International Journal on Software Tools for Technology (2006) 8(3): 180–194
DOI 10.1007/s10009-005-0191-z

SPECIAL SECTION ON TOOLS AND ALGORITHMS FOR THE CONSTRUCTION AND ANALYSIS OF SYSTEMS

Antti Valmari

What the small Rubik's cube taught me about data structures, information theory, and randomisation

Published online: 16 February 2006

Abstract This article discusses observations made when the state space of the 2×2×2 Rubik's cube was constructed with various programs based on various data structures, gives theoretical explanations for the observations, and uses them to develop more memory-efficient data structures. The cube has 3,674,160 reachable states. The fastest program runs in 20 s and uses 11.1 million bytes of memory for the state set structure. It uses a 31-bit representation of the state and also stores the rotation through which each state was first found. Its memory consumption is remarkably small, considering that 3,674,160 × 31 bits is about 14.2 million bytes. Getting below this number was made possible by sharing common parts of states. Obviously, it is not possible to reduce memory consumption without limit. We derive an information-theoretic hard average lower bound of 6.07 million bytes that applies in this setting. We introduce a general-purpose variant of the data structure and end up with 8.9 million bytes and 48 s. We also discuss the performance of BDDs and perfect state packing in this application.

Keywords Explicit state spaces

1 Introduction

In spring 2001 I was looking for an example with which to teach state space construction algorithms without first having to give a series of lectures on the background of the example. The traditional 'dining philosophers' seemed neither motivating nor challenging enough.

It occurred to me that the 2 × 2 × 2 version of the familiar *Rubik's cube*[1] might have a suitable number of states. I will soon show that it has 3,674,160 states. This number is low enough that I expected its state space to fit in the main memory of cheap computers of the time yet high enough that differences between bad and good state space construction algorithms would become clear.

The 2 × 2 × 2 Rubik's cube proved to be a more fruitful example than I could ever have guessed. Working with it provided surprises about the C++ standard library as well as information-theoretical lower limits and led to the development of a very tight hash table data structure. In this paper I will present some highlights of that enterprise.

Lower limits we are discussing can be used for answering questions like 'The computer has 1 GiB memory, and 100 bits are needed to store a state. What is the largest number of different states that fits in the memory?' Perhaps surprisingly, the answer is not 1 GiB/100 bits ≈ 86,000,000, but roughly 115,000,000. This is a theoretical figure that is probably not obtainable in practice. On the other hand, the very tight hash table is usable in practice, and with it one can make about 100,000,000 states fit in.

2 The state space of the cube

The 2 × 2 × 2 Rubik's cube is illustrated in Fig. 1. Thanks to an ingenious internal structure, any half of the cube – top, bottom, left, right, front, back – can be rotated relative to the opposite half. Thus each of the eight corners is actually a moving piece. Figure 1 shows a rotation of the top half.

Each face of the cube is painted in a unique colour. When the faces are rotated, the colours get mixed. Getting the colours back to their original positions requires some experience, although is not extremely difficult with the 2 × 2 × 2 cube. The 'standard size' of the cube is 3 × 3 × 3.

To construct the state space of the cube, it helps to normalise the orientation of the cube in space. One might, for instance, always let the red-blue-yellow piece be the right back bottom corner with the red face down. Then the states of the cube consist of the other seven pieces changing their positions and orientations.

A. Valmari (✉)
Tampere University of Technology, Institute of Software Systems, PO Box 553, 33101 Tampere, Finland
E-mail: Antti.Valmari@tut.fi

[1] Rubik's cube is a trademark of Seven Towns Ltd.

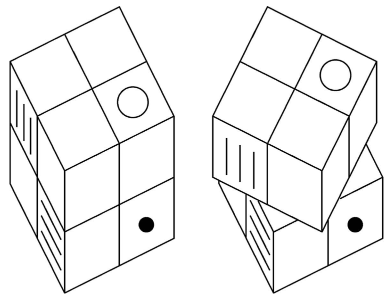

Fig. 1 Rotation of top face of $2 \times 2 \times 2$ Rubik's cube

The corners have altogether $7! = 5{,}040$ different orders. Because each corner can be in three different orientations, the number of states of the cube has the upper limit $7! \times 3^7$. Everyone with enough experience with the cube knows, however, that it is not possible to rotate the cube into a state where everything is as in the original state with the exception of one corner with the wrong orientation. Because of this, the cube has only $7! \times 3^6 = 3{,}674{,}160$ states.

The state of the cube can be represented on a computer in several different ways. For instance, one might encode the colour with three bits and list the colours of the four quarters of the front face, top face, and so on. Because the right back bottom piece does not move, its colours need not be listed. Thus this representation would take $21 \times 3 = 63$ bits.

There are $2^{63} \approx 10^{19}$ bit combinations in 63 bits. This is vastly more than the 3,674,160 states that are possible in reality. Some of the 'extra' bit combinations have a meaningful interpretation. For instance, about 7 million of them can be produced by breaking the cube into pieces and assembling it again, and many more are obtained by repainting the faces of pieces.

The interpretation of the extra bit combinations is not essential, however. What is important is that there are many of them, many more than of 'real' states. This is common in state space applications. As will become clear by the end of Sect. 9, this fact has a great effect on the choice of data structures for representing state spaces.

The bit combinations that the system in question can get into are often called *reachable states*. The set of all bit combinations does not have an established name, but it can be called the set of *syntactic* (or *syntactically possible*) *states*.

The set of syntactically possible states depends on the representation of the states. The state of the $2 \times 2 \times 2$ Rubik's cube can be represented much more densely than was described above. The possible positions of the moving pieces can be numbered $0, \ldots, 6$ and the orientations $0, \ldots, 2$. These can be combined into a code that ranges from 0 to 20 with the expression $3 \times \mathit{position} + \mathit{orientation}$. We shall call this the *pos-or code*.

The left-hand side of Fig. 2 shows the numbering of the positions that I used. The encoding of the orientations is more difficult to explain. In the original state, each piece is in orientation number 0. The right-hand side of Fig. 2 shows how the orientation code changes when the piece moves in an anticlockwise rotation. '+' means increment modulo 3, and '−' denotes decrement. When rotating clockwise, '+' is replaced by '−' and vice versa.

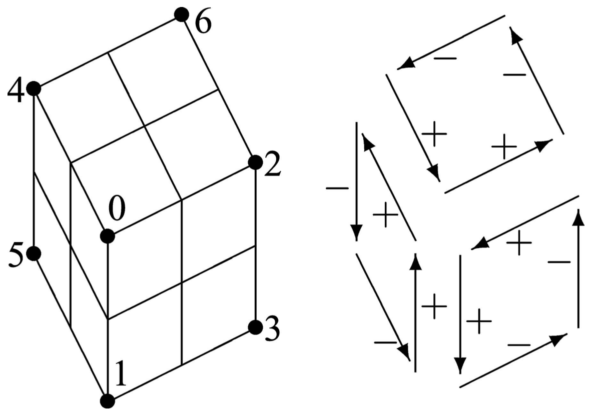

Fig. 2 *Left*: Numbering of positions of pieces. *Right*: Changes in orientations due to rotation

Because each basic rotation increments the orientations of two pieces and decrements those of two other pieces, the sum of orientations modulo 3 stays constant. This proves the fact mentioned earlier that it is impossible to rotate just one corner into a wrong orientation.

The pos-or codes for the seven moving pieces can be combined into a single number with the expression $\sum_{i=0}^{6} c_i \times 21^i$. We will denote this operation by $\mathit{pack}(\langle c_0, c_1, \ldots, c_6 \rangle)$. The resulting number is at most approximately 1.8×10^9 and fits in 31 bits. With this representation, there are only $2^{31} \approx 2.1 \times 10^9$ syntactically possible states. We will also need an unpacking operation $\mathit{unpack}(p) = \langle c_0, \ldots, c_6 \rangle$ such that $c_i = \lfloor p/21^i \rfloor \bmod 21$ for $0 \le i \le 6$.

This representation of states has advantages over the 63-bit representation. It requires less memory, and it is handy that a state fits in the 32-bit word of typical inexpensive computers. Furthermore, we will next see that it facilitates very efficient computation of the effects of rotations. This is important because over 22 million rotations are computed when constructing the state space.

There are six different rotations: one can rotate the top, front or left-hand side face of the cube, and do that either clockwise or anticlockwise. One can reason from Fig. 2 how the position and orientation of any corner piece will change in a given rotation. For instance, rotating the front face anticlockwise moves the piece that is in position 1 and orientation 2 to position 3 orientation 1. Thus the rotation changes the pos-or code of the piece from $(3 \times 1) + 2$ to $(3 \times 3) + 1$, that is from 5 to 10.

