Proof Pearl: Revisiting the Mini-rubik in Coq

Laurent Théry

Marelle Project INRIA, France
Laurent.Thery@inria.fr

Abstract. The Mini-Rubik is the 2x2x2 version of the famous Rubik's cube. How many moves are required to solve the 3x3x3 cube is still unknown. The Mini-Rubik, being simpler, is always solvable in a maximum of 11 moves. This is the result that is formalised in this paper. From this formalisation, a solver is also derived inside the Coq prover. This rather simple example illustrates how safe computation can be used to do state exploration in order to derive non-trivial properties inside a prover.

1 Introduction

A recent paper [5] has shown that 26 moves are sufficient to solve Rubik's cube. This is the best-known upper bound (the exact value is conjectured to be around 20 moves). It uses some clever approximation of the problem but relies mainly on heavy parallel computations: 8000 CPU hours are needed to get the result. In this paper, we tackle the more elementary Mini-Rubik. Instead of 26 small cubes, the Mini-Rubik is composed of 8 small cubes only. It is a well-known result that it is always solvable in a maximum of 11 moves [1]. This is the result we formalise in this paper.

In Coq [8], there is no native data-structure. All basic types such as integer, boolean, string are tree-like structures built using the standard `Inductive` command. For example, the natural numbers use Peano representation with two constructors `S` and `O`. The natural number 3 is then internally represented as `(S (S (S O)))`. Although this is perfectly adequate for proofs (for example, the usual inductive principle is given for free), Peano numbers are useless for computing. With a binary representation, the situation gets slightly better but is still not satisfactory. In [7], a generic mechanism is proposed for associating a dedicated data-structure for computing to the standard one for proving while preserving all the nice properties of the type system. This mechanism is applied to integer arithmetic in the following way. First, a special type `Int31` is defined that contains a single constructor with a list of 31 booleans as arguments. This is the reference data-structure. Then, this type is associated to the internal 31-bit Ocaml integers[1] in a straightforward manner. So, computing within the `Int31` type directly benefits from the processor arithmetic with the corresponding speed-up. For example, the `Int31` type was used in [9] to get the primality of some large numbers using elliptic curves.

[1] In Ocaml [6], the last bit of a word indicates if it should be interpreted as either a value or a pointer, the arithmetic has then only 32 - 1 bits.

O. Ait Mohamed, C. Muñoz, and S. Tahar (Eds.): TPHOLs 2008, LNCS 5170, pp. 310–319, 2008.

In this paper, we are going to use the `Int31` type in a slightly different manner. The Mini-Rubik has 3,674,160 possible configurations. In order to get our final result, we need to visit all these configurations while recording those which have already been encountered. It amounts to manipulating subsets of a set of size 3,674,160. In order to use as little memory as possible, we use the `Int31` type to encode subsets of small sets of size 31. We then represent a subset of the configurations with 118,522, i.e. 3,674,160/31, of these `Int31`. This capability of encoding small subsets in a single word is the key aspect that makes our formalisation work.

The paper is organised as follows. In Section 1, we present a naive formalisation that is used to get the basic properties of the problem. In Section 2, we introduce a second formalisation whose main concern is memory consumption. The main result is obtained from this second formalisation. Finally, in Section 3, we give more details about our formalisation.

2 Direct Formalisation

To represent the Mini-Rubik inside Coq, we have chosen a model that is particularly well-suited for the computation we need to perform. If we believe that one can relatively easily get convinced that we have modelled the Mini-Rubik faithfully, ultimately we should also provide a more intuitive model and formally prove the equivalence with our model.

In our model, the front-upper-left corner remains fixed. So, a configuration needs to take into consideration 7 small cubes only. Information about a small cube is split in two: its position and its orientation. Small cubes are numbered from 1 to 7. The cube on the left of Figure 1 shows which ordering has been used to label the small cubes: the fixed cube and cubes 1,2, 3 compose the front face. Small cubes are represented by elements of the enumerate type `cube`:

```
Inductive cube: Set := C1 | C2 | C3 | C4 | C5 | C6 | C7.
```

Each small cube is rigid and has 3 coloured faces only. Choosing arbitrarily the vertical direction, in any configuration, a small cube has exactly one face that belongs to either the top face or the bottom face. Knowing the colour of this face is sufficient to deduce the colours of the other faces. This means that a cube has only 3 possible orientations that can be represented by elements of the enumerate type `orientation`:

```
Inductive orientation: Set := O1 | O2 | O3.
```

Note that in the following we really manipulate positions and orientations as if they were natural numbers. We have not been using directly natural numbers for efficiency reason only. With enumerate types, checking if a position is `C7` is done by one elementary pattern matching, while checking if a number is 7, i.e. `(S (S (S (S (S (S (S O)))))))`, requires a more expensive pattern matching.

