & 3 & 4 & 1 & 3 & 3 & 1 \\ 3 & 1 & 5 & 1 & 3 & 5 & 4 & 3 & 3 & 2 & 3 & 3 \\ 1 & 4 & 3 & 1 & 5 & 5 & 4 & 4 & 2 & 4 & 4 & 2 \\ 4 & 2 & 3 & 1 & 4 & 2 & 2 & 2 & 1 & 3 & 1 & 4 \\ 4 & 5 & 2 & 1 & 3 & 4 & 1 & 1 & 1 & 5 & 3 & 1 \\ 3 & 5 & 4 & 4 & 4 & 3 & 5 & 5 & 3 & 2 & 2 & 3 \\ 3 & 5 & 2 & 2 & 1 & 3 & 1 & 4 & 2 & 2 & 2 & 2 \\ 2 & 2 & 3 & 2 & 1 & 4 & 4 & 3 & 5 & 4 & 4 & 4 \end{pmatrix}$$

Choosing some colors, we can define the function `View` for a nicer display of the board.

```
Needs["Graphics`Colors`"];
HMakiColors = {LightBeige, DeepSkyBlue,
   CadmiumLemon, LimeGreen, Cobalt, DeepMadderLake};
```

```
View[] := Show[
  Graphics[RasterArray[Transpose[Map[Reverse, Transpose[Board]]]] /.
    x_Integer :> HMakiColors[[x + 1]]]]
```

```
View[];
```



□ Transition Function

To deal with corner and boundary situations, we define the `BoardValue` function to access the values of the board. It returns −1 if the arguments for location are outside the bounds of the board. This allows us to ignore the boundaries of the board when considering the neighbours of a location.

```
BoardValue[{l_, c_}] :=
 If[Or[Not[0 < l ≤ Height], Not[0 < c ≤ Width]], -1, Board[[l, c]]]
```

The positions of neighbours are computed by the following two functions.

```
FaceNeighbours[p_] :=
 Map[Plus[p, #] &, {{0, 1}, {1, 0}, {0, -1}, {-1, 0}}]
CornerNeighbours[p_] :=
 Map[Plus[p, #] &, {{-1, 1}, {1, 1}, {1, -1}, {-1, -1}}]
```

To get the *spot* enclosing the selected location, (i.e., the neighbourhood of locations with the same color), we use a fixed-point algorithm applied to a neighbourhood extension function.

```
Spot[p_] := FixedPoint[Function[y,
   Union[y, Apply[Union, Map[Select[FaceNeighbours[#], Function[x,
         BoardValue[x] == BoardValue[First[y]]]] &, y]]]], {p}]
```

`IsPlayable` is now easy to implement: only a non-empty location within a spot of at least two can be played. The result of the function is the list of the locations within the spot.

```
IsPlayable[p_] := If[BoardValue[p] == 0, False,
  Module[{s = Spot[p]}, If[Length[s] > 1, s, False]]]
```

`PlayBoard` will "remove" the spot from the board, propagating individual tiles to the bottom and columns to the left whenever it is possible.

```
PlayBoard[p_, sp_] := Board = Transpose[
    PadRight[Select[Map[PadLeft[Select[#, Positive], Height] &,
        Transpose[ReplacePart[Board, 0, sp]]], Positive[
         Apply[Plus, #]] &], Width, x] /. x → Table[0, {i, Height}]];
```

□ Main Loop

To check whether the game is over, it is necessary to decide if there remain two spots of the same color contiguous by a face.

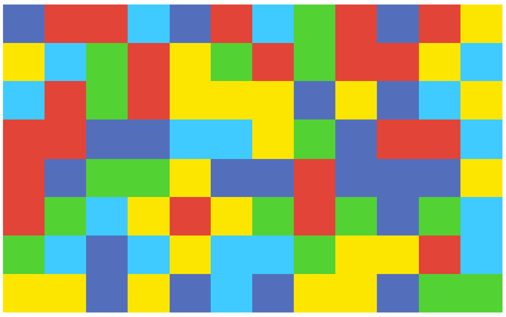
```
GameOver[] := And[Length[Select[Select[Flatten[Map[Split, Board], 1],
      Length[#] != 1 &], First[#] != 0 &]] == 0,
  Length[Select[Select[Flatten[Map[Split, Transpose[Board]], 1],
      Length[#] != 1 &], First[#] != 0 &]] == 0]
```

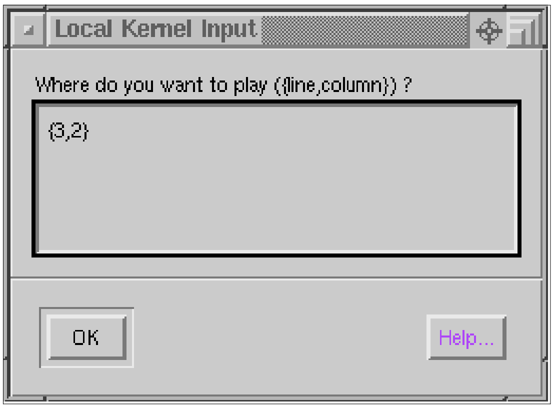
Interaction with the player here is very basic.

```
GetPlay[] := Input["Where do you want to play ({row,column}) ? "]
```

And we are able to play—but the interface is a bit cumbersome.

```
PlayGame["new"]
```





Where do you want to play ({line,column}) ?

{3,2}

With a Notebook Interface

To improve the interface, only one function has to be defined: one which colors the buttons and manages the end of the game (Figure 6).

```
NBMakeButton[c_, {i_, j_}] := ButtonBox[" ", ButtonData → {i, j},
  Background → HMakiColors[[c + 1]], ButtonFunction →
   (If[Not[GameOver[]], NBPlayThere[#2] &, NotebookClose[Nbview]]),
  ButtonEvaluator → Automatic]
```

And we are able to play without the keyboard!

```
NBPlayGame["new"]
```

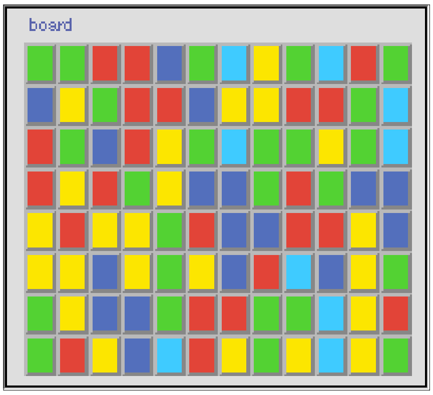


Figure 6. A view of the notebook interface of the board of HMaki.