Because there are 21 different pos-or codes, any rotation corresponds to a permutation of the set $\{0, \ldots, 20\}$. Table 1 shows these permutations for the anticlockwise rotations as three 21-element look-up arrays. Because, as we saw above, rotating the front face anticlockwise changes the pos-or code from 5 to 10, the '**F**' row of the table contains 10 in the column labelled '5'. The clockwise rotations can be obtained as the inverses of the permutations in the table.

Table 1 The three anticlockwise rotation arrays: **F**ront, **L**eft and **T**op face

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
F	4	5	3	11	9	10	1	2	0	8	6	7	12	13	14	15	16	17	18	19	20
L	13	14	12	1	2	0	6	7	8	9	10	11	17	15	16	5	3	4	18	19	20
T	7	8	6	3	4	5	20	18	19	9	10	11	1	2	0	15	16	17	14	12	13

The total effect of a rotation can be computed by extracting the pos-or codes of each of the seven moving corners from the 31-bit state representation, applying the table seven times to get the new pos-or codes, and packing the results.

3 The first program

My goal was to write a program that would construct the 3,674,160 states of the $2 \times 2 \times 2$ Rubik's cube in breadth-first order. Associated with each state there should be information on the rotation via which the state was first found. Then, given any mixed state of the cube, one can reorder the cube with the smallest possible number of rotations by repeatedly finding the rotation that led to the current state and applying its inverse. The data structure can also be used for investigating the structure of the state space. There will be an example of this in Sect. 11.

The basic principle of such a program is shown in Fig. 3. The 'initial state' is the one where the cube is in order. The program uses two data structures: an ordinary queue and a state structure that stores packed states and associates a number with each stored packed state. The number indicates the rotation by which the state was found. For the initial state, its value does not matter. $R_0, \ldots, R_5$ are the 21-element look-up arrays that represent the rotations.

It is common in state space tools to save memory by inserting in queues pointers to states instead of states proper. However, because a state now fits in one 4-byte word, it uses as little memory as a typical pointer, so no savings would be obtained. On the other hand, storing states as such is quicker. Therefore, the program does it.

I knew that the maximum length of the queue would be less than the number of states, so less than 15 million bytes suffice for the queue. Afterwards I found out that the maximum length was 1,499,111, so a good queue implementation uses less than 6 million bytes.

```
$s := pack($ initial state $)$
add $s$ to the queue; add $s$ with 0 to the state structure
**repeat** until the queue is empty
    remove the first state $p$ from the queue
    $\langle c_0, \ldots, c_6 \rangle := unpack(p)$
    **for** $r := 0$ **to** 5 **do**
        **for** $i := 0$ **to** 6 **do**
            $c'_i := R_r[c_i]$
        $p' := pack(\langle c'_0, \ldots, c'_6 \rangle)$
        **if** $p' \notin$ state structure **then**
            add $p'$ with $r$ to the state structure;
            add $p'$ to the end of the queue
```

Fig. 3 Construction of the reachable states in breadth-first order

The C++ standard library[2] offers the queue as a ready-made data structure. (To be precise, it offers several queues that have the same interface but different implementations.) In the first version of the program I utilised C++ standard library services wherever possible. An efficient queue that meets the needs of my program would have been easy to implement, but I never found a reason to do so.

The situation was different with the state structure. In the first version of my program I used the C++ standard library map. The standard does not fix its implementation, but in practice it is a red-black binary search tree. When I ran the first version of the program on my laptop, the hard disk soon started to rattle, and the printing of intermediate results slowed down. The amount of payload was $3{,}674{,}160 \times 5 \approx 18.4$ million bytes (the fifth byte was the rotation information). Leaving the rotation information out, reducing the payload to 14.7 million bytes, did not help. The laptop I had at that time had 42 million bytes (40×2^{20}, to be more precise) of main memory. Was that not sufficient?

Each node of a red-black tree contains, in addition to the payload, three pointers to other nodes (two children and the parent) and one bit which denotes its colour (red or black). The colour bit is used to maintain the balance of the tree and thus to ensure that tree operations obey the $O(\log n)$ time bound. Each pointer uses 4 bytes of memory. These make up 12 bytes plus one bit. It is customary in some computer architectures to align multibyte data items to addresses that are multiples of two, four, or eight because that speeds up the processing. Because the C++ map is generic, its implementor had probably prepared for the worst and used multiples of eight for the payload. As a consequence, the 31-bit payload expanded to 8 bytes, and then the node expanded to a multiple of 8 bytes, making the total size of a node 24 bytes.

So the state structure needed $3{,}674{,}160 \times 24$ bytes, that is more than 88 million bytes of memory. That was too much even for the workstation back at the office. I felt that that was quite a lot for storing a payload of less than 15 million bytes, so I started to think about a more efficient data structure.

4 An optimised hash table for the $2 \times 2 \times 2$ cube

A *chained hash table* is an efficient and often-used way of representing sets and mappings in a computer. It consists of a fairly large number of linked lists. Each list record

[2] For historical reasons, the data structures that are part of the C++ standard library are often called 'STL'. The abbreviation is misleading, and the C++ standard never mentions it.

typically contains one element, its associated data, and a link to the next record. The list that an element belongs to is determined by the hash function. The number of lists is much smaller than the number of syntactically possible elements. (If the number of syntactically possible elements is so small that one can use an array of that size, then better data structures are available than hash tables. We shall return to this in Sect. 9.)

An element is usually stored in its entirety in its list record. Even so, Knuth had already pointed out that, in principle, that is not necessary. All elements in the same list have some information in common, and that information need not necessarily be stored [5, Sect. 6.4, Problem 13, p. 543]. Isolating and leaving out the shared information is, however, somewhat clumsy, and the savings obtainable are often insignificant compared to the rest of the record. That is, if there are n hash lists, then the savings per record can be at most $\log_2 n$ bits, while already the link to the next record spends more bits, and the record also contains the remaining part of the payload.

The amount of the payload is, however, small in the case of the $2 \times 2 \times 2$ Rubik's cube, making it worthwhile to leave out shared information. One can reduce the memory occupied by the links by putting several elements into each record and by using numbers instead of pointers as links. I eventually ended up with the data structure that is illustrated in Fig. 4. I will call it the *Rubik hash table* and describe it next.

A major requirement of a hash function is that it distribute the elements over the lists as evenly as possible. To ensure this, it is often recommended that the value of the hash function be made dependent on every bit of the element. Therefore, the Rubik hash table starts the processing of an element by randomising its 31-bit representation with simple arithmetic and bit operations. I used the C++ code shown in Fig. 5. I ensured that it never maps two different states into the same mixed state and that every bit affects the least significant bit, but other than that it has not been designed, just written. In Sect. 7 the quality of this algorithm is analysed.

The Rubik hash table uses the 18 least significant bits of the randomised state for choosing a hash list. I will call these bits the *index*. There are thus $2^{18} = 262{,}144$ lists. The remaining 13 bits of the element make up the *entry* that will

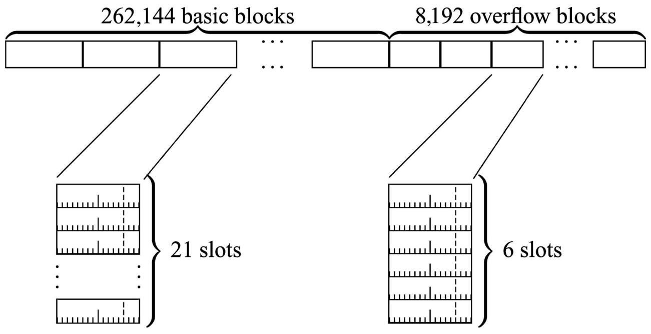

Fig. 4 A hash table optimised for the $2 \times 2 \times 2$ Rubik's cube

```
state ^= state >> bits_in_index;
state *= 1234567; state += 5555555;
state &= 0x7fFFffFF;
state ^= state >> bits_in_index;
state *= 1234567; state += 5555555;
state &= 0x7fFFffFF;
```

Fig. 5 Randomisation of the state

be stored in the list record. The reason for the choices 18 and 13 will become clear after I have described the structure of records.

To keep the number of links down, a list consists of *blocks* of two kinds: *basic blocks* containing 21 entries each and *overflow blocks* with 6 entries each. As I will discuss later, I chose the sizes by experimenting. There is precisely one basic block per list. The number of overflow blocks is 8,192, because that is the greatest number allowed by the structure described below. A slightly smaller number would suffice, but optimising it would not help much because the memory consumed by overflow blocks is insignificant compared to the basic blocks.

