Playing the Minesweeper with Constraints

Raphaël Collet

Université catholique de Louvain,
B-1348 Louvain-la-Neuve, Belgium.
raph@info.ucl.ac.be

Abstract. We present the design and implementation of a Minesweeper game, augmented with a digital assistant. The assistant uses contraint programming techniques to help the player, and is able to play the game by itself. It predicts safe moves, and gives probabilistic information when safe moves cannot be found.

1 Introduction

The Minesweeper game has been popular for several years now. Part of its popularity might come from its simplicity. A board represents a mine field, with mines hidden under the squares. The game consists in finding the mines without making them explode. You get new hints each time you uncover a non-mined square. Though, the simplicity does not make the game easy. The Minesweeper problem is hard: it has been proven NP-complete by Richard Kaye [1]. So simple techniques are not enough to solve it.

In this paper we show how the problem of finding safe moves can be modeled as a Constraint Satisfaction Problem (CSP). Techniques from the field of constraint programming can be used to program a digital assistant for a player. We applied them in a real application, the Oz Minesweeper. This relatively small program demonstrates the power of the programming language Oz.

Paper organization. Section 2 recalls the rules of the game, and proposes a simple mathematical model for it. Section 3 investigates how constraint programming techniques can be applied in order to solve the problem with reasonable efficiency. Section 4 then gives an overview of the implementation of the Oz Minesweeper. Section 5 evaluates and quickly compares our work to other similar products.

2 The Game as a Constraint Satisfaction Problem

Let us recall the rules of the game. A mine field is given to the player as a rectangular board. Each square on the board may hide at most one mine. The total number of mines is known by the player. A move consists in uncovering a square. If the square holds a mine, the mine explodes and the game is over. Otherwise, a number in the square indicates how many mines are held in the surrounding squares, which are the adjacent squares in the eight directions north, north-east, east, south-east, south, south-west, west, and north-west. The goal of the game is to uncover all the squares that do not hold a mine.

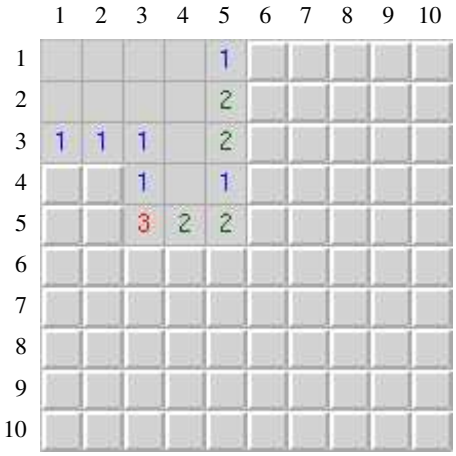

Fig. 1. An example of a board with 20 mines.

An example of a board is given in Fig. 1. This board contains 20 mines. Each square is identified by its coordinates $(row, column)$. The squares (1,1), (1,2), (1,3), (1,4), (2,1), (2,2), (2,3), (2,4), (3,4), and (4,4) have already been played, and have no mine in their respective surrounding squares. The squares (1,5), (3,1), (3,2), (3,3), (4,3), and (4,5) have been played, too, and are surrounded by one mined square each. The square (2,5), (3,5), (5,4), and (5,5) each have two mines in their neighborhood, while the square (5,3) has three mines around it. In this example, the player might deduce from (3,3) that (4,2) is mined, and by (3,2) that (4,1) is a safe move.

Finding safe moves on the board consists in solving the problem given by those numbers in the squares. The unknown of the problem is the positions of the mines. We model this as a binary matrix that represents the mine field, with one entry per square. The value 1 means that the corresponding square is mined, while 0 means a safe square.

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & & x_{2n} \\ \vdots & & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{pmatrix}$$

From now on, x_{ij} will always denote the matrix entry corresponding to the square at position (i, j). The problem can be written as linear equations over the x_{ij}'s. In the example, we have 20 mines (first equation below), and the played squares are not mined (second equation below). The other equations are given by the numbers in the squares.