■ Playing Lines

"Your task is to build lines of balls of the same color on the checker-board. Every time you move a ball, 3 new balls appear. When you build a line of 5 or more balls, these balls are removed from the board. Easy? And exciting!!!" *5star Free Lines—How to play* (www.5star-shareware.com/Windows/Games/Logic/5star-beelines.html)

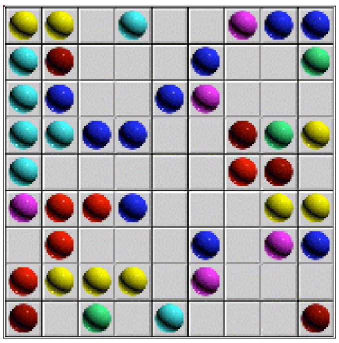


Figure 7. The board of the five-star version of the game Lines.

Once again, the design pattern will help us quickly implement this game (Figure 7).

□ Configuration of the Board

The following values define the physical parameters of the board.

```
Width = 9;
Height = 9;
Patterns = Range[7];
NeedRandomness = True;
```

Initially, the board is empty.

```
Clear[InitPosition]
InitPosition[l_, c_] := 0
```

But before the first play, the computer makes three balls appear.

```
Clear[InitPlay]
InitPlay[] := (Next[]; AddBall[NextBall]; Next[]; FromLoc = {};)
```

The colors of the next three balls to be added are selected in advance and must be shown to the player to help him play. They are selected randomly by the `Next` function and kept in a global variable, `NextBall`.

```
Next[] :=
  (NextBall = Table[RandomElement[Range[Length[Patterns]]], {3}];)
```

Adding Balls

The positions at which to add the balls are also selected randomly in the list of the empty positions.

```
AddBall[l_List] := Map[AddBall, l]
AddBall[c_Integer] := Module[{p = RandomElement[Position[Board, 0]]},
   (Board = ReplacePart[Board, c, p]; NewLine[p];)]
```

`RandomElement` is a miscellaneous function randomly selecting an element of a list.

```
RandomElement[l_List] := Part[l, Random[Integer, Length[l] - 1] + 1]
```

□ Visualizing the Board

Let us initialize the game and visualize it.

```
NewGame["new"]
```

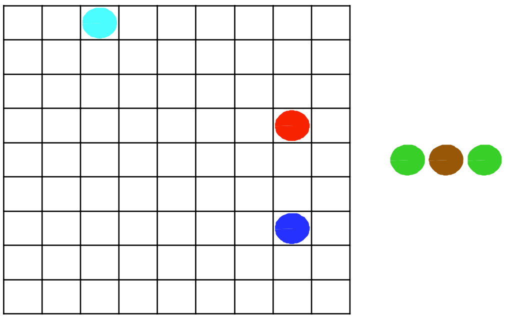
With some colors and a few graphics primitives, we are able to display the board and the next three balls.

```
LinesColors = { RGBColor[0.8, 0.8, 0.8], Blue, Red,
   PermanentGreen, Yellow, SaddleBrown, Cyan, Magenta};

ViewBall[{i_, {l_, c_}}] :=
  {LinesColors[[i + 1]], Disk[{c - 0.5, Height - l + 0.5}, 0.45]}

View[] := Show[
   Graphics[Join[Map[Line[{{#, 0}, {#, Height}}] &, Range[0, Width]],
     Map[Line[{{0, #}, {Width, #}}] &, Range[0, Height]],
     Map[ViewBall, Module[{p = Position[Board, x_ /; x != 0]},
       Transpose[{Extract[Board, p], p}]]],
     MapIndexed[ViewBall[{#1, {5, 10 + #2[[1]]}}] &, NextBall]]]]
```

```
View[];
```



□ Transition Function

In this game, a play consists of three possible successive actions: selecting and adding three balls, moving a ball, and deleting a line of balls. When a player selects a ball and a new location for it, this ball is moved. If possible a line is deleted and the computer adds three balls, possibly deleting a line.

Moving Balls

To decide whether a motion is valid or not, we again use a fixed-point algorithm applied to a neighbourhood extension function: the set of reachable positions is incrementally built from the starting position by exploring empty locations.

We reuse the function `BoardValue` to simplify the computations of the neighbourhood.

```
BoardValue[{l_, c_}] :=
 If[Or[Not[0 < l ≤ Height], Not[0 < c ≤ Width]], -1, Board[[l, c]]]
```

`Neighbour` is slightly different than `FaceNeighbour` from the previous example.

```
Neighbour[{l_, c_}] :=
 {{l - 1, c}, {l, c - 1}, {l, c}, {l, c + 1}, {l + 1, c}}
```

`Possible` searches for empty locations in the neighbourhood.

```
Possible[{l_Integer, c_Integer}] :=
 Select[Neighbour[{l, c}], MemberQ[Position[Board, 0], #] &]
Possible[lp_List] := Apply[Union, Map[Possible, lp]]
```

The fixed-point algorithm is performed by `IsPlayable`, which returns a Boolean.

```
IsPlayable[p1_, p2_] := And[BoardValue[p1] != 0,
  BoardValue[p2] == 0, MemberQ[FixedPoint[Possible, p1], p2]]
```

`PlayBoard` updates the board, checks for lines of five balls to delete, and adds three balls.

```
PlayBoard[p1_, p2_, _] :=
 (Board = ReplacePart[Board, BoardValue[p1], p2];
  Board = ReplacePart[Board, 0, p1];
  If[Not[NewLine[p2]], (AddBall[NextBall]; Next[];)])
```

Checking for Lines to Delete

To check if there is a line of five balls to delete, we try to extend the position in four (nonoriented) directions as long as the color of the ball remains the same. This is performed by `Extend`.

```
Extend[pos_, dir_] :=
 FixedPoint[If[BoardValue[First[#] - dir] == BoardValue[First[#]],
    Prepend[#, First[#] - dir], #] &,
  FixedPoint[If[BoardValue[Last[#] + dir] == BoardValue[Last[#]],
     Append[#, Last[#] + dir], #] &, pos]]
Extend[pos_] := Map[Extend[{pos}, #] &,
  {{1, 0}, {1, 1}, {0, 1}, {-1, 1}}]
```

`DeleteBall` suppresses the balls from the line.

```
DeleteBall[p_] := (Board = ReplacePart[Board, 0, p];)
```

And `NewLine` is the top-level function, returning a Boolean.

```
NewLine[p_] :=
 Module[{l = Flatten[Select[Extend[p], Length[#] > 4 &], 1]},
  (Map[DeleteBall, l]; Not[l == {}])]
```

□ Main Loop

Playing

The game continues until fewer than three slots are available for the balls.