To reduce the size of a link, the blocks are stored in an array. The array consists of $(262{,}144 \times 21) + (8{,}192 \times 6)$, that is 5,554,176 two-byte *slots*. A basic block is stored in 21 successive slots, and an overflow block occupies 6 slots. The first slot of the basic block that corresponds to an index value is found by simply multiplying the index by 21. Numbers from 0 to 8,191 are used as links, and the corresponding overflow block is found by multiplying the link by 6 and then adding the total size of the basic blocks.

Each slot consists of 16 bits, three of which are used to denote the status of the slot. The status may be 'empty', 'overflow', or 'in use x', where $0 \leq x \leq 5$. If it is 'in use', then the remaining 13 bits of the slot store the entry of some state, and the value x tells the basic rotation via which the state was first found. In the case of an empty slot, the contents of the remaining 13 bits have no significance. The 13 bits of an overflow slot contain a link; thus the link can obtain at most 2^{13} different values, which is why there are that many overflow blocks.

When new states are added to a hash list, they are stored in the basic block starting from its beginning until it fills up. If still another state has to be stored in the list, then the first free overflow block is used. The last entry of the basic block is moved to the first slot of the overflow block, and the new state is stored in the second slot. The last slot of the basic block is marked 'overflow', and the number of the overflow block is written into its entry field. If necessary, the list is continued into another overflow block, and so on.

Now one can see the reason for the 'magic' numbers 18, 13, and 8,192. To ensure that entries, links, and status data can be easily manipulated, the size of a slot must be an integral multiple of bytes. Three bits suffice for representing the rotation information and handily offer two extra values that can be used to denote 'empty' and 'overflow'. A one-byte slot would leave 5 bits for the entry. The number of lists would be $2^{26} \approx 6.7 \times 10^7$, resulting in the consumption of a lot of memory. Three bytes in a slot would make the lists very long on the average (namely 3,588 states), which would make the scanning of the lists slow and reduce the savings obtained from not storing the index bits. Two-byte slots yield 262,144 lists with an average length of about 14, which are good numbers.

The Rubik hash table uses slightly more than 11 million bytes of memory. The improvement compared to the 88 million bytes used by the C++ map is very good. The program that uses the Rubik hash table runs on my now-ancient laptop with no problems at all. It needs about 105 s to construct and store all 3,674,160 reachable states. The workstation I have at work has changed since the first experiments, so I can no longer take measurements with it. Later on the C++ compiler also changed, speeding the programs up. The workstation I have now can run the program based on the C++ map. It needs 98 s with the current compiler (106 s with the previous one). The same machine runs the Rubik hash table version of the program in about 20 s (23 s).

As mentioned above, I chose the sizes of the basic block and overflow block by experimenting. I do not remember how long it took, but it did not take long. The average length of a list was known in advance: it is the number of states divided by the number of lists, that is about 14. If we make the size of a basic block twice that much, then most of the lists fit in their basic blocks. There are so few overflow blocks that their memory consumption does not matter much. Let us fix their size to be 28, the same as that of the basic blocks. With these values the memory consumption is not much more than 15 million bytes. That fits my ancient laptop well: the program runs in 106 s.

One can then find the optimal size for the basic block by reducing the size until the capacity of the Rubik hash table becomes too small and then do the same for the overflow blocks. Alternatively, after succeeding in generating all reachable states for the first time, one can make the program print the number of lists for each length. The optimal sizes can be computed from these data. We will return to this topic in Sect. 7.

5 An analysis of the memory consumption

When computing the memory consumption of the Rubik hash table I made a startling observation. Although the data structure uses only 11.1 million bytes, it stores 3,674,160 elements of 31 bits, making a total of 14.2 million bytes. What is more, it also stores the rotation information within the 11.1 million bytes. It thus copes with much less memory than what might at first seem to be the amount of information in the data. Unexpectedly small memory consumption is often caused by some regularity in the stored set, but this explanation does not apply here because the states are randomised before storing.

The real explanation is that a set contains less information than the number n of elements in it times the width w of an element, except when $n \leq 1$. This is because a set does not specify an ordering for its contents, and it cannot contain the same element twice. If some elements of a set are already known, learning another one conveys less than w bits of new information, because it was clear beforehand that it cannot be any of the known elements. In a similar fashion, if the order in which the set is listed is known in advance, then the next element cannot be just anything; it must be larger (in that order) than the already listed elements. If the order is not known in advance, then one learns from the list more than just the set: one also learns the order.

Indeed, it is well known that one can store any subset of a set of u elements (the *universe*) in u bits by associating one bit to each element of the set and making it 1 for precisely those elements that are in the subset. When the elements are w bits wide, $u = 2^w$. For large values of n this is less than nw. This *bit array* representation is optimal for arbitrary sets because it uses just as much memory as is needed to give each set a unique bit combination.

However, in addition to being less than nw, the amount of memory used by the Rubik hash table is also far less than 2^w. What is a valid information-theoretic lower bound in this setting?

Before answering this question, let me point out that there is a meaningful sense in which the 14.2 million bytes is a valid point of comparison. Any data structure that stores each element separately as such uses that much memory for the elements, and perhaps also additional memory for pointers and other things used to maintain the data structure. This applies to ordinary hash tables, binary search trees, etc., so one cannot get below 14.2 million bytes with them.

To derive an information-theoretic lower bound that is relevant for the $2 \times 2 \times 2$ Rubik's cube, let us recall Shannon's definition of the information content of an event: $-\log_2 p$, where p is the a priori probability of the event. Probabilities depend on earlier information. In this case, there is one potential event for each possible set of states, namely that the program constructs precisely that set. There are 2^{2^w} different sets of w-bit elements. If each of them were equally likely, then any of them would bring $-\log_2 2^{-2^w} = 2^w$ bits of information. The weighted average would then be $\sum -p_i \log_2 p_i = 2^w$. This is precisely the memory consumption of the bit array representation.

In the case of the $2 \times 2 \times 2$ Rubik's cube, not every set has the same probability. This is because the number of reachable states is known in advance. Any set of the wrong size has probability 0. On the other hand, every set of the correct size is assumed to be equally likely (and that assumption was enforced by randomisation). The number of n-element sets drawn from a universe of u elements is $\frac{u!}{n!(u-n)!}$. This yields the information-theoretic lower bound

$$\log_2 \frac{u!}{n!(u-n)!} \tag{1}$$

bits, where $u = 2^w = 2^{31} = 2{,}147{,}483{,}648$ is the number of all possible 31-bit elements and $n = 3{,}674{,}160$ is the size of the set.

Theorem 1 *Let the* universe *be a fixed set with u elements. For any data structure for storing subsets of the universe, the expected amount of memory needed to store an arbitrary subset is at least u bits. For any data structure for storing subsets of size n, the expected amount of memory needed to store an arbitrary subset of that size is at least* $\log_2 \frac{u!}{n!(u-n)!}$ *bits.*

It would perhaps be a good idea to discuss the nature of the lower bound (Eq. (1)) a bit. (Similar comments apply to Eq. (5), which will be presented soon.) It says that if we first fix u, n, and the data structure, and then pick an *arbitrary* set of size n, the *expected* memory consumption will be at least what the formula says. In other words, if we store each subset of size n in turn and compute the average memory consumption, the result will be at least the bound. The theorem does not claim that no set can be stored in less memory. There may be individual sets which use much less memory. However, such sets must be rare. No matter what the data structure is, if the set is picked at random, then the memory consumption is at least the bound with high probability.

To illustrate this, consider *interval list* representations of sets. The universe can be thought of as consisting of the numbers $0, 1, \ldots, u-1$. An interval list $\langle a_1, a_2, \ldots, a_k\rangle$ is a sequence of elements of the universe in strictly increasing order. a_1 is the smallest element of the universe that is in the set. Any pair (a_{2i-1}, a_{2i}) indicates that all x such that $a_{2i-1} \le x < a_{2i}$ are in the set, while elements between a_{2i} (inclusive) and a_{2i+1} (exclusive) are not in the set. If k is odd, then every x such that $a_k \le x$ is in the set.

The interval list $\langle 0, 3{,}674{,}160\rangle$ represents a 3,674,160-element set of the $2 \times 2 \times 2$ Rubik's cube universe with very little memory. So do, for instance, $\langle 1{,}996{,}325{,}840, 2{,}000{,}000{,}000\rangle$, and $\langle 2{,}143{,}809{,}488\rangle$. However, the interval list representation of the set $\{0, 2, 4, \ldots, 7{,}348{,}318\}$ is $\langle 0, 1, 2, \ldots, 7{,}348{,}319\rangle$ and takes a lot of memory. In general, the interval list representation of a random 3,674,160-element set of the $2 \times 2 \times 2$ Rubik's cube universe consumes a lot of memory. This is because 3,674,160 is small compared to 2^{31}, implying that the successor of an element in the set is rarely in the set. So most elements require the listing of two numbers.