A configuration of the Mini-Rubik is represented by a constructor `State` with 7 positions and 7 orientations:

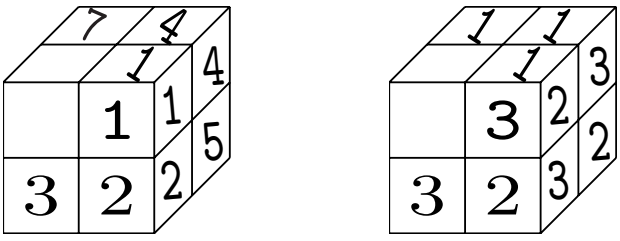

Fig. 1. Positions and Orientations of the Mini-Rubik

```
Inductive state: Set :=
  State (p1 p2 p3 p4 p5 p6 p7: cube) (o1 o2 o3 o4 o5 o6 o7: orientation).
```

where $\mathtt{p}_i$ indicates which small cube is at position i and $\mathtt{o}_i$ gives its orientation. We define the initial orientation in such a way that the initial state of the cube is represented as:

```
Definition init_state = State C1 C2 C3 C4 C5 C6 C7 O1 O1 O1 O1 O1 O1 O1.
```

The front-upper-left corner being fixed, there are only three elementary rotations of the cube: right, back and down. Following Figure 1, cubes 1-4-5-2 compose the right face, cubes 4-5-6-7 the back face and cubes 2-5-6-3 the bottom face. Note that our decision to have the front-upper-left corner fixed means that in our model a rotation of the left face (resp. front and up) is simulated by a rotation in the opposite direction of the right face (resp. back and down). Also, half-turns and anti-clockwise rotations are obtained by iterating twice, resp. thrice, the respective elementary rotation. All this is rather standard.

Orientations are the less intuitive part of our model. The cube on the right of Figure 1 tries to explain how orientations work. Orientations are numbered from 1 to 3 following the clockwise order. For each cube, we arbitrarily decide that it is the face that belongs to the top face (or the bottom face) that holds the orientation. So, from the definition of `init_state`, it follows that initially the top and bottom faces contain only 1, i.e $\mathtt{O}_1$.

After a rotation, the orientation of a cube is either unaffected, modified in a clockwise manner, or modified in an anti-clockwise manner. Clockwise and anti-clockwise changes are represented by the functions `up` and `down` respectively:

```
Definition   up o = match o with O1 ⇒ O2 | O2 ⇒ O3 | O3 ⇒ O1 end.
Definition down o = match o with O1 ⇒ O3 | O2 ⇒ O1 | O3 ⇒ O2 end.
```

The three elementary rotations are modelled as functions from `State` to `State`:

```
Definition rright s := match s with
State p1 p2 p3 p4 p5 p6 p7 o1 o2 o3 o4 o5 o6 o7 ⇒
State p2 p5 p3 p1 p4 p6 p7 (up o2) (down o5) o3 (down o1) (up o4) o6 o7
end.
```

```
Definition rback s := match s with
State p1 p2 p3 p4 p5 p6 p7 o1 o2 o3 o4 o5 o6 o7 ⇒
State p1 p2 p3 p5 p6 p7 p4 o1 o2 o3 (up o5) (down o6) (up o7) (down o4)
end.

Definition rdown s := match s with
State p1 p2 p3 p4 p5 p6 p7 o1 o2 o3 o4 o5 o6 o7 ⇒
State p1 p3 p6 p4 p2 p5 p7 o1 o3 o6 o4 o2 o5 o7
end.
```

Note that our decision to use the top and bottom faces to read orientations is reflected by the fact that the down rotation does not modify any orientation.