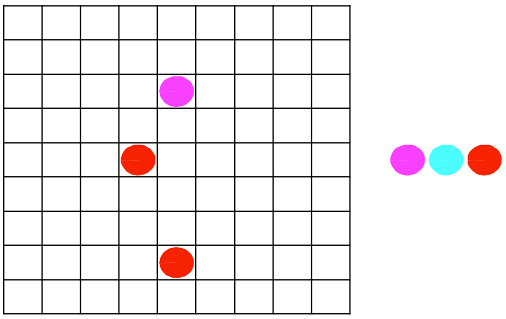
```
GameOver[] := Length[Select[Flatten[Board], # == 0 &]] < 3
```

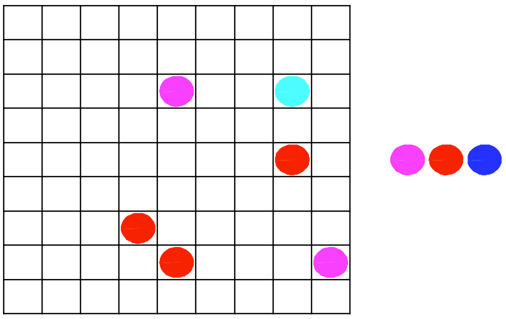
Once again, the interface with the player is extremely basic.

```
GetPlay[] := Input["which motion (
    {{start_line,start_column},{end_line,end_column}})?"]
```

But we can play!

```
PlayGame["new"]
```





□ A Notebook Interface

To cope with the display of the "balls to come", it is necessary to adapt the `NBView` function and develop the dedicated function `NBNext`, which is similar to `NBBoard`.

```
NBView[] :=
 Notebook[{Cell[BoxData[NBNext[NextBall]], CellTags → {"next"}],
   Cell[BoxData[NBBoard[Board]], CellTags → {"board"}]},
  Apply[Sequence, NBOptions]]

NBNext[l_] :=
 GridBox[{Map[ButtonBox[" ", Background → LinesColors[[# + 1]]] &, l]}]
```

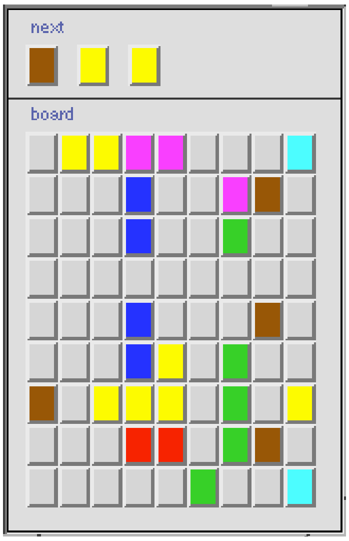
To select two locations, the player clicks twice in a play on the board. This is taken into account through the variable `FromLoc` and by a small difference in the implementation of `NBPlayThere`.

```
NBMakeButton[c_, {i_, j_}] :=
 ButtonBox[" ", ButtonData → {i, j}, Background → LinesColors[[c + 1]],
  ButtonFunction → (If[Not[GameOver[]], NBPlayThere[{FromLoc, #2}] &,
     NotebookClose[Nbview]]), ButtonEvaluator → Automatic]

NBPlayThere[{{}, p_}] := (FromLoc = p; NBRefresh[])
NBPlayThere[p_] := (PlayThere[p]; FromLoc = {}; NBRefresh[])
```

With the notebook interface, playing becomes easier, even if the graphical design is not so nice.

```
NBPlayGame["new"]
```



■ Mancala: A Two-Player Game

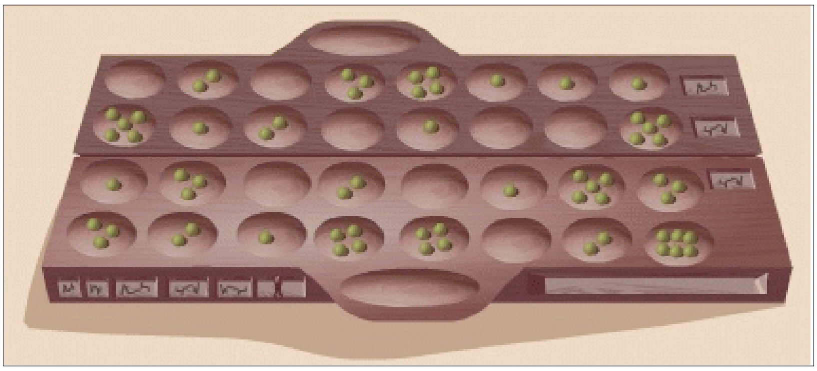


Figure 8. A board of 4 x 8 Mancala.

Mancala is the ancient game of counting and strategy where each player must attempt to collect as many stones as possible before one of the players clears his side of stones (Figure 8). There are many versions of this traditional game.

Here are the rules of the version we will implement:

Each player has a side of the board. The six cups nearest each player belong to him and his mancala—another cup in which to place the captured stones—is to the right. Players alternate turns. During his turn, each player selects a cup of stones from one box on his side of the board. Each stone is placed one by one in the cups around the board (going counterclockwise), including his mancala but not the opponent's mancala. If the last stone lands in the player's own mancala, that player goes again. If the last stone lands in an empty cup on the player's own side, he captures all the stones from the opponent's cup directly opposite that cup. The game is over when a player has no more stones in play on the board. The winner is the player with the greatest total of stones in his mancala.