Let us apply our formula to the $2 \times 2 \times 2$ Rubik's cube. 2,147,483,648! is quite unpleasant to evaluate, but we can derive the following approximation for Eq. (1):

Theorem 2 *Let* $w' = \log_2 u - \log_2 n$. *If* $n \ge 1$, *then*

$$nw' \;\le\; \log_2 \frac{u!}{n!(u-n)!} \;<\; nw' + 1.443n. \tag{2}$$

Proof Formula 6.1.38 in [1] (a version of Stirling's formula) says that if $x > 0$, then

$$x! = \sqrt{2\pi}\,x^{x+\frac{1}{2}} e^{-x+\frac{\theta}{12x}}, \tag{3}$$

where $0 < \theta < 1$. Therefore, $\log_2 n! > n\log_2 n - n\log_2 e$, and, furthermore,

$$\begin{aligned}
&\log_2 \frac{u!}{n!(u-n)!}\\
&\quad= \log_2 \frac{u!}{(u-n)!} - \log_2 n!\\
&\quad< \log_2 u^n - n\log_2 n + n\log_2 e \qquad\qquad (4)\\
&\quad= n(\log_2 u - \log_2 n) + n\log_2 e\\
&\quad= nw' + n\log_2 e \;<\; nw' + 1.443n.
\end{aligned}$$

On the other hand,

$$\begin{aligned}
\log_2 \frac{u!}{n!(u-n)!} &= \log_2\left(\frac{u}{n} \times \frac{u-1}{n-1} \times \cdots \times \frac{u-n+1}{n-n+1}\right)\\
&\ge\; \log_2\left(\frac{u}{n}\right)^n = n(\log_2 u - \log_2 n) \;=\; nw'. \qquad \square
\end{aligned}$$

The use of w' makes Eq. (2) look simple. More interestingly, w' has a meaningful interpretation. The number of bits in an element is $w = \log_2 u$. A set can be at most as big as the universe whose subset it is, so $n \le u$ or, equivalently, $w \ge \log_2 n$. Thus each element contains $\log_2 n$ 'obligatory' and $w - \log_2 n = w'$ 'additional' bits. The formula says that for each element, it is necessary to store at least its additional bits plus at most 1.443 bits.

When analysing memory consumption, w' is a more useful parameter than w. This is because n and w' may get values and approach infinity independently of each other, whereas n cannot grow without limit unless w grows without limit at the same time. Therefore, the meaning of such formulas as $O(nw)$ is unclear. The formula $O(nw' + n\log n)$ does not suffer from this problem.

The lower bound given by Eq. (2) is precise when $n = 1$ or $w' = 0$. (The case $w' = 0$ will be discussed in Sect. 9.) When n and w' grow, Eq. (1) approaches the upper bound given in Eq. (2) so that, for instance, Eq. (1) $> nw' + 1.0n$ when $w' \ge 2$ and $n \ge 12$, and Eq. (1) $> nw' + 1.4n$ when $w' \ge 5$ and $n \ge 268$.

In the case of the $2 \times 2 \times 2$ Rubik's cube we have $n = 3{,}674{,}160$, $u = 2^{31}$, and $w' \approx 9.191$, so the upper bound yields 4.884 million bytes. With some more effort we can verify that the precise value is between 4.8826 and 4.8835 million bytes. The upper bound is tightened by approximating $u!/(u-n)!$ with $u^{n/2}(u - \frac{n}{2})^{n/2}$ instead of u^n. The lower bound is obtained by first noticing that the until now ignored terms $\log_2 \sqrt{2\pi}$, $\frac{1}{2}\log_2 n$, and $\frac{\theta}{12n}\log_2 e$ in Eq. (3) contribute less than 13 bits and then replacing $(u-n+1)^n$ for u^n in Eq. (4). As a consequence, the value 4.883 million bytes is precise in all shown digits.

The rotations comprise $\log_2 6^{3{,}674{,}160}$ bits $\approx$ 1.187 million bytes of information, so altogether at least 6.070 million bytes are needed. This is the real, hard information-theoretic lower bound, to which the 11.1 million bytes that were actually used should be compared.

To be very precise, one should also take into account the fact that the complete set of reachable states does not emerge all at once but is constructed by adding one state at a time. One should thus count how many sets of *at most* n elements exist and take the logarithm of that number:

$$\log_2 \sum_{k=0}^{n} \frac{u!}{k!(u-k)!}. \tag{5}$$

However, when n is at most one third of u – as is the case with the $2 \times 2 \times 2$ Rubik's cube – the resulting number differs from Eq. (1) by less than one bit.

Theorem 3 *If $n \leq \frac{u+1}{3}$, then*

$$\log_2 \sum_{k=0}^{n} \frac{u!}{k!(u-k)!} < \log_2 \frac{u!}{n!(u-n)!} + 1.$$

Proof Because $n \leq \frac{u+1}{3}$, we have $3k \leq u+1$ and $2k \leq u-k+1$ when $1 \leq k \leq n$, so

$$\begin{aligned} &\frac{u!}{(k-1)!(u-(k-1))!} \\ &= \frac{k}{u-k+1} \times \frac{u!}{k!(u-k)!} \\ &\leq \frac{1}{2} \times \frac{u!}{k!(u-k)!}. \end{aligned}$$

Thus

$$\begin{aligned} &\sum_{k=0}^{n} \frac{u!}{k!(u-k)!} \\ &< \left(\cdots + \frac{1}{2^2} + \frac{1}{2^1} + \frac{1}{2^0}\right) \times \frac{u!}{n!(u-n)!} \\ &= 2 \times \frac{u!}{n!(u-n)!}. \end{aligned}$$ □

One should keep in mind that the 4.883-million-byte and 6.070-million-byte bounds for the $2 \times 2 \times 2$ Rubik's cube were derived assuming that states are represented with 31 bits. If some other number of bits is used, the bounds will be different.

6 Close to the lower bound, in theory

Let us mention as a curiosity that there is a very simple but woefully slow data structure with which one can get quite close to the theoretical lower bound.

Theorem 4 *When $n \geq 1$, there is a data structure for storing n-element subsets of a universe of u elements whose memory consumption is less than $nw' + 2n$ bits, where $w' = \log_2 u - \log_2 n$.*

Proof The element is divided into an index and an entry just like with the Rubik hash table. The size of the entry is $\lfloor w' \rfloor$ bits. Let us define c as $w' - \lfloor w' \rfloor$. Thus $0 \leq c < 1$ and $c + \lfloor w' \rfloor = w'$.

The data structure is a long sequence of bits that consists of units of two kinds: '1' and '$0b_1b_2 \cdots b_{\lfloor w' \rfloor}$', where the b_i are bits. When the sequence is scanned from the beginning, the kind of each unit – and thus also its length – can be recognised from its first bit. A unit of the form $0b_1b_2 \cdots b_{\lfloor w' \rfloor}$ means that the data structure contains the element $x2^{\lfloor w' \rfloor} + b$, where $b = \sum_{i=1}^{\lfloor w' \rfloor} b_i 2^{\lfloor w' \rfloor - i}$ and x is the number of 1-units encountered so far. The end of the sequence is distinguished by keeping track of the number of units that start with zero: there are altogether n of them.

Let us count the number of bits that the data structure uses. The 0-units contain altogether n starter zeroes and $n\lfloor w' \rfloor$ b_i-bits. No more than $\lceil n2^c - 1 \rceil$ 1-units are needed because that suffices up to the number $(n2^c - 1)2^{\lfloor w' \rfloor} + 2^{\lfloor w' \rfloor} - 1 = n2^{w'} - 1 = u - 1$, which is the largest number in the universe. Thus, altogether $n\lfloor w' \rfloor + n + \lceil n2^c - 1 \rceil < nw' + n(1 + 2^c - c)$ bits are needed. The expression $1 + 2^c - c$ evaluates to the value 2 at the ends of the interval $0 \leq c \leq 1$. Within the interval it evaluates to a smaller value. □

One can make this data structure capable of storing any set of at most n elements by adding so many 1-units to its end that their total number becomes $\lceil n2^c \rceil$. Then the end is distinguished by the number of 1-units. The memory consumption is less than $nw' + 2n + 1$ bits.

7 Distribution of list lengths

Despite its great success in the $2 \times 2 \times 2$ Rubik's cube application, my Rubik hash table seems quite inefficient in one respect. It has room for more than 5.5 million elements, so almost 1.9 million – or one third – of its slots are empty.