The corresponding square coordinates are given on the left of each equation.

$$
\begin{array}{ll}
 & \sum_{i,j\in\{1,\ldots,10\}} x_{ij} = 20 \\
 & x_{11} = x_{12} = x_{13} = \cdots = 0 \\
(1,1) & x_{12} + x_{21} + x_{22} = 0 \\
(1,2) & x_{11} + x_{13} + x_{21} + x_{22} + x_{23} = 0 \\
 & \cdots \\
(1,5) & x_{14} + x_{16} + x_{24} + x_{25} + x_{26} = 1 \\
 & \cdots \\
(2,5) & x_{14} + x_{15} + x_{16} + x_{24} + x_{26} + x_{34} + x_{35} + x_{36} = 2 \\
 & \cdots
\end{array}
$$

This binary model of the game defines a CSP, which we can solve to find hints for the player's next move. If we have all the solutions of the problem, we can look at what is common to all those solutions. For instance, if all solutions give $x_{41} = 0$, we know that the square at position $(4,1)$ is a safe move.

But we even go further than this. Assuming that all those solutions have the same probability, we can compute the expected solution, i.e., the mean of all solutions. This gives us a probability for each square to be mined. In case no safe move can be found, the player might use this information to choose its next move.

3 Propagation, Search, and Probabilities

We now present specific information related to the implementation of the inference engines. Each of them provides a way to solve the CSP defined by the current state of the game. Sections 3.1 and 3.2 shows two implementation based on constraint propagation only. Sections 3.3 and 3.4 presents two solvers, and explains how their results are used to compute mine probabilities.

3.1 Simple Propagators

The simplest inference engine uses the binary model of the Minesweeper game, and posts the propagators that trivially implement the constraints of the model. We illustrate this with the example shown in Fig. 1. A quick sketch of the CSP is given at the end of Sect. 2. All those constraints can be implemented with the Oz propagator `FD.sum`, taking a list of `FD` variables with domain `0#1`. For instance, the propagator for (2,5) is created by a statement like

```
{FD.sum [X14 X15 X16 X24 X26 X34 X35 X36] ´=:´ 2}
```

Let us now examine the effect of those propagators. For the sake of simplicity, we assume that the "zero" constraints like (1,1) have been propagated, and we simplify the

remaining constraints using the known values. The constraints are

$$
\begin{array}{ll}
(1,5) & x_{16}+x_{26}=1 \\
(2,5) & x_{16}+x_{26}+x_{36}=2 \\
(3,1) & x_{41}+x_{42}=1 \\
(3,2) & x_{41}+x_{42}=1 \\
(3,3) & x_{42}=1 \\
(3,5) & x_{26}+x_{36}+x_{46}=2 \\
(4,3) & x_{42}+x_{52}=1 \\
(4,5) & x_{36}+x_{46}+x_{56}=1 \\
(5,3) & x_{42}+x_{52}+x_{62}+x_{63}+x_{64}=3 \\
(5,4) & x_{63}+x_{64}+x_{65}=2 \\
(5,5) & x_{46}+x_{56}+x_{64}+x_{65}+x_{66}=2
\end{array}
$$

The propagator for (3,3) immediately infers $x_{42} = 1$, which means that we have found the position of a mine. This information allows propagators (3,1) and (3,2) to infer $x_{41} = 0$, while the propagator (4,3) infers $x_{52} = 0$.

The remaining propagators cannot infer new constraints, and thus wait for more information to come. Still, more information can be deduced from those constraints. But the propagators that we have considered here cannot do it, because they share too few information with each other. For instance, propagators (1,5) and (2,5) could infer $x_{36} = 1$ if they were sharing $x_{16} + x_{26} = 1$ as a basic constraint. This insight leads us to an improvement in the propagation of the constraints.

3.2 The Set Propagators

We now show propagators that infer information about sets of squares, hence the name "set" propagators. We continue with the example shown in Fig. 1. Let us assume that the simple propagators have determined the variables as explained above. We consider the remaining constraints

$$
\begin{array}{ll}
(1,5) & x_{16}+x_{26}=1 \\
(2,5) & x_{16}+x_{26}+x_{36}=2 \\
(3,5) & x_{26}+x_{36}+x_{46}=2 \\
(4,5) & x_{36}+x_{46}+x_{56}=1 \\
(5,3) & x_{42}+x_{52}+x_{62}+x_{63}+x_{64}=3 \\
(5,4) & x_{63}+x_{64}+x_{65}=2 \\
(5,5) & x_{46}+x_{56}+x_{64}+x_{65}+x_{66}=2
\end{array}
$$

Remember that the weakness of the simple propagators was coming from their unability to share information about subterms like $x_{16} + x_{26}$. Consider for instance constraint (2,5). The improved implementation of this constraint will actually create as many propagators as partitions of the set of indices $\{16, 26, 36\}$.