□ Configuration of the Board

The following values define the physical parameters of the board (i.e., in our example a set of six cups for each of the two players). The last column of each row corresponds to the mancala of the player. The patterns are the number of stones in a cup.

```
Width = 6 + 1;
Height = 2;
Patterns = Range[48];
NeedRandomness = False;
```

Initially, the stones are equally distributed in the cups.

```
Clear[InitPosition]
InitPosition[l_, Width] := 0
InitPosition[l_, c_] := Last[Patterns] / (Height (Width - 1))
```

We use an additional global variable, `Player`, to manage the players.

```
Clear[InitPlay]
InitPlay[] := (Player = 1;)
```

□ Visualizing the Board

A very basic visualization of the board is sufficient for playing—mancalas are separated.

```
NewGame[]

View[] := Print[{Last[First[Board]],
   MatrixForm[{Reverse[Most[First[Board]]], Most[Last[Board]]}],
   Last[Last[Board]]}]

View[]
```

$$\left\{0, \begin{pmatrix} 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 \end{pmatrix}, 0\right\}$$

□ Transition Function

In this game, playing is just selecting one of the cups. `IsPlayable` only verifies that the selected cup is not empty.

```
IsPlayable[{p_, Width}] := False
IsPlayable[{p_, c_}] :=
 Module[{n = Board[[p, c]]}, If[n == 0, False, n]]
```

The rules of the evolution of the board are much trickier to implement as many different cases may arise. `GiveStone` redistributes the stones of the selected cup.

```
GiveStone[{1, c_}, n_Integer] :=
 Map[Partition[#, Width, Width, {1, 1}, 0] &,
  Partition[Join[Array[0 &, c], Array[1 &, n]],
   2 Width - 1, 2 Width - 1, {1, 1}, 0]]
GiveStone[{2, c_}, n_Integer] := Partition[
  Append[Prepend[Flatten[GiveStone[{1, c}, n], 1], Array[0 &, Width]],
   Array[0 &, Width]], 2]
```

```
Other[1] = 2; Other[2] = 1;
```

`Other` just permutes 1 and 2, while `PlayBoard` manages the complete evolution of the board.

```
PlayBoard[{p_, c_}, n_Integer] :=
 (Board = ReplacePart[Board, 0, {p, c}];
  Board = Fold[Plus, Board, GiveStone[{p, c}, n]];
  Module[{m = Mod[c + n, 2 Width - 1]}, If[m == Width, Null,
    (Player = Other[Player]; If[And[m < Width, Board[[p, m]] == 1],
      (Board = ReplacePart[Board, Board[[Other[p], Width - m]] +
          Board[[p, Width]] + 1, {p, Width}];
       Board = ReplacePart[Board, 0, {Other[p], Width - m}];
       Board = ReplacePart[Board, 0, {p, m}];)])]])
```

□ Main Loop

The game ends when one player's cups are empty.

```
GameOver[] := Apply[Or, Map[Union[Most[#]] == {0} &, Board]]
```

For this game, the basic textual interface for playing is quite convenient, as the player is only asked for one digit at each play.

```
GetPlay[] :=
 {Player, Input["Player " <> ToString[Player] <> ": which cup (1-6) ?"]}
```

When one player has no more stones, the other player puts all his remaining stones in his mancala. This is implemented by a refinement of the function `EndGame`.

```
EndGame[] := (Board =
   Map[Append[Array[0 &, Width - 1], Apply[Plus, #]] &, Board]; View[])
```

```
PlayGame[]
```

$$\left\{0, \begin{pmatrix} 4 & 4 & 4 & 4 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 \end{pmatrix}, 0\right\}$$

$$\left\{1, \begin{pmatrix} 5 & 5 & 5 & 0 & 4 & 4 \\ 4 & 4 & 4 & 4 & 4 & 4 \end{pmatrix}, 0\right\}$$

$$\left\{2, \begin{pmatrix} 0 & 5 & 5 & 0 & 4 & 4 \\ 5 & 5 & 5 & 5 & 4 & 4 \end{pmatrix}, 0\right\}$$

$$\left\{2, \begin{pmatrix} 0 & 5 & 5 & 0 & 4 & 4 \\ 5 & 0 & 6 & 6 & 5 & 5 \end{pmatrix}, 1\right\}$$

$$\left\{2, \begin{pmatrix} 0 & 5 & 6 & 1 & 5 & 5 \\ 5 & 0 & 6 & 6 & 5 & 0 \end{pmatrix}, 2\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 6 & 7 & 2 & 6 & 0 \\ 0 & 0 & 6 & 6 & 5 & 0 \end{pmatrix}, 2\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 6 & 7 & 2 & 7 & 1 \\ 0 & 0 & 0 & 7 & 6 & 1 \end{pmatrix}, 3\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 7 & 8 & 0 & 7 & 1 \\ 0 & 0 & 0 & 7 & 6 & 1 \end{pmatrix}, 3\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 7 & 8 & 0 & 7 & 1 \\ 0 & 0 & 0 & 7 & 6 & 0 \end{pmatrix}, 4\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 7 & 9 & 1 & 8 & 2 \\ 0 & 0 & 0 & 0 & 7 & 1 \end{pmatrix}, 5\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 7 & 10 & 0 & 8 & 2 \\ 0 & 0 & 0 & 0 & 7 & 1 \end{pmatrix}, 5\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 7 & 10 & 0 & 8 & 2 \\ 0 & 0 & 0 & 0 & 7 & 0 \end{pmatrix}, 6\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 8 & 11 & 1 & 9 & 3 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, 7\right\}$$

$$\left\{8, \begin{pmatrix} 0 & 8 & 12 & 2 & 10 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, 7\right\}$$

$$\left\{40, \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, 8\right\}$$

■ Interface with *GUIKit*

"*GUIKit* simplifies the construction and layout of common user interface programming and eliminates the need to write code using the underlying Java programming language." *GUIKit online documentation* (documents.wolfram.com/solutions/guikit).