A state is reachable if it can be reached from the initial state using the three elementary rotations. This is easily defined inductively by:

```
Inductive reachable: state → Prop :=
  reach0: reachable init_state
| reachr: ∀s, reachable s → reachable (rright s)
| reachb: ∀s, reachable s → reachable (rback s)
| reachd: ∀s, reachable s → reachable (rdown s).
```

The fact that 11 moves are sufficient to solve the Mini-Rubik is true for the half-turn metric. This means that not only elementary rotations need to be considered but also anti-clockwise rotations and half turns. This is done in the `move` relation:

```
Definition move (s1 s2: state) :=
s2 = rright s1 ∨ s2 = rright (rright s1) ∨ s2 = rright (rright (rright s1))
  ∨ s2 = rback s1 ∨ s2 = rback(rback s1) ∨ s2 = rback(rback(rback s1))
  ∨ s2 = rdown s1 ∨ s2 = rdown(rdown s1) ∨ s2 = rdown(rdown(rdown s1)).
```

Once moves are defined, the reachability in n moves is defined inductively as:

```
Inductive nreachable: nat → state → Prop :=
  nreach0: nreachable 0 init_state
| nreachS: ∀n s1 s2, nreachable n s1 → move s1 s2 → nreachable (S n) s2.
```

We also define the property of being reachable in *less than* n moves and the property of being reachable in *exactly* n moves:

```
Definition nlreachable n s := ∃m, m ≤ n ∧ nreachable m s.
Definition nsreachable n s :=
  nreachable n s ∧ ∀m, m < n → ¬ nreachable m s.
```

Now, the theorem we want to prove can be expressed as:

```
Lemma reach11: ∀s, reachable s → nlreachable 11 s.
```

Turning this lemma into a computational problem is quite direct. For each n, we are going to compute the states that are reachable in less than n moves and the states that are reachable in exactly n moves. We represent states by a simple list of states. On such a list, the function `in_states` checks if a state belongs to the list. We first define the list of all possible moves

```
Definition movel :=
 rright :: rright o rright :: rright o rright o rright ::
 rback  ::  rback o rback  ::  rback o  rback o rback  ::
 rdown  ::  rdown o rdown  :: rdown  o  rdown o rdown  :: nil.
```

All the states that are reachable in exactly $n+1$ states are included in the states that are within one move of states that are reachable in exactly n states. This is the basic idea of the algorithm. The function `nexts` does this computation for a single state:

```
Definition nexts (ps: states * states) s :=
  fold_left
    (fun (ps: states * states) f ⇒
      let (states, nstates) := ps in
      let s₁ := f s in
      if in_states s₁ states then ps else (s₁ :: states, s₁ :: nstates))
    movel ps.
```

where `fold_left` is the tail recursive version of the usual iterative `fold` function on lists. For each state $\mathtt{s}_1$ that is one move from `s`, the `nexts` function checks if $\mathtt{s}_1$ has already been visited. If not, it is added to the list of visited states (the first element of the pair) and to the list of the new states (the second element of the pair). Finally, to get the states that are reachable in less than n moves and the states that are reachable in exactly n moves, we just need to iterate the `nexts` function starting from the lists composed of the initial state only:

```
Function iters_aux n (ps: states * states) :=
  match n with
       0 ⇒ ps
  | S n₁ ⇒ let (m,p) := ps in iters_aux n1 (fold_left nexts p (m,nil))
  end.
Definition iters n := iters_aux n (init_state::nil, init_state::nil).
```

It is relatively easy to show that if the second element of the pair returned by `(iters n)` is the empty list, the Mini-Rubik is solvable in `n-1` moves. This is formally stated by the following theorem:

```
Lemma iters_final: ∀n,
  match iters n with
    (_, nil) ⇒ ∀s, reachable s → nlreachable (pred n) s
  |       _  ⇒ True
  end.
```

It is by applying this theorem that we turn the proof of the theorem `reach11` into computing `(iters 12)`.