For each subset I of indices, we consider the "set" variable x_I defined by

$$x_I = \sum_{i \in I} x_i.$$

The definition of x_I can be implemented by a simple propagator. We can now express the constraint (2,5) as follows. For each partition $P = \{I_1, \ldots, I_k\}$ of the indices, we create one propagator for the constraint

$$x_{I_1} + \cdots + x_{I_n} = 2,$$

which is equivalent to (2,5). We thus have propagators for

$$
\begin{array}{ll}
(2,5.a) & x_{\{16\}}+x_{\{26\}}+x_{\{36\}}=2 \\
(2,5.b) & x_{\{16,26,36\}}=2 \\
(2,5.c) & x_{\{16\}}+x_{\{26,36\}}=2 \\
(2,5.d) & x_{\{26\}}+x_{\{16,36\}}=2 \\
(2,5.e) & x_{\{36\}}+x_{\{16,26\}}=2
\end{array}
$$

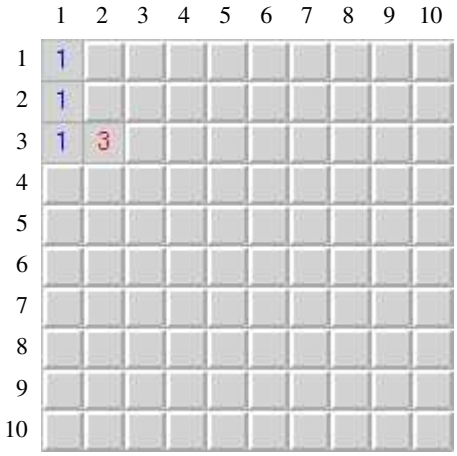

Fig. 2. An example for search.

Note that (2,5.a) has the same effect as the simple propagator for (2,5).

Let us observe the effect of those propagators in the example. The improved propagators for (1,5) infer $x_{\{16,26\}} = 1$, which makes (2,5.e) infer $x_{\{36\}} = 1$, giving $x_{36} = 1$. The simple propagator (4,5) then infers $x_{46} = x_{56} = 0$. The propagation of (3,5) and (1,5) then gives $x_{26} = 1$ and $x_{16} = 0$.

3.3 A Binary Solver

As we said in Sect. 2, useful information can be deduced from the set of solutions of the Minesweeper problem. The issue is, there usually are *many* solutions. Consider the board in Fig. 2, with dimension 10×10, and a total of 20 mines. Four squares have been played. The corresponding CSP has 3.33×10^{18} solutions! Computing all solutions is simply impossible, except for very small boards.

Though, that issue can be addressed. We simply restrict the problem to some of its variables. Each solution of the restricted problem defines a class of solutions of the full problem[1]. In the example, only the variables x_{12}, x_{22}, x_{23}, x_{33}, x_{41}, x_{42}, and x_{43} are relevant. The remaining unknowns can be determined by other simple means. The solutions of the restricted problem are given in Table 1. If we consider the solution s_1 in the table, there remains 89 unknowns, out of which 17 must be mined. The number of ways to choose 17 elements out of 89 is given by the binomial $\binom{89}{17}$. This is the size of the class of solutions defined by s_1. We can now compute the sum of the elements in this class. For instance, the sum of the x_{12}'s is 0, because they are all equal to 0. In the same way, the sum of the x_{22}'s is $\binom{89}{17}$. And the sum of the x_{15}'s is $\binom{88}{16} = \frac{17}{89}\binom{89}{17}$. Computing all restricted solutions, and summing them all, we obtain a probability for each square. We thus find that the probability that $x_{12} = 1$ is 0.318.