Although we have succeeded in easily building a graphical interface for playing using the notebooks and buttons in them, the result is very poor in terms of graphics and functionalities.

Until a new version of graphics management exists in *Mathematica*, there is no way, within notebooks, to make a part of a graphic object active. But *GUIKit* makes the power of *J/Link* and the richness of AWT and Swing Java libraries accessible.

In this section, we go back to the HMaki example and show how to build a nice and efficient interface with this new package.

□ Visualizing the Board

GUIKit provides `Widget["IndexedImagePanel"]`, which allows us to display an array or rectangles with different colors. Very few settings are necessary to represent the HMaki board with this widget.

```
Needs["GUIKit`"];

Width = 20; Height = 15; Patterns = Range[5];
NeedRandomness = True; PixelSize = 12;

IndexedImagePanelWidget :=
  Widget["IndexedImagePanel", {"preferredSize" →
     Widget["Dimension", {"width" → PixelSize (Width + 2),
       "height" → PixelSize (Height + 2)}], "imageWidth" → Width,
    "imageHeight" → Height, "imagePixelSize" → PixelSize,
    "imageColorMapSize" → Length[HMakiColors],
    "imageGrid" → True, "imageGridColor" →
     Widget["Color", InitialArguments → {255, 255, 255}],
    "scaleImage" → False, "imageColorComponents" → Flatten[
      Map[Floor[255 #] &, Map[Apply[List, #] &, HMakiColors]]],
    "imagePixels" → Flatten[Board], MouseClickBinding},
   Name → "hmakiPanel"];
```

To transform this widget into an active one, we have to bind the event of clicking with the mouse to the action of playing and refreshing the board.

```
MouseClickBinding = BindEvent["mouseClicked",
   Script[mouseEvent = WidgetReference["#"]; coords = InvokeMethod[
      {"hmakiPanel", "getImagePixelCoordinatesAt"}, mouseEvent];
    PlayThere[Reverse[coords] + {1, 1}]; SetPropertyValue[
     {"hmakiPanel", "imagePixels"}, Flatten[Board]]]];
```

A JPlayGame function is then defined, initializing the game, loading, and running the interface definition.

```
JPlayGame[x___] := (NewGame[x]; GUIRunModal[
   Widget["Frame", {"title" → "HMaki",
     "Size" → Widget["Dimension", {"width" → 300, "height" → 257}],
     "resizable" → False, IndexedImagePanelWidget}],
   IncludedScriptContexts → {$Context}])
```

Here comes the playable Java version of HMaki!

```
JPlayGame["new"]
```

□ Adding Interactivity

We now would like to be able to start a new game or replay the current game directly from the interface.

A good way to implement this interactivity is to define `Widget["Action"]` for each action and attach the corresponding `Script` to it. The script can refer to any *Mathematica* function, provided its context is within the scope of the script. Here, we use the function `NewGame` in our scripts.

```
ActionsWidget := {Widget["Action", {"name" → "New Game",
     BindEvent["action", Script[playGame["new"]]]},
   Name → "newGameAction"], Widget["Action", {"name" → "Replay",
     BindEvent["action", Script[playGame[]]]}, Name → "replayAction"],
  Script[playGame[x___] := (NewGame[x]; SetPropertyValue[
      {"hmakiPanel", "imagePixels"}, Flatten[Board]])]}
```

It is then easy to define a menu, associating an action to each of the menu items

```
MenuBarWidget := Widget["MenuBar", {Widget["Menu",
     {"text" → "Game", Widget["MenuItem", {"text" → "NewGame",
        "action" → WidgetReference["newGameAction"]}], Widget[
       "MenuItem", {"action" → WidgetReference["replayAction"]}]}]}]
```

or associating actions with buttons.

```
BottomWidget := WidgetGroup[
  {Widget["Button", {"action" → WidgetReference["newGameAction"]}],
   Widget["Button", {"action" → WidgetReference["replayAction"]}],
   WidgetFill[]}]
```

```
JPlayGame[x___] := (NewGame[x]; GUIRunModal[
   Widget["Frame", {"title" → "HMaki",
     "resizable" → False, Append[ActionsWidget, Widget["Panel",
       {IndexedImagePanelWidget, WidgetFill[], BottomWidget}]],
     "menus" → MenuBarWidget}, Name → "hmakiFrame"],
   IncludedScriptContexts → {$Context}])
```

```
JPlayGame["new"]
```

□ Adding Scores

When you play HMaki, you can target various goals: suppressing as many tiles as possible, having as many tiles as possible once all the groups have been suppressed, suppressing as many biggest groups as possible, and so on. From the board, it is possible to compute a score, which measures how close to your goal you are.

Computation

The simplest score function computes the number of tiles on the board.

```
Score[] := Length[Select[Flatten[Board], # != 0 &]]
```

Visualization

To get the score on the interface, we should first set a label widget to show it as text.

```
BottomWidget := WidgetGroup[{WidgetSpace[10],
   Widget["Button", {"action" → WidgetReference["newGameAction"]}],
   Widget["Button", {"action" → WidgetReference["replayAction"]}],
   WidgetFill[], Widget["Label", {"text" → "tiles:"}],
   Widget["Label", {"text" → ToString[Width Height]},
    Name → "hmakiScore"], WidgetSpace[10]}]
```

We should also modify what happens when the mouse is clicked inside the board (i.e., change the definition of `MouseClickBinding`).

```
MouseClickBinding = BindEvent["mouseClicked",
   Script[mouseEvent = WidgetReference["#"]; coords = InvokeMethod[
      {"hmakiPanel", "getImagePixelCoordinatesAt"}, mouseEvent];
    PlayThere[Reverse[coords] + {1, 1}];
    SetPropertyValue[{"hmakiPanel", "imagePixels"}, Flatten[Board]];
    SetPropertyValue[{"hmakiScore", "text"}, Score[]]]];

JPlayGame["new"]
```

The previous command produces the Java interface (Figure 9).