The majority of the empty slots are in the basic blocks. Because there are 3,674,160 states and 262,144 lists, the average list length is about 14.02. Even so, I mentioned in Sect. 4 that, despite the overflow mechanism, the reachable states do not fit in the data structure unless the size of the basic block is at least 21. Why must the basic block be so big?

The answer becomes apparent if we investigate the first two columns in Table 2. The number in the column labelled 'len.' indicates the length of a list, and the next number 'meas.' (measured) tells how many lists of that length the data structure contains when all reachable states of the cube have been constructed. We observe that $4{,}971 + 3{,}162 + \cdots + 1 = 12{,}574 > 8{,}192$ lists are longer than 20 states. Therefore, no matter how big the overflow blocks are, the data structure will run out of them if the size of the basic blocks is 20 or less.

One can see from Table 2 that quite a few lists are much longer than the average length. The longest list has

Table 2 Some list length distributions

Len.	Meas.	Theor.	No r.	Len.	Meas.	Theor.	No r.	Len.	Meas.	Theor.	No r.
0	0	0.2	7,621	15	26,055	25,983.5	3,444	30	18	20.0	4,734
1	3	3.0	16,878	16	22,399	22,758.5	4,425	31	9	9.0	4,110
2	22	20.9	20,427	17	18,753	18,758.9	5,441	32	1	3.9	3,396
3	82	97.7	19,144	18	14,783	14,601.4	6,305	33	1	1.7	2,887
4	344	343.0	16,106	19	10,810	10,765.8	6,948	34		0.7	2,327
5	969	962.6	13,740	20	7,489	7,540.0	7,492	35		0.3	1,928
6	2,247	2,251.0	11,556	21	4,971	5,028.6	8,119	36		0.1	1,403
7	4,404	4,511.5	9,132	22	3,162	3,200.9	8,387	37		0.0	1,047
8	7,948	7,910.9	6,682	23	1,997	1,948.7	8,420	38			779
9	12,297	12,328.8	4,498	24	1,156	1,136.8	7,972	39			559
10	17,436	17,290.5	3,005	25	600	636.5	7,617	40			421
11	22,088	22,041.7	2,038	26	360	342.7	7,470	41			306
12	25,380	25,753.9	1,864	27	168	177.6	6,742	42			221
13	28,057	27,773.2	2,022	28	89	88.8	6,149	43			97
14	28,004	27,808.0	2,622	29	42	42.8	5,442	44			83

33 entries, more than twice the average. This observation might entice us to conclude that the algorithm in Fig. 5 does not randomise states well. This is not the problem, however.

The real reason can be illustrated with a small thought experiment. For the sake of argument, assume that the number of lists was 367,416. Then the average list length would be precisely 10. The only way the states could be distributed perfectly evenly on the lists would be if, just before the last state is added, one list contained nine states and the others contained ten each, and the last state fell into the only partly filled list. The probability that the last state will fall into the right list is quite small, namely $\frac{1}{367{,}416} \approx 0.0003\%$. It is thus almost certain that the distribution will not be perfectly flat.

Of course, this phenomenon occurs when inserting any state, not just the last one. It has a strong effect on the distribution of the list lengths. The precise distribution (given below as Eq. (7)) is somewhat clumsy to use, but it can be approximated with the *Poisson distribution*:

$$m_k \approx \frac{n}{h}\frac{h^k}{k!}e^{-h}. \tag{6}$$

In the equation, n is the number of elements, h is the average length of the lists, and m_k tells how many lists are of length k.

Only small values of k are interesting because with large values both sides of Eq. (6) are close to 0. We will use l to denote the number of elements of the universe that would go to the same list if the whole universe were added to the data structure. Thus $l = \frac{uh}{n}$. In our case, $n = 3{,}674{,}160$, $h \approx 14.02$, $u = 2^{31}$, and $l = 8{,}192$. So the assumptions of the following theorem hold.

Theorem 5 *If $k \ll n \ll u$ and $k \ll l \ll u$, then Eq. (6) holds.*

Proof If i elements have fallen into some list and altogether j elements into the other lists, then the next element goes to the list in question with probability $\frac{l-i}{u-i-j}$ and elsewhere with probability $\frac{u-l-j}{u-i-j}$. There are $\frac{n!}{k!(n-k)!}$ ways to throw n elements to the lists such that precisely k elements fall into the list in question. The probability of any one such way is given by a division whose numerator is a product of the numbers $l, l-1, \ldots, l-k+1, u-l, u-l-1, \ldots, u-l-(n-k)+1$ in some order and denominator of the numbers $u, u-1, \ldots, u-n+1$. Therefore, the precise value of the number m_k is

$$\frac{n}{h} \times \frac{n!}{k!(n-k)!} \times \frac{l!}{(l-k)!} \times \frac{(u-l)!}{(u-l-n+k)!} \times \frac{(u-n)!}{u!}. \tag{7}$$

We have $\frac{n!}{(n-k)!} \approx n^k$, $\frac{l!}{(l-k)!} \approx l^k$, $\frac{(u-l)!}{u!} \approx u^{-l}$, and $\frac{(u-n)!}{(u-l-n+k)!} \approx (u-n)^{l-k} = (u-n)^l u^{-k}(1-\frac{n}{u})^{-k}$. Let $x = (\frac{nl}{u})^k(1-\frac{nl/u}{l})^l(1-\frac{n}{u})^{-k}$. Thus $m_k \approx \frac{n}{h}\frac{1}{k!}x$. Because $h = \frac{nl}{u}$, we get $x = h^k(1-\frac{h}{l})^l(1-\frac{n}{u})^{-k} \approx h^k e^{-h}$ because $(1-\frac{n}{u})^{-k} \approx 1$ and $(1-\frac{h}{l})^l \approx e^{-h}$. □

The column 'theor.' (theoretical) of Table 2 contains an estimation of the list length distribution that has been numerically computed with the precise formula (Eq. (7)). The numbers given by the approximate formula (Eq. (6)) differ from the precise numbers by less than 2.3% when length < 35. Judging from the table, the measured numbers seem to be in very good agreement with the theoretical ones.

Figure 6 shows the same data as Table 2, plus some additional data. The curve 'double randomisation' is the same as the column 'Meas.' of the table (the name 'double …' will be explained shortly). The Poisson distribution has been drawn into the figure as horizontal lines, but it differs so little from the precise theoretical distribution that it is almost impossible to see.

No doubt there is some statistical test with which one could verify the conclusion that the measured distribution matches the theory. There is an easier and perhaps even more useful thing to do, however: compute the memory consumption that results from the theoretical distribution and compare it to the measured memory consumption. It is easy to compute from the list length distribution how big the basic and overflow blocks must be to make all reachable states fit in the data structure. (One must take into account that

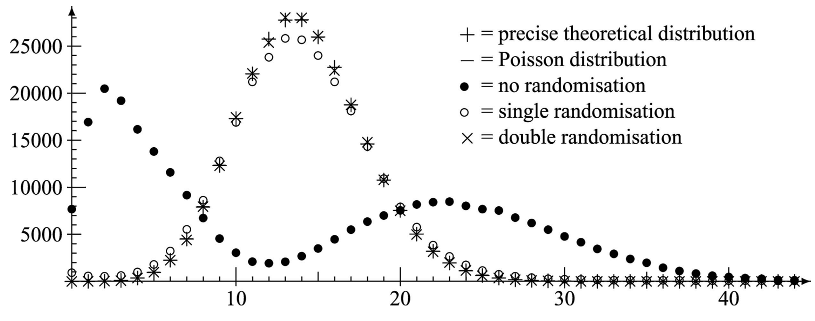

Fig. 6 Some list length distributions

one slot is wasted per overflow block to store the overflow link.) I rounded the theoretical numbers to integers in such a way that the total numbers of lists and states are as close to the true numbers as possible despite the rounding errors.

The theory recommends the block sizes that were actually used and predicts a memory consumption of 11.106 million bytes. The measured figure is 11.105 million bytes. When computing these numbers, I did not include those overflow blocks that were not used. If all 8,192 overflow blocks are taken into account in the figures, as I did earlier, then there is no difference at all between the theoretical and measured memory consumption.

The measured memory consumption is thus very precisely what the theory predicts. This does not quite prove that the measured list length distribution matches the theoretical one, but it does show that it is at least as good: one cannot reduce memory consumption further by using the theoretical distribution. As a consequence, the randomisation algorithm in Fig. 5 need not be thrown away.

Out of curiosity, I also computed the distribution and memory consumption without randomisation of states and with some variants of the randomisation algorithm. The distribution obtained without randomisation is in the column ‘No r.’ (no randomisation) of Table 2, and it is also shown in Fig. 6. List lengths between 45 and 57 have been omitted to save space. It is manifestly different from the theoretical distribution. The size of the basic block would be 34, the overflow block 7, and the memory consumption 17.9 million bytes.