[1] The word "class" is used with the meaning of "subset" here. The subset we consider is actually an equivalence class in the set of solutions.

solution	s_1	s_2	s_3	s_4	s_5	s_6	s_7	s_8	s_9
x_{12}	0	0	0	1	1	1	1	1	1
x_{22}	1	1	1	0	0	0	0	0	0
x_{23}	0	1	1	0	1	1	0	1	1
x_{33}	1	0	1	1	0	1	1	0	1
x_{41}	0	0	0	0	0	0	1	1	1
x_{42}	0	0	0	1	1	1	0	0	0
x_{43}	1	1	0	1	1	0	1	1	0
class size	$\binom{89}{17}$	$\binom{89}{17}$	$\binom{89}{17}$	$\binom{89}{16}$	$\binom{89}{16}$	$\binom{89}{16}$	$\binom{89}{16}$	$\binom{89}{16}$	$\binom{89}{16}$

Table 1. Solutions of the restricted binary problem.

3.4 The Set Solver

The binary solver still computes too many solutions. In the example, one can see that the problem has *symmetries*. For instance, each permutation of the values of x_{23}, x_{33}, x_{43} in one solution leads to another solution. This symmetry comes from the fact that those three variables are constrained by $x_{23} + x_{33} + x_{43} = 2$ only.

The improved solver reformulates the CSP in terms of the set variables x_I in order to eliminate those symmetries. Taking all equations that define the binary problem, it computes a partition of the variable's indices. Every subset I in the partition is such that, for each equation $x_J = k$ in the problem, $I \cap J = I$ or $\emptyset$. The subsets are chosen to be *maximal*, so that symmetries are eliminated.

The CSP of Fig. 2 is given by

$$\begin{aligned}
&\textstyle\sum_{i,j\in\{1,\ldots,10\}} x_{ij} = 20\\
&x_{11} = x_{21} = x_{31} = x_{32} = 0\\
(1,1)\quad &x_{12} + x_{21} + x_{22} = 1\\
(2,1)\quad &x_{11} + x_{12} + x_{22} + x_{31} + x_{32} = 1\\
(3,1)\quad &x_{21} + x_{22} + x_{32} + x_{41} + x_{42} = 1\\
(3,2)\quad &x_{21} + x_{22} + x_{23} + x_{31} + x_{33} + x_{41} + x_{42} + x_{43} = 3
\end{aligned}$$

The partition of the indices is

$$P = \big\{\{11\},\{12\},\{21\},\{22\},\{31\},\{32\},\{41,42\},\{23,33,43\},R\big\},$$

which gives the reformulated problem

$$\begin{aligned}
&\textstyle\sum_{I\in P} x_I = 20\\
&x_{\{11\}} = x_{\{21\}} = x_{\{31\}} = x_{\{32\}} = 0\\
(1,1)\quad &x_{\{12\}} + x_{\{21\}} + x_{\{22\}} = 1\\
(2,1)\quad &x_{\{11\}} + x_{\{12\}} + x_{\{22\}} + x_{\{31\}} + x_{\{32\}} = 1\\
(3,1)\quad &x_{\{21\}} + x_{\{22\}} + x_{\{32\}} + x_{\{41,42\}} = 1\\
(3,2)\quad &x_{\{21\}} + x_{\{22\}} + x_{\{31\}} + x_{\{41,42\}} + x_{\{23,33,43\}} = 3
\end{aligned}$$

This problem has two solutions, given in Table 2. Each class of solutions is the product of the possible combinations of the set variables of the reformulated problem. Each valuation $x_I = k$ has $\binom{n}{k}$ solutions, where n is the size of I. Therefore the size of each class is given by a product of binomials. The computation of the probabilities is similar to what the binary solver does. For instance, the mean value of x_{41} is $\left(\frac{0}{2}N_1 + \frac{1}{2}N_2\right)/(N_1 + N_2) = 0.158879$.

solution	s_1	s_2
$x_{\{12\}}$	0	1
$x_{\{22\}}$	1	0
$x_{\{41,42\}}$	0	1
$x_{\{23,33,43\}}$	2	2
x_R	17	16
class size	N_1	N_2

$$N_1 = \binom{1}{0}\binom{1}{1}\binom{2}{0}\binom{3}{2}\binom{89}{17}$$
$$N_2 = \binom{1}{1}\binom{1}{0}\binom{2}{1}\binom{3}{2}\binom{89}{16}$$

Table 2. Solutions of the reformulated problem.