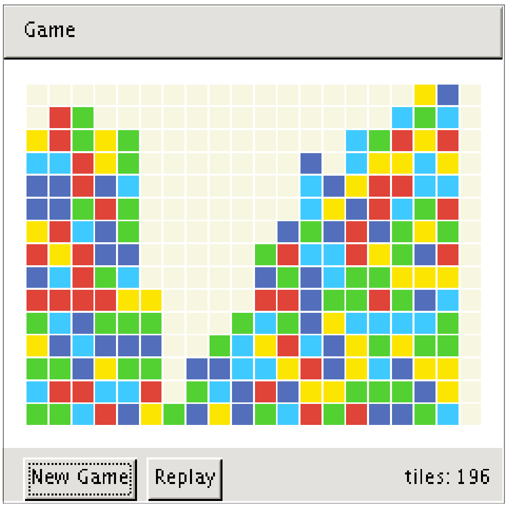


Figure 9. *GUIKit*-generated Java version of HMaki.

■ Analysis and Strategies

In this section, we will show how easy it is to make the computer play on its own in order to analyze games or analyze, study, and improve playing strategies. We will focus again on the HMaki game, which is complex enough to demonstrate how powerful the analysis can be with our design.

```
Width = 20; Height = 15; Patterns = Range[5];
NeedRandomness = True; PixelSize = 12;
```

□ Automatic Play

As randomness is involved in the initialization of the game, we need to play repeatedly to study the game with statistical tools. So the computer should be able to play a game alone and give us feedback on it.

Scoring

We are interested in the final score and the number of plays necessary to reach the end. Moreover, we consider two kind of scores: the number of remaining tiles and the sum of the scores obtained for removing each group of tiles—growing with the square of the size of the group. The global value `Score` contains a list of this information; it is initialized by `InitScore` and updated by `UpdateScore`.

```
InitScore[] := Score = {0, 0, Width Height}
```

```
UpdateScore[] :=
 Module[{n = Width Height - Length[Select[Flatten[Board], # == 0 &]]},
  Score = {First[Score] + 1, Score[[2]] + (Last[Score] - n - 2)^2, n}]
```

Where to Play

Before playing, the computer should know where it is valid to play and then where to play. The function `PlayableList` computes the list of all the playable locations on the board, using an almost trivial algorithm.

```
PlayableList[] := Fold[PlayableList, {}, Patterns]
PlayableList[l_List, i_Integer] :=
 Join[l, PlayableList[Position[Board, i]]]
PlayableList[l_] := Select[l, Not[FalseQ[IsPlayable[#]]] &]
```

The `GetPlay` function is rewritten to randomly select an element in the list of playable locations.

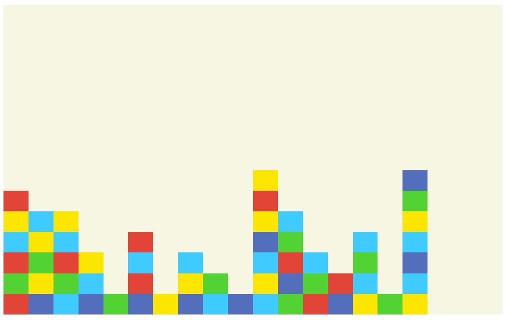
```
GetPlay[] := RandomElement[PlayableList[]]
```

Playing

We define `BatchPlayGame`, an analog function to `PlayGame`, to make the computer play. Note the presence of the bound `MaxPlay` to avoid infinite loops and the call to `SeedRandom` to control random initialization of the board and select a play among the list of possible beginnings.

```
MaxPlay = Width Height / 2 ;
BatchPlayGame[x___] := (NewGame[x]; InitScore[];
  SeedRandom[]; While[And[! GameOver[], First[Score] < MaxPlay],
   PlayThere[GetPlay[]]; UpdateScore[]]; EndGame[])

Timing[BatchPlayGame["new"];]
```



```
{3.984 Second, Null}

Score

{76, 270, 60}
```

In the preceding results, the high computation time is due to the poor implementation of `PlayableList`: 76 is the number of plays in the game; 270 is the score when "playing for big"; and 60 is the number of remaining tiles.

□ Statistical Analysis

Once the computer is able to play alone and produce information on the game, we can make it play enough games to give us statistical results. Let us conduct an experiment.

Playing 100 Games

The first experiment consists of playing 100 different games; the second in playing the same game 100 times.

```
EndGame[] := Null;
ScoreData1 = {};
Do[(BatchPlayGame["new"]; AppendTo[ScoreData1, Score]), {k, 100}]
```

```
ScoreData2 = {};
Do[(BatchPlayGame[]; AppendTo[ScoreData2, Score]), {k, 100}]
```

Statistical standard functions describe the dispersion of this data.

```
Map[N[{Mean[#], Variance[#], StandardDeviation[#], Median[#]}] &,
 Transpose[ScoreData1]]
```

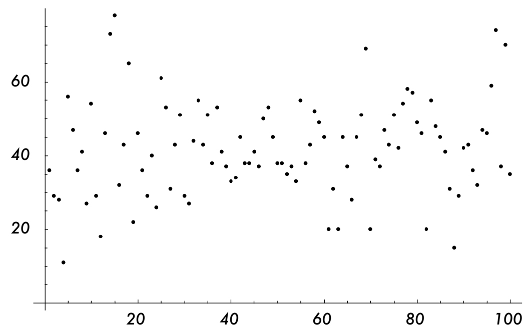
```
{{82.56, 41.562, 6.44686, 82.},
 {317.68, 14366.9, 119.862, 302.5}, {43.8, 170.525, 13.0585, 43.5}}
```

```
Map[N[{Mean[#], Variance[#], StandardDeviation[#], Median[#]}] &,
 Transpose[ScoreData2]]
```

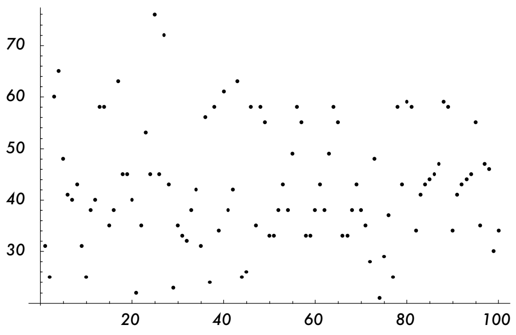
```
{{85.21, 29.4403, 5.42589, 85.},
 {221.57, 2916.09, 54.0008, 215.}, {48.59, 144.608, 12.0253, 49.}}
```

We can view the data using standard graphical functions for plotting.

```
ListPlot[Map[Last, ScoreData1]]
```



```
ListPlot[Map[Last, ScoreData2]]
```



The fairly even dispersion we observe in these graphics is a testimony to the interest of the game for a human player.