If the randomisation algorithm did only once what the algorithm in Fig. 5 does twice, the memory consumption would still be 11.9 million bytes. Three repetitions would imply memory usage of 11.1 million bytes, and the same holds for four repetitions. We see that one randomisation step is imperfect, but two or more randomise perfectly – at least from the point of view of the $2 \times 2 \times 2$ Rubik’s cube application.

I also tested what would happen if the bit representation of the state were reversed before storing. The index then consists of the originally most significant bits instead of the least significant ones. The distribution without randomisation became weirder still and led to memory consumption of more than 28 million bytes. For instance, there were 178,400 empty lists, the second and third highest peaks were 28,062 at 54 and 8,764 at 42, and the greatest list length was 60. With double randomisation the distribution was again very close to the theoretical distribution, yielding 11.1 million bytes as the memory consumption.

Sometimes one can exploit the skewness of the distribution to save memory. For instance, if the length of the longest list were 16, then one could avoid overflow blocks and cope with about 8.4 million bytes by choosing 16 as the basic block size. Skewness can be exploited only if one can find suitable regularity in the structure of the state space. Sometimes this works out, but often it does not, and when it fails, the result may be dramatically bad, as we can see from the above results on non-randomised distributions. On the other hand, techniques that use randomisation work independently of the nature of the original distribution.

8 Very tight hashing

It is obvious from Table 2 that memory consumption could be reduced further by making the basic blocks smaller and increasing the number of overflow blocks. The size of the overflow link should then be increased. This can be implemented by introducing a new array which is indexed by a block number and returns the extra bits of the overflow link that possibly resides at the end of the block.

I did not implement this kind of program because it would have liberated my data structure of only one artificial restriction caused by the word lengths of contemporary computers. Another restriction is the location of the borderline between the index and the entry: nothing guarantees that the division 18:13 will be the best. It was chosen only because it facilitated handy storage of the data in 8-bit bytes. Instead, I introduced the data structure to my post-graduate student Jaco Geldenhuys and asked if he could develop a general-purpose data structure on the basis of it.

After trying a couple of designs, Jaco and I ended up with the following design, which we call ‘very tight hash table’ or ‘VTH’ [4]. That a slot is empty or stores an overflow link is no longer expressed with unused values of the data that are associated with the key because there are no longer necessarily any associated data, and even if there were, they would not necessarily have unused values. Instead, a counter is attached to each block which indicates how many elements the block contains, and it has one extra

Table 3 Measurements with very tight hash table

Bits in index	List length	Memory (10^6 bytes)	Time (s) laptop	Time (s) workstation
15	112.13	9.2	1160	125
16	56.06	9.0	664	74
17	28.03	8.9	410	48
18	14.02	9.0	286	36
19	7.01	9.4	229	30

value to indicate that the block has overflowed. The counter of an overflow block need not be able to represent the value zero because an overflow block is not used unless it contains something.

Memory for overflow links is reserved in overflow blocks, but not in basic blocks. If a basic block overflows, its overflow link is stored at the start of the data area of the block, and bits that it overwrites are moved to the overflow link field of the last block in the overflow list. In this way the otherwise useless overflow link field of the last block can be exploited.

To find out how big the basic and overflow blocks should be and where the element should be divided into the index and the entry, we first did a large number of different simulation experiments and then measurements with a prototype implemented by Jaco. A good rule of thumb proved to be to use the average list length as the size of the basic block and make the size of the overflow block be three. The average length of the lists should not be too big because scanning long lists takes a long time. Furthermore, the longer the lists are, the shorter the index and the smaller are the savings obtained by not storing the index bits. On the other hand, if the lists are very short, then there are very many of them, which means that the links and unused slots take up a lot of memory. A good average list length is somewhere between 10 and 100.

We analysed the memory consumption of the VTH theoretically and computed values for the constants in the theoretical formula numerically. Among other things, we found that if the average list length is 20, then the memory consumption with a random set is below

$$1.13nw' + 0.04n \log_2 n + 5.05n$$

bits and with an average length of 50 below

$$1.07nw' + 0.02n \log_2 n + 6.12n.$$

These might be compared to the bound (Eq. (2)) $1.00nw' + 1.443n$.

By experimenting with different values for n and w' one can see that unless the number of elements to be stored is very small compared to the number of syntactically possible elements, the memory consumption of the data structure stays below nw, that is the number of elements multiplied by the size of the element, and thus clearly below the memory consumption of the ordinary hash table. For instance, the latter formula gives the result mentioned in the introduction, namely that 1 GiB suffices if $n = 10^8$ and $u = 2^{100}$ (yielding $w' \approx 73.4$). If n is rather small compared to the number of syntactically possible elements ($\leq 5\%$, if $n \leq 10^9$), the memory consumption stays below twice the information-theoretic lower bound.

The VTH is thus clearly an improvement over the ordinary hash table, for instance. On the other hand, it does not leave much room for dramatic further improvements because of the information-theoretic lower bound.

One cannot compute from the formulas how much memory the VTH would use in the case of the $2 \times 2 \times 2$ Rubik's cube because they do not take into account the memory consumption of the rotation information. By extending the element by 3 bits to cover the rotation bits as well, both formulas yield about 9.0 million bytes.

We tested the $2 \times 2 \times 2$ Rubik's cube program with the VTH with the number of bits in the index ranging from 15 to 19. Rotation bits were stored as associated data, not as a part of the element. That is, they did not affect the distribution of elements to lists but were included in memory consumption figures. The results are shown in Table 3. They match the theoretical estimation given above. When the index size grows, memory consumption first decreases and then increases, as was predicted above. Time consumption grows along with the list length. Regarding time consumption, the VTH cannot compete with the Rubik hash table, but it wins in memory consumption.

The VTH and theoretical, simulation, and measured results associated with it have been presented in [4].

9 Perfect packing

By *perfect packing* or *indexing* any compression method is meant whose outputs can be read as numbers in the range $0, \dots, n-1$, where n is the number of reachable states. That is, a packed state consumes just enough memory to give every state a unique packed representation. If states can be packed perfectly, then the set of reachable states can be represented with a very simple, fast, and memory-efficient data structure: a bit array that is indexed with the packed representation of a state and that tells whether the state has been reached. If the states have any associated data, then these data must be stored in a separate array. If the associated data have any unused value, as is the case with the $2 \times 2 \times 2$ Rubik's cube, then it can be used to denote that the state has not been reached, so that the original bit array is no longer needed.

This design needs only 459,270 bytes of memory to represent the reachable states of the $2 \times 2 \times 2$ Rubik's cube. If the rotations are also stored, then altogether about 1.4 million bytes are needed. These figures are far below the information-theoretic lower bound of 4.883 million bytes derived in Sect. 5. There is no contradiction, however, because that lower bound assumes that states are represented with 31 bits, which is no longer the case. The formulas are valid, but now $w' = 0$. Equation (1) predicts that representing the complete set of reachable states (without rotations) needs no memory at all. This is true: it is known beforehand that it will be $0, \dots, 3{,}674{,}159$. The 459,270 bytes are needed to represent the intermediate sets in the middle of the construction of the state space. Theorem 3 says that the memory needed by intermediate sets is insignificant, but its antecedent does not hold now.

With state space applications, one seldom finds a usable perfect packing algorithm because that would require a better understanding of the system under analysis than is usually available. In the case of the $2 \times 2 \times 2$ Rubik's cube, however, the structure of the states is understood very well. States differ from each other in that the seven moving corners are permuted to different orders, and six of them can be in any of three orientations. The orientation of the seventh corner is determined by the orientations of the other moving corners.

The orientations can easily be packed perfectly by interpreting them as digits of a number in base 3. Perfect packing of the order of the corners is equivalent to indexing a permutation. An efficient algorithm that does that was published in [7].