The efficiency is typically one order of magnitude faster compared to the binary solver. Let us illustrate this with an example. Figure 3 shows a snapshot of the application's window. The squares containing a mine have been marked with a black disk. The probabilities are drawn as filled rectangles in the squares. The more a rectangle is filled, the greated the mine probability. A precise value of a probability is shown in the bottom right of the window when the player moves its mouse cursor over a given square. The set solver has computed 6 solutions to find the probabilities, while the binary solver would search for 246 solutions for the same problem!

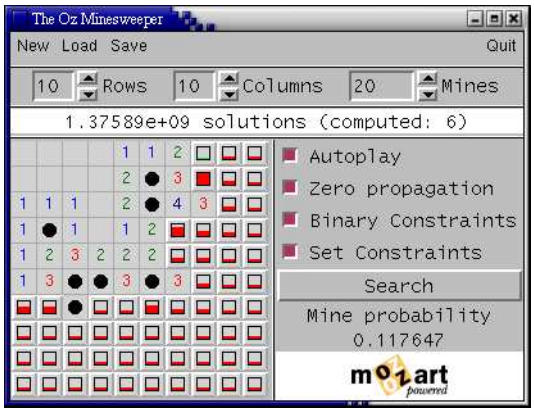

Fig. 3. A snapshot showing probabilistic information.

4 Architecture of the Application

The general architecture of the Oz Minesweeper is depicted in Fig. 4. Boxes refer to concurrent agents (active objects), while "Symbolic field" and "Symbolic constraints" are simply shared data. Arrows from data to agents (resp. from agents to data) correspond to *ask* (resp. *tell*) operations. Arrows between agents represent messages or procedure calls. The removal of the components in the dashed box gives an implementation of the game without digital assistance.

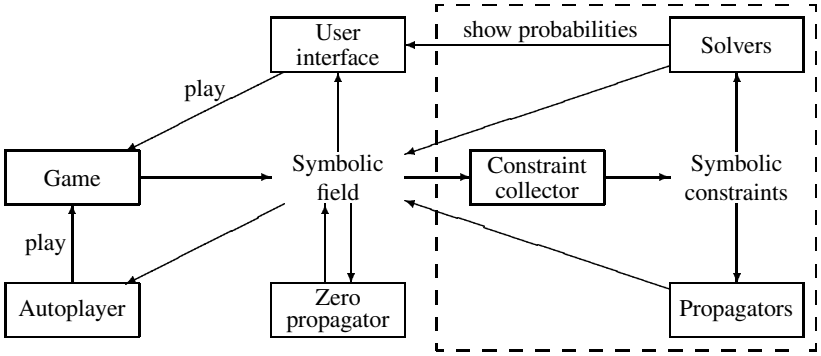

Fig. 4. Dataflow diagram of the Oz Minesweeper.

4.1 The Core Components

The central point in the application is the *symbolic field*, which simply reflects the information known about the mine field. The symbolic field is basically a matrix whose entries can be either `safe(K)` or `mine(X)`. The value `safe(K)` means that the corresponding square is not mined, and `K` gives the number of mines in the surrounding squares. Note that `K` can be unbound, if the square is known to be safe, but has not been played yet. The value `mine(X)` means that the square is mined, and `X` is bound to `exploded` if the mine has exploded, i.e., the game is over.

The *user interface* updates the board by threads that synchronize on the symbolic field. For instance, if an entry in the symbolic field is `safe(K)` and `K` is unbound, the square is marked with a dash "–". This shows the user that this square is safe. As soon as `K` is determined, its value is shown in the square, which becomes inactive. When the user clicks on a square, the user interface calls the *game* agent to play that square. The game automatically tells the result in the symbolic field, which wakes up the thread that updates the square.

4.2 The Zero Propagator and Autoplayer

The *zero propagator* simply asks and tells information in the symbolic field. If a square has no mines around it, which correspond to value `safe(0)` in the symbolic field, the

surrounding squares are told to be safe. The code of the propagator is shown below. The symbolic field appears as the tuple `SymField`. The procedure `WaitEnabled` blocks until the user enables the propagator. The same mechanism is used by all inference engines, and allows to user to experiment with them. The call to function `BoxI` returns the coordinates of all the squares in a box around square `I`.

```
for I in 1..{Width SymField} do
   thread
      case SymField.I of safe(0) then
         {WaitEnabled}
         for J in {BoxI I 1} do SymField.J = safe(_) end
      else skip end
   end
end
```

The *autoplayer* works in a similar way. When enabled, it plays all the squares known to be safe in the symbolic field. So the user can let the various inference engines discover safe moves, and decide whether they should be played automatically.