3 Optimising Memory Consumption

The implementation of `iters` is far too naive to let us prove the `reach11` theorem. Computing `(iters 5)`, which involves 12,224 states only, is already

impossible inside Coq. Nevertheless, `iters` is useful as a reference implementation to which our optimised version is going to be proved equivalent. What `iters` actually does is to compute the diameter of the Cayley graph of the group generated by the three elementary rotations. As explained in [1], having a compact representation in memory of the graph is mandatory to perform this computation. If we go back to how configurations have been encoded, the values of the 14 arguments of `State` are strongly constrained. First of all, the seven arguments (p_1, ..., p_7) which represent positions must be a permutation of (C_1, C_2, C_3, C_4, C_5, C_6, C_7). Also, the orientation of the last cube can be guessed from the orientations of the other cubes. These constraints are captured by the predicate `valid_state`:

```
Definition valid_state s := match s with
State c1 c2 c3 c4 c5 c6 c7 o1 o2 o3 o4 o5 o6 o7 ⇒
  perm (C1::C2::C3::C4::C5::C6::C7::nil) (c1::c2::c3::c4::c5::c6::c7::nil)
  ∧ o1 ⊕ o2 ⊕ o3 ⊕ o4 ⊕ o5 ⊕ o6 ⊕ o7 = O1
end.
```

where `perm` is the permutation predicate between two lists and $\oplus$ is the projection of the addition modulo 3 to the orientation, i.e. adding O_n is done by applying the function `up` $n-1$ times. The `valid_state` predicate is proved to hold for the initial state and to be preserved by the reachability predicate. So, we have:

```
Lemma reachable_valid: ∀s, reachable s → valid_state s.
```

Note that this theorem already indicates that there are at most $7!3^6 = 3{,}674{,}160$ configurations ($7!$ is the contribution of the permutations, and the 3^6 corresponds to the fact that the value of the last orientation is determined by the value of the other orientations). Later, we explain how we formally prove that this is actually the exact number of configurations.

An accurate encoding of permutations of length n should take into consideration the facts that the first element of the permutation has n possible values, the second element $n-1$ and so on. This is done with the following two functions that manipulate permutations as lists:

```
Function encode_aux l p :=
  match l with
      nil   ⇒ nil
  | m :: l1 ⇒ (if p <c m then downc m else m) :: encode_aux l1 p
  end.
Function encode l n:=
  match l, n with
    m :: l1, (S n1) ⇒ m :: encode (encode_aux l1 m) n1
  |    _    ,   _    ⇒ nil
  end.
```

where $<_c$ and $\texttt{down}_c$ are the projections of the comparison and the predecessor functions from the natural numbers to the enumerate type `cube`, i.e. $C_2 <_c C_3$ and $\texttt{down}_c\ C_3 = C_2$. For the definition of the `encode` function, as the recursion is

not structural, an extra argument `n` is required to ensure termination. It bounds the length of the resulting list. With this encoding on a permutation of length n, the i^{th} element is ensured to be in $\{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_{n-i+1}\}$. In particular, the last element is always $\mathtt{C}_1$ and can be discarded. If n is the length of the permutation we want to encode, the extra argument of the `encode` function is $n-1$. For example, if we consider the permutation of the initial configuration, its encoding is computed by `(encode (`$\mathtt{C}_1\mathtt{::}\mathtt{C}_2\mathtt{::}\mathtt{C}_3\mathtt{::}\mathtt{C}_4\mathtt{::}\mathtt{C}_5\mathtt{::}\mathtt{C}_6\mathtt{::}\mathtt{C}_7$`::nil) 6)` and evaluates to $\mathtt{C}_1\mathtt{::}\mathtt{C}_1\mathtt{::}\mathtt{C}_1\mathtt{::}\mathtt{C}_1\mathtt{::}\mathtt{C}_1\mathtt{::}\mathtt{C}_1\mathtt{::nil}$. To sum up, the information about positions in a state can be encoded by 6 elements of type `cube (`$\mathtt{p}'_1, \mathtt{p}'_2, \mathtt{p}'_3, \mathtt{p}'_4, \mathtt{p}'_5, \mathtt{p}'_6$`)` with $\mathtt{p}'_i \in \{\mathtt{C}_1, \mathtt{C}_2, \ldots, \mathtt{C}_{8-i}\}$ and the information about orientations can be encoded by 6 elements of type `orientation (`$\mathtt{o}'_1, \mathtt{o}'_2, \mathtt{o}'_3, \mathtt{o}'_4, \mathtt{o}'_5, \mathtt{o}'_6$`)`. Furthermore, as the `iters` function intensively uses encoding and decoding of permutations, we actually use co-inductive types, which are evaluated lazily, to get the memoisation of these operations. This speeds up our computation by a factor of 2.