Figures 7 and 8 show a perfect packing and unpacking algorithm for the reachable states of the $2 \times 2 \times 2$ Rubik's cube based on these ideas. A packed state is a number of the form $3^6 \times p + a$, where $a = \sum_{i=0}^{5} a_i \times 3^i$ stores the orientations of six corners and p represents the permutation of the corners. It is of the form $\sum_{i=1}^{6} i!p_i$. The number p_6 is the position of corner 6 and is in the range $0, \dots, 6$. The number p_5 is the location of element 5 in the array that is obtained from the previous array by exchanging element 6 and the last element with each other. As a consequence, $0 \le p_5 \le 5$. The algorithm continues like this until the value of p_1 has been computed. At this point, element 0 is in location 0. Therefore, the corresponding p_0 would always be zero, so it need not be stored.

```
packed_state pack() const {
  packed_state result = 0;

  /* Construct the original and inverse permutation. */
  int A[ 7 ], P[ 7 ];
  for( int i1 = 0; i1 < 7; ++i1 ){
    A[ i1 ] = content[ i1 ] / 3; P[ A[ i1 ] ] = i1;
  }

  /* Pack the position permutation of the moving corners. */
  for( int i1 = 6; i1 > 0; --i1 ){
    result = ( i1 + 1 ) * result + P[ i1 ];
    A[ P[ i1 ] ] = A[ i1 ]; P[ A[ i1 ] ] = P[ i1 ];
  }

  /* Pack the orientations of the moving corners except the last one.
     Its orientation is determined by the other corners. */
  for( int i1 = 5; i1 >= 0; --i1 ){
    result *= 3; result += content[ i1 ] % 3;
  }

  return result;
}
```

Fig. 7 A state packing algorithm based on the indexing of permutations

```
void unpack( packed_state ps ){

  /* Unpack the orientations of the moving corners except the last. */
  for( int i1 = 0; i1 < 6; ++i1 ){
    content[ i1 ] = ps % 3; ps /= 3;
  }
  content[ 6 ] = 0;

  /* Unpack the positions of the moving corners. */
  int A[ 7 ]; A[ 0 ] = 0;
  for( int i1 = 1; i1 < 7; ++i1 ){
    int position = ps % ( i1 + 1 ); ps /= i1 + 1;
    A[ i1 ] = A[ position ]; A[ position ] = i1;
  }
  for( int i1 = 0; i1 < 7; ++i1 ){ content[ i1 ] += 3 * A[ i1 ]; }

  /* Find out the orientation of the last moving corner as a
     function of the orientations of other moving corners.
     (Its position has already been found out.) */
  int tmp = 120;  // a big enough multiple of 3 to ensure tmp >= 0
  for( int i1 = 0; i1 < 6; ++i1 ){ tmp -= content[ i1 ]; }
  content[ 6 ] += tmp % 3;

}
```

Fig. 8 A state unpacking algorithm based on the indexing of permutations

The algorithm in Fig. 7 differs from the above description in that it has been optimised by discarding assignments to array slots which will not be used after the assignment. Similarly, unnecessary operations have been left out of the algorithm in Fig. 8. The computation of the orientation of the last moving corner is based on the fact that, as was mentioned in Sect. 2, the sum of the orientations of all moving corners is divisible by three.

The program that is based on perfect packing of states is, of course, superior to others as far as memory consumption is concerned: 1.4 million bytes (in addition to which there is always the queue). To my great surprise it was not the fastest, however. It took 169 s to run on my laptop, which is 60% slower than the program that uses the Rubik hash table. On my workstation the time was 28 s, 40% more than with the Rubik hash table. (With the earlier compiler the time on the workstation was as much as 67 s.) When measuring the speed, I reserved one byte per state to avoid the slowdown caused by stowing 3-bit rotation information in the middle of bytes. Such a state array uses 3.67 million bytes, which is still much less than with the other programs.

That perfect packing lost in speed to the Rubik hash table was surprising, but it has a simple explanation. Both programs use most of their time for packing and unpacking states and for seeking packed states in the data structure for reached states. Although searching through the Rubik hash table requires scanning lists and extracting 13-bit entries from 2-byte slots, this does not take much time because the lists are not long and the information is always in the same place within the bytes. On the other hand, despite their simplicity, the algorithms in Figs. 7 and 8 do dozens of operations more than the packing and unpacking algorithms used by the Rubik hash table. The latter algorithms work similarly to the handling of orientations in the former, except that the orientation of every corner is stored. In brief, the program based on perfect packing does altogether much more work.

10 BDDs and the $2 \times 2 \times 2$ Rubik's cube

A *binary decision diagram* (BDD) is an important data structure for representing large sets of bit vectors [2]. When talking about the efficiency of algorithms for constructing big sets of states, the question of how BDDs would have performed is almost inevitable.

A BDD is a directed acyclic graph. There are two special nodes which we will call False and True. They have no outgoing arcs. Each of the remaining nodes has precisely two outgoing arcs, leading to nodes that may be called the 0-*child* and 1-*child* of the node. Each node except False and True has a *variable number*, and the variable number of any node is smaller than the variable numbers of its children. Each variable number corresponds to a bit position in the bit vectors; however, the variable numbering may be any permutation of bit positions.

Each node represents a set of bit vectors. Whether or not a given bit vector is in the set is determined by starting at the node, reading the bit identified by its variable number, going to its 0-child or 1-child according to the value of the bit, and continuing like this until arriving at the False or True node. The bit vector is in the set if and only if the traversal ends at the True node.

In a *normalised* BDD the 0-child and 1-child of any node are different, and for any two different nodes they have different 0-children or different 1-children (or both). Each set has a unique (up to isomorphism) normalised BDD representation for each variable numbering. From now on, by BDD is meant a normalised BDD.

An upper bound of $(2+\varepsilon)\frac{2^w}{w}$ (where $\varepsilon \to 0$ as $w \to \infty$) for the number of BDD nodes was proven in [6]. In [9] a formula for the average number of nodes was given for the variant where the 0-child and 1-child of a node may be the same, and the variable number of any node is precisely one less than the variable numbers of its children. The next theorem does the same for the standard normalised BDDs (with a simpler proof).

Theorem 6 *The expected number of nodes (not including the* False *and* True *nodes) of a BDD for a random set of* w*-bit elements is*

$$2^{-2^w} \sum_{k=0}^{w-1} 2^{2^{w-k-1}} \left(2^{2^{w-k-1}} - 1\right)\left(2^{2^w} - \left(2^{2^{w-k}} - 1\right)^{2^k}\right).$$

When $w \geq 3$, *this is roughly* $2 \times \frac{2^w}{w}$.

Proof There are $2^{2^{w-k}}$ different Boolean functions over the $w - k$ last variables. Let us go through all 2^{2^w} functions f over all variables and count how many of them have a particular function g over $w - k$ last variables as a subfunction. Each f invokes 2^k such subfunctions (not necessarily distinct), one for each possible combination of values of the k first variables. None of these is g in $(2^{2^{w-k}} - 1)^{2^k}$ cases. Thus g is present in $2^{2^w} - (2^{2^{w-k}} - 1)^{2^k}$ cases. Not every g corresponds to a node labelled with k because the 0- and 1-child of a node must be different. The node exists in $2^{2^{w-k-1}}(2^{2^{w-k-1}} - 1)$ cases. The product of these yields the total number of nodes labelled with k, when all f are gone through. Summing them up and then dividing by the number of different f gives the precise formula.

That the number of nodes is $\approx 2 \times \frac{2^w}{w}$ has been verified numerically up to $w = 26$ (when $3 \leq w \leq 26$, it is between $1.4 \times \frac{2^w}{w}$ and $2.3 \times \frac{2^w}{w}$). When $w > 26$, consider any $k \leq w - \log_2 w$. We have $2^{w-k} \geq 2^{\log_2 w} = w \gg \max\{2, k\}$. Thus, $2^{2^{w-k-1}}(2^{2^{w-k-1}} - 1) \approx 2^{2^{w-k}}$. Furthermore, $2^{-2^w}(2^{2^w} - (2^{2^{w-k}} - 1)^{2^k}) = 1 - (1 - 2^{-2^{w-k}})^{2^k} = 2^k 2^{-2^{w-k}} - \frac{1}{2}2^k(2^k - 1)(2^{-2^{w-k}})^2 + \cdots \approx 2^k 2^{-2^{w-k}}$. The number of nodes on layer k is thus $\approx 2^{2^{w-k}} 2^k 2^{-2^{w-k}} \approx 2^k$. (This means that the layer is almost as full as it can be.)

If $w - \log_2 w$ is an integer, then the layers $0, \ldots, w - \log_2 w$ contain altogether $\approx 2 \times 2^{w - \log_2 w} \approx 2 \times \frac{2^w}{w}$ nodes. The total number of nodes on layers $k \geq w - \log_2 w + 1$ is negligible because there is at most one node per each different Boolean function over the $\log_2 w - 1$ last variables and there are $2^{2^{\log_2 w - 1}} = 2^{w/2} \ll \frac{2^w}{w}$ such functions.