□ Strategies

Another goal of statistical analysis is to compare strategies and get a better understanding of how to play.

Groups

Simple strategies for playing HMaki are based on knowledge of the different groups of tiles of the same colors. The `SpotList` function returns the list of the groups computed from the board.

```
SpotList[] := Fold[SpotList, {}, Patterns]
SpotList[l_List, i_Integer] :=
 Join[l, SpotList[{}, Position[Board, i]]]
SpotList[sl_List, {f_, o___}] :=
 Module[{s = IsPlayable[f]}, If[FalseQ[s], SpotList[sl, {o}],
   SpotList[Append[sl, s], Complement[{o}, s]]]]
SpotList[sl_List, {}] := sl
```

Strategies

A new version of `BatchPlayGame` takes an integer argument as an index of the selected strategy, and an option for visualization.

```
Options[BatchPlayGame] = {Viewer -> False, ViewStart -> False};
BatchPlayGame[{x___}, i_Integer, v___?OptionQ] :=
 (If[ViewStart /. {v} /. Options[BatchPlayGame],
   NewGame[x]; View[], NewGame[x]]; InitScore[];
  SeedRandom[]; While[And[! GameOver[], First[Score] < MaxPlay],
   PlayThere[GetPlay[i]]; UpdateScore[]];
  If[Viewer /. {v} /. Options[BatchPlayGame], EndGame[]])
```

Different instances of `GetPlay` correspond to different strategies of selecting the group of tiles to remove:

1. the group is randomly selected in the list of groups
2. the group is randomly selected among the biggest groups

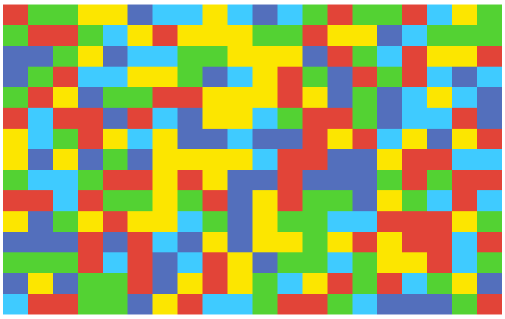
3. the first biggest group (starting from the upper-left corner, then from left to right and top to bottom)
4. the group is randomly selected among the smallest groups
5. the first smallest group (starting from the upper-left corner, then from left to right and top to bottom)

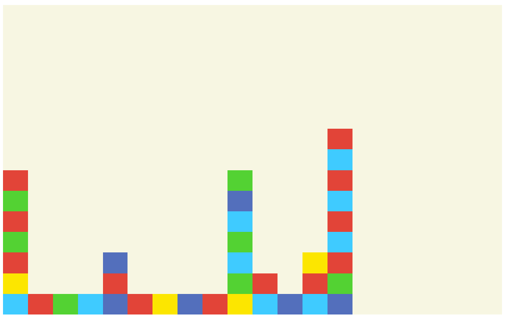
```
GetPlay[0] := First[RandomElement[SpotList[]]]
GetPlay[1] := First[RandomElement[Module[{s = SpotList[]},
    Module[{l = Map[Length, s]}, Extract[s, Position[l, Max[l]]]]]]]
GetPlay[2] := First[First[Module[{s = SpotList[]},
    Module[{l = Map[Length, s]}, Extract[s, Position[l, Max[l]]]]]]]
GetPlay[3] := First[RandomElement[Module[{s = SpotList[]},
    Module[{l = Map[Length, s]}, Extract[s, Position[l, Min[l]]]]]]]
GetPlay[4] := First[First[Module[{s = SpotList[]},
    Module[{l = Map[Length, s]}, Extract[s, Position[l, Min[l]]]]]]]
```

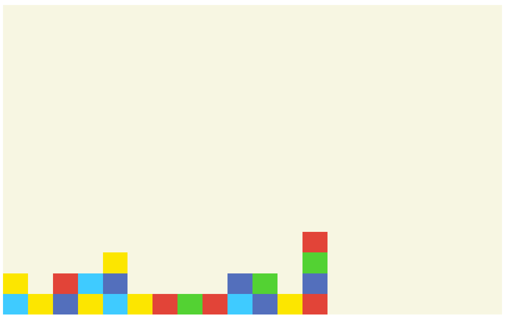
Experiment

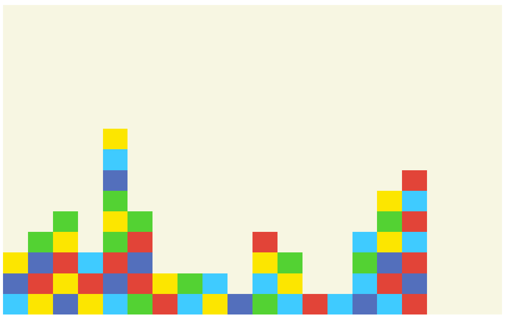
Results of 170 games were collected—10 different games played 17 times: five times for the strategies involving randomness, only once for the two others. These results took several minutes to produce.