4.3 The Constraint Inference Engines

The *constraint collector* incrementally builds the *symbolic constraints*, a list of the constraints that appear implicitly in the symbolic field. The inference engines using constraint programming simply read this list to get the constraints of the current problem. A constraint in the list has the form `sum(Is K)`, where `Is` is a list of square coordinates, and `K` is a nonnegative integer. Its semantics is the equation $\sum_{i \in \mathtt{Is}} x_i = \mathtt{K}$. All the constraints in the Minesweeper problem can be written in this way.

Both the simple and set propagators read the symbolic constraints and progressively post simpler propagators as explained in Sect. 3. Those propagators are posted on local binary constrained variables, and told in the symbolic field when determined. The translation for each square is done by a statement like

```
SymField.I = case X.I of 0 then safe(_) [] 1 then mine(_) end
```

Recall that the set propagator for an equation $\sum_{i \in I} x_i = k$ considers all partitions of I, and reformulates the equation as $x_{I_1} + \cdots + x_{I_n} = k$. If I has 8 elements (the typical case in the Minesweeper), we thus have 255 set variables and 4140 set equations! Those numbers are greatly reduced by the actual implementation, which first subtracts the known x_i's, and thus only considers the coordinates of the unknown squares.

A search with a solver is triggered by pushing a button in the user interface. The solver first takes the known part of the symbolic constraints list, and solves the problem given by those constraints. In case new safe moves or mine positions are found, they are told to the symbolic field. Otherwise, the mine probabilities are shown on the board.

5 Evaluation and Related Work

The Oz Minesweeper has been entirely written in Mozart [2], and is only about 1000 lines long. The digital assistant is capable to find all the safe moves in a given situation.

The set propagator proved to be effective at this task, it usually finds most of them. The solver rarely finds new moves, and provides mine probabilities instead. It leaves the player with the toughiest decision, involving a cost-benefit strategy. An interesting observation we have made is that the proportion of mined squares should be around 20% to make the game interesting. A proportion less than 20% makes the problem too easy, while more than 20% quickly makes the game unplayable.

We have found only one other application that solves the Minesweeper problem and computes the mine probabilities, called *Truffle-Swine Keeper* [3]. It seems efficient, but we have found the interaction with the solver not as practical as the Oz Minesweeper.

History. The Oz Minesweeper is born six years ago. It started as a student work, in a seminar course on constraint programming. The goal was to study this strange language Oz, and give a presentation on it. I found that the Minesweeper was a good example of a CSP in the "real world". And I hacked a small solver that was playing the game.

Later I rewrote it as a demonstration program, and gave it a graphical user interface. Only the simple propagator was implemented. It already impressed quite a lot of visitors. The next step was the idea of the binary solver. I wrote several implementations of it, notably by hacking a special search engine. I eventually got the idea of solving the restricted problem, and came up with mine probabilities. I rewrote everything from scratch. The hacked special search engine went to the trash can.

The last step happened one year ago. I understood the issue of the symmetries, and designed the set solver. The set propagator was designed while implementing the set solver. I reworked a bit the implementation, and integrated the inference engines in a proper way. I finally improved the user interface the week before submitting this paper.

6 Conclusion

We have designed and implemented a Minesweeper application with a digital assistant. The latter is based on a simple mathematical model of the Minesweeper game, and various techniques coming from the field of constraint programming. It proved to be effective, and is capable to infer every logical consequence of the problem to solve. It computes mine probabilities without computational burden.

The simplicity and efficiency of our application relies on the language Oz and the platform Mozart. The dataflow concurrency, symbolic data, and constraint system makes the application's architecture modular and elegant.

References

1. Kaye, R.: Minesweeper is NP-complete. Mathematical Intelligencer (2000) See also `http://web.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm` (07/16/2004).
2. Mozart Consortium (DFKI, SICS, UCL, UdS): The Mozart programming system (Oz 3) (1999) Available at `http://www.mozart-oz.org`.
3. Kopp, H.: Truffle-swine keeper (2001) Program available at `http://people.freenet.de/hskopp/swinekeeper.html` (07/16/2004).