We use decision trees to represent sets of states. The 12 elements that encode a state (`p`$'_1$, `p`$'_2$, `p`$'_3$, `p`$'_4$, `p`$'_5$, `p`$'_6$, `o`$'_1$, `o`$'_2$, `o`$'_3$, `o`$'_4$, `o`$'_5$, `o`$'_6$) are used to denote a path to a boolean leaf in the decision tree. If this leaf is `true`, the state is in the set. In Coq, a constructor with n arguments allocates $(n+1)$ 32-bit words. It is then better to have the elements with the largest number of arguments at the bottom of the tree structure. This is why the reordering of the path (`p`$'_6$, `o`$'_1$, `o`$'_2$, `o`$'_3$, `o`$'_4$, `o`$'_5$, `o`$'_6$, `p`$'_5$, `p`$'_4$, `p`$'_3$, `p`$'_2$, `p`$'_1$) is favoured. Furthermore, instead of boolean leaves, we can use elements of the `Int31` type to encode sets of 31 elements with the usual convention that the i^{th} element of the set is present if and only if the i^{th} bit is set to one. In our path, `p`$'_2$ has 6 possible values and `p`$'_3$ has 5 possible values, this means that the pair $(\mathtt{p}'_2, \mathtt{p}'_3)$ has 30 possible values which can be effectively represented by a single `Int31` element (a single bit is then unused). The actual path that is used is then (`p`$'_6$, `o`$'_1$, `o`$'_2$, `o`$'_3$, `o`$'_4$, `o`$'_5$, `o`$'_6$, `p`$'_5$, `p`$'_4$, `p`$'_1$, $5(\mathtt{index}(p'_2)-1)+\mathtt{index}(p'_3)-1$) where `index` is the function that maps elements of the type `cube` to natural numbers, i.e. $\mathtt{index}(\mathtt{C}_5) = 5$. With this encoding, the last element of a path is always a natural number strictly less than 30. Two functions `encode_state` and `decode_state` are defined to relate states and paths and their composition is proved to be the identity on valid states. With this representation, a set of states requires a maximum of 295,001 32-bit words which means 2.6 bits per configuration.

It is also possible to derive a solver by slightly modifying the `iters` program. This follows from the observation that, given a state s that is reachable in n moves, the states which are one move from s are reachable in $n-1$, n, or $n+1$ moves. Out of all these states, a solver just needs to be capable to pick one state that is reachable in $n-1$ move. Since for any n, $(n-1)$ `mod` 3, n `mod` 3, and $(n+1)$ `mod` 3 are always 3 distinct values, it is sufficient to be able to associate for each state s the two-bit value n `mod` 3 where n is the number of moves that are necessary to reach s. For this, we just need to split in two the states that are reachable in less than n moves to get the two-bit information. The modified version `iter`$_2$`s` of the function `iters` for the solver is the following:

```
Definition next2s m (ps: states * states * states) s :=
  fold_left
    (fun (ps: state * states * states) f ⇒
      let (states1, states2, nstates) := ps in
      let s1 := f s in
      if (in_states s1 states1 || in_states s1 states2) then ps
      else match m with
             0 ⇒ (s1::states1, states2, s1::nstates)
           | 1 ⇒ (states1, s1::states2, s1::nstates)
           | _ ⇒ (s1::states1, s1::states2, s1::nstates)
           end)
    movel ps.
Function iter2s_aux (n m: nat) (ps: states * states * states) :=
  match n with
        0 ⇒ ps
    | S n1 ⇒ let (ps1,ps2,ps3) := ps in
    iter2s_aux n1 ((m+1) mod 3) (fold_left (next2s m) ps3 (ps1,ps2,nil))
  end.
Definition iter2s n :=
  iter2s_aux n 1 (init_state::nil, nil, init_state::nil).
```

4 Running the Solver

The complete formalisation is available at

`ftp://ftp-sop.inria.fr/marelle/Laurent.Thery/Rubik.zip`

It is composed of 7000 lines of code: 3000 lines for the naive formalisation, 4000 for the optimised version. On a Pentium 4 with 1 Gigabyte of RAM, getting the `reach11` theorem takes 260 seconds. Most of the time is spent in computing `(iters 12)`. Note that, once this computation has been performed, it can also be used to get another interesting result:

```
Lemma valid11: ∀s, valid_state s → reachable s.
```

This proves that the number of configurations of the Mini-Rubik is exactly 3,674,160. This is done by checking that the first element of the pair computed by `(iters 12)` with the optimised version has all its leaves equal to $2^{30} - 1$.