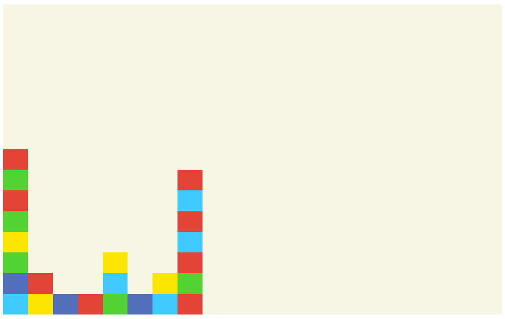
```
StrategyData = {};
(DataLine = {}; DataWord = {};
  BatchPlayGame[{"new"}, 0, ViewStart → True, Viewer → True];
  AppendTo[DataWord, Score];
  Do[(BatchPlayGame[{}, 0]; AppendTo[DataWord, Score]), {4}];
  AppendTo[DataLine, DataWord]; DataWord = {};
  BatchPlayGame[{}, 1, Viewer → True]; AppendTo[DataWord, Score];
  Do[(BatchPlayGame[{}, 1]; AppendTo[DataWord, Score]), {4}];
  AppendTo[DataLine, DataWord]; BatchPlayGame[{}, 2, Viewer → True];
  AppendTo[DataLine, Score]; DataWord = {};
  BatchPlayGame[{}, 3, Viewer → True]; AppendTo[DataWord, Score];
  Do[(BatchPlayGame[{}, 3]; AppendTo[DataWord, Score]), {4}];
  AppendTo[DataLine, DataWord]; BatchPlayGame[{}, 4, Viewer → True];
  AppendTo[DataLine, Score]; AppendTo[StrategyData, DataLine]);
Do[(DataLine = {}; DataWord = {}; BatchPlayGame[{"new"}, 0];
  AppendTo[DataWord, Score];
  Do[(BatchPlayGame[{}, 0]; AppendTo[DataWord, Score]), {4}];
  AppendTo[DataLine, DataWord]; DataWord = {};
  Do[(BatchPlayGame[{}, 1]; AppendTo[DataWord, Score]), {5}];
  AppendTo[DataLine, DataWord]; BatchPlayGame[{}, 2];
  AppendTo[DataLine, Score]; DataWord = {};
  Do[(BatchPlayGame[{}, 3]; AppendTo[DataWord, Score]), {5}];
  AppendTo[DataLine, DataWord]; BatchPlayGame[{}, 4];
  AppendTo[DataLine, Score]; AppendTo[StrategyData, DataLine];), {9}]
```

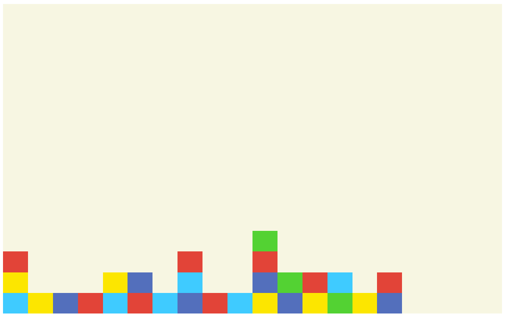












The following command produces the results in Table 1.

```
TableForm[Partition[Map[N[Mean[#]] &,
   Transpose[Map[Flatten, Map[Function[1, Map[If[Depth[#] > 2,
          Map[Function[x, N[Mean[x]]], Transpose[#]], #] &, 1]],
      StrategyData]]]], 3], TableHeadings → {Range[5] - 1,
   {"Number of plays", "Playing for big", "Remaining tiles"}}]
```

	Number of plays	Playing for big	Remaining tiles
0	86.76	343.4	39.92
1	76.54	302.04	41.6
2	75.3	319.	42.
3	97.26	411.44	36.96
4	96.2	450.2	39.2

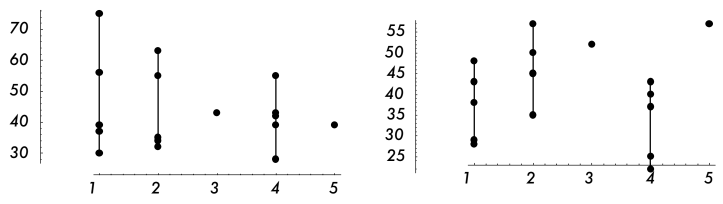
Table 1. Mean of the results over 10 games.

We can also visualize these results with an ad hoc plot function. The following plots show the numbers of remaining tiles for the first strategy.

```
Draw[y_Integer, {x_Integer}] := Point[{x, y}]
Draw[y_List, {x_Integer}] :=
 Append[Map[Draw[#, {x}] &, y], Line[{{x, Min[y]}, {x, Max[y]}}]]
```

```
PlotStrategy[n_Integer] :=
 Graphics[Join[{PointSize[0.03]}, MapIndexed[Draw, Map[
     If[Depth[#] > 2, Map[Last, #], Last[#]] &, StrategyData[[n]]]]],
   Axes → True, AxesOrigin → {0, Min[Flatten[StrategyData[[1]]]] - 5}]

Show[GraphicsArray[Map[PlotStrategy, Partition[Range[2], 2], {2}]]]
```



Conclusion

This exercise proved that a couple of hundreds lines of *Mathematica* code are enough to completely implement three playable prototypes of board games together with the corresponding basic graphical user interface, plus an attractive Java Swing version of one of them. With the design pattern presented, we demonstrated that most of this code will be reusable with minor changes for implementing other board games.

As playing is one of the oldest human activities, trying to understand how the games work and why some of them are so addictive could be a recreational and exciting challenge. The library of applications that we created could be extended to different games. Many strategies could be implemented and tested, even on our three examples, to better understand and enjoy them.

References

[1] D. Parlett, *Oxford History of Board Games*, Oxford: Oxford University Press, 1999.

[2] The International Society for Board Games Studies, The Research School CNWS, Leiden, The Netherlands: Leiden University, (Nov 2005).

[3] S. Wolfram, *The Mathematica Book*, 5th ed., Champaign, Oxford: Wolfram Media/ Cambridge University Press, 2003.

[4] M. Loy, R. Eckstein, ed., D. Wood, J. Elliott, and B. Cole, *Java Swing*, 2nd ed., Sebastopol, CA: O'Reilly & Associates, Inc., 2002.

Y. Papegay, "Exploring Board Game Strategies," *The Mathematica Journal*, 2012. dx.doi.org/10.3888/tmj.10.2-6.

About the Author

Yves Papegay has been a computer science researcher at INRIA Sophia Antipolis in southern France since 1994. One of his major research interests is symbolic computation methods, tools, and applications to modeling and simulation processes.

Papegay joined the French team of the Wolfram Education Group in its early weeks. He is a daily user of *Mathematica* and has developed several application packages for industrial purposes.

Yves Papegay
INRIA Sophia Antipolis
COPRIN 2004 route des Lucioles
F-06902 Sophia Antipolis
France
Yves.Papegay@sophia.inria.fr