The solver returns the list of moves in the half-turn metric that leads to the initial state. We use co-inductive types and memoisation to compute only once the table that associates each state with its index of reachability modulo 3. So, the first time the solver is called, the table is actually computed:

```
Time Eval compute in solve init_state.
     = nil
Finished transaction in 384. secs (384.562537u,0.292956s)
```

The next invocations are then immediate. For example, we can try to swap two adjacent corners

```
Time Eval compute in solve (State C2 C1 C3 C4 C5 C6 C7 O1 O1 O1 O1 O1 O1 O1).
  = Right::Back^-1::Down^2::Right^-1::Back::Right^-1::Back^-1::Right::Down^2::
      Right::Back::nil
Finished transaction in 0. secs (0.00100000000009u,0.s)
```

or two opposite corners

```
Time Eval compute in solve (State C7 C2 C3 C4 C5 C6 O1 O1 O1 O1 O1 O1 O1 1).
  = Right::Back^-1::Right^2::Back^-1::Right^-1::Down^-1::Right::Down^2::Back::
      Down^-1::Back::nil
Finished transaction in 0. secs (0.00100000000009u,0.s)
```

5 Conclusions

Proof systems like Coq are not well-suited for dealing with state exploration. Mike Gordon has already shown in [3] how one can benefit from an external link to a Bdd package to solve a solitaire game inside the Hol prover [4]. In our work, everything has been done withing the theorem prover using safe computation. The main contribution of this paper is to show that we can actually use this safe computation to effectively model problems of relatively large size like the Mini-Rubik. As in [3], what we really gain by doing this inside a prover is the formal connection between what we want to prove (the model) and what we actually compute.

The key aspect of the formalisation is its memory consumption. Most of the issues we have addressed here is not specific to theorem proving and can also be found in the model checking community. For example, in [10], the author shows how a careful design is necessary to be able to solve this problem with Bdds. Having a certified formalisation in a purely functional setting that uses 2.6 bits only per configuration is rather satisfactory. The 260 seconds to complete the exploration are less satisfactory but it is difficult to see how we could go significantly faster in a programming language without side-effects. Finally, if our decision trees are for the moment ad-hoc for the specific configurations of the Mini-Rubik, deriving a generic library that uses `Int31` to represent large finite sets could be useful for other formalisations.

A natural continuation of this work would be to tackle the full Rubik's cube. Obviously, formalising results like [5] is outside reach but getting simpler bounds like the one of 52 moves [2] seems feasible.

References

1. Cooperman, G., Finkelstein, L.: New Methods for Using Cayley Graphs in Interconnection Networks. Discrete Applied Mathematics 37(38), 95–118 (1992)
2. Frey, A.H., Singmaster, D.: Handbook of Cubik Math. Enslow Publishers (1982)
3. Gordon, M.J.C.: Reachability Programming in HOL98 using BDDs. In: Aagaard, M.D., Harrison, J. (eds.) TPHOLs 2000. LNCS, vol. 1869, pp. 179–196. Springer, Heidelberg (2000)

4. Gordon, M.J.C., Melham, T.F.: Introduction to HOL: a theorem proving environment for higher-order logic. Cambridge University Press, Cambridge (1993)
5. Kunkle, D., Cooperman, G.: Twenty-Six Moves Suffice for Rubik's Cube. In: ISSAC 2007, pp. 235–242 (2007)
6. Leroy, X.: Objective Caml (1997), `http://pauillac.inria.fr/ocaml/`
7. Spiwack, A.: Efficient Integer Computation in Type Theory, Draft paper (2007)
8. The Coq development team. The Coq Proof Assistant Reference Manual v7.2. Technical Report 255, INRIA (2002), `http://coq.inria.fr/doc`
9. Théry, L., Hanrot, G.: Primality proving with elliptic curves. In: Schneider, K., Brandt, J. (eds.) TPHOLs 2007. LNCS, vol. 4732, pp. 319–333. Springer, Heidelberg (2007)
10. Valmari, A.: What the small Rubik's cube taught me about data structures, information theory, and randomisation. International Journal for Software Tools Technology 8(3), 180–194 (2006)