If $w - \log_2 w$ is not an integer, then $2 \times \frac{2^w}{w}$ overestimates the number of nodes on the layers up to $\lfloor w - \log_2 w \rfloor$, but then the size of the layer $\lceil w - \log_2 w \rceil$ is not necessarily negligible. Because a layer cannot contain more than twice as many nodes as the previous layer, the approximation must be multiplied by a factor that is between $\frac{1}{2}$ and 2. □

A BDD node contains two pointers (or other children specifiers) and the variable number. BDD nodes usually also contain other fields supporting normalisation, memory management, etc., but we ignore them here. Therefore, a BDD with q nodes uses at least

$$q(2 \log_2 q + \log_2 w) \tag{8}$$

bits of memory. Letting $q \approx 2 \times \frac{2^w}{w}$ we see that the memory consumption of a BDD for a random set is at least roughly 4×2^w bits, that is roughly four times the information-theoretic lower bound.

Unfortunately, not much is known (or at least I do not know much) about the sizes of BDDs that store sets that are much smaller than the universe from which they are drawn. We can, however, experiment with the issue in the $2 \times 2 \times 2$ Rubik's cube setting.

Let us first investigate the case of random sets. In three experiments, storing 3,674,160 randomly chosen 31-bit elements (each different from the others) required 3,362,953, 3,364,366, and 3,363,225 BDD nodes.[3] Thus

[3] One must be careful with this kind of experiment. I first used a typical linear congruent pseudorandom number generator and got

Table 4 BDD sizes for the 2 × 2 × 2 Rubik's cube state space

	No rotations		Rotations first		Rotations last	
Model	lsb first	msb first	lsb first	msb first	lsb first	msb first
31-bit end	2,251,928	406,549	3,533,649	2,285,862	3,475,829	3,207,427
Max	2,251,928	1,030,589	3,533,649	2,308,429	3,475,829	3,207,427
35-bit end	2,964	2,964	1,870,059	1,965,956	2,389,372	2,684,500
Max	722,180	775,096	2,069,208	2,009,657	2,389,372	2,684,500

about 20 million bytes of memory are required even without the rotation information.

Why do BDDs need so much more memory than the information-theoretic lower bound of 4.883 million bytes? One reason is that several different bit combinations can represent the same BDD. For instance, if the contents of two nodes are swapped and the pointers leading to them are changed accordingly, the result represents the original BDD, although the bit pattern in memory has changed. Thus a BDD with q nodes has $q!$ different representations. This corresponds to $\log_2 q!$ bits of useless information. According to Eq. (3), in the case of 3.363 million nodes this makes 8.5 million bytes. Furthermore, many bit patterns are wasted because they fail to represent valid BDDs (for instance, when a node specifies a node with too small a variable number as its child). Yet another reason is that many small BDDs represent sets that are larger than $3{,}674{,}160$ elements. From the point of view of the intended use of BDDs this is a good thing, but from the point of view of storing sets of $3{,}674{,}160$ elements they are wasted bit patterns.

We see that BDDs clearly lost in the case of random sets of $3{,}674{,}160$ elements of 31 bits. This is not a shame, because BDDs were not designed for storing smallish random sets; they were designed for storing (some) very large sets in relatively little memory. As was discussed in Sect. 5, for any given data structure, only a small minority of large sets can have small representations. BDDs have been successful because many sets arising in BDD applications are not arbitrary but have some kind of regularity that BDDs exploit.

The obvious question now is if this is true also of the set of (non-randomised) reachable states of the 2 × 2 × 2 Rubik's cube. That can be found out by experimenting. The results of this and related experiments are in Table 4. The first entry of the table reveals that when the bits are indexed starting from the least significant, 2,251,928 BDD nodes, and consequently a lot of memory, are needed. With the opposite order the number of nodes is only $406{,}549$. Curiously, this is the ordering with which the non-randomised Rubik hash table worked particularly badly.

The latter number is promising, but we have to take into account an extra issue: it is common that the intermediate BDDs arising in the middle of the construction of a final BDD are much larger than the final BDD. (This issue was investigated in [3].) In the msb first case, when states were added one at a time, the largest number of BDD nodes needed to represent the states found so far was $1{,}114{,}730$. BDD programs usually add states one breadth-first frontier at a time, however. In that case, the biggest intermediate BDD contained $1{,}030{,}589$ nodes, corresponding to 5.8 million bytes of memory according to Eq. (8) (that is, assuming extremely dense representation for the BDD).

A reviewer of this paper pointed out that the mediocre performance of BDDs in storing the 2 × 2 × 2 Rubik's cube's reachable states may be due to the use of a very dense packing of the state. The reviewer suggested an alternative packing, where each position and each orientation are given bits of their own. Because positions range from 0 to 6 and orientations from 0 to 2, three bits suffice for representing a position and two for an orientation, so altogether 35 bits are needed. As can be seen from the first two entries on line '35-bit end', this suggestion was very successful. Peak BDD sizes are still somewhat of a problem, though. Storing $722{,}180$ BDD nodes takes at least 4.0×10^6 bytes of memory according to Eq. (8).

(The method presented in [3] reduces the peak BDD size to 8,303 nodes. The maximum number of nodes in simultaneous existence ever during the computation – including the nodes used for representing the transition relation, and temporarily used during the computation of BDD operations – was $186{,}318$ in my BDD implementation. The method of [3] is not, however, particularly useful in this case because it is based on constructing the reachable states in a non-breadth-first order. So information about the lengths of the rotation sequences is lost. Just obtaining the final set of reachable states is not interesting because it has been known all the time: it was fully described in Sect. 2.)

The 35-bit representation of states thus makes it possible to construct the reachable states in breadth-first order within a reasonable number of BDD nodes. From the result it is possible to find out, for instance, the length of the longest rotation sequence needed to solve the cube. However, to really compete with the other approaches discussed in this paper, the BDD implementation should also store the rotation information. Without it, the BDD program cannot be used for actually solving the cube. The latter four columns of Table 4 show BDD sizes when the rotation information is present. BDDs are clearly not competitive any more (at least with the variable orderings that I tried).

3,316,523, 3,316,727, and 3,316,535. However, generating fully random sets (thanks to Theorem 6, their expected BDD sizes are precisely known up to $w = 26$) revealed the results I was getting were too small. I added the algorithm of Fig. 5 as a postprocessing step and started to get credible results.

11 Conclusions

This research started as a modest programming exercise but led to many useful observations and to the development of the very tight hash table data structure. The first observation was that data structures picked from high-quality libraries – the C++ standard library map in this case – may be very inefficient for a particular application. By designing an optimised application-specific data structure one can get a tremendous improvement in both speed and memory consumption at the same time. These results are summarised in Table 5.

More importantly, this research demonstrated the usefulness of the information-theoretic view for the design of efficient data structures for large sets. The basic idea is that, given a certain amount of memory, one can have only so many bit combinations, and one should try to exploit them in the best possible way. Bit combinations that do not represent any set are wasted and thus increase memory usage. It is also wasteful if the same set can be represented with several different bit combinations. The memory consumption of BDDs, for instance, is significantly affected by these considerations.

If the universe (that is the set of syntactically possible states) is big, as is almost always the case with state space applications, then no data structure can represent an arbitrary set with small memory requirements, simply because there are so many sets that a lot of memory is needed to have enough bit combinations to distinguish between them. Therefore, to have any chance of success, one has to concentrate on sets in some special category. BDDs have been successful because many large sets arising in verification applications happen to be among those that have small BDD representations. Interval lists have not been successful in the same sense and have thus not found much use in state space applications.

In this research we concentrated on sets whose special property is that *they are small compared to the universe*. This is a very reasonable assumption in explicit state space applications because there the set starts its life as the empty or singleton set and then grows one element at a time until it is complete or the memory has been exhausted. Furthermore, we assumed that *other than that, the set is arbitrary*. If the set is known not to be arbitrary in its size class, then it may be possible to obtain much better results than we did in this research. If it is not known whether the set is arbitrary in its size class, then one can always ensure that it is by randomising the elements before storing them. This is important, because uncontrolled regularity can also be harmful, as we saw. (Curiously, a set can easily be randomised among the sets of the same size, and that is useful, while randomising it in general seems difficult and would be senseless. General randomisation would make it an arbitrary subset of the universe, and an arbitrary subset almost certainly could not be stored. This supports our view that set size should not be considered arbitrary.)

In this setting we derived the lower bound $nw - n\log_2 n + 1.443n$ for the average memory consumption, where n is the number of elements and w is the number of bits in an element. It can be rewritten as $nw' + 1.443n$, where w' is the number of 'extra' bits in the element – extra in the sense that $w - w'$ bits suffice for the universe to contain n different elements.

The lower bound allows for going well below nw, which is the amount of memory needed for representing every element explicitly. Since most classic data structures do represent every element explicitly, it is possible to improve considerably on their memory usage. To that end, we designed the very tight hash table data structure. It is both very memory efficient and reasonably fast.

Another observation was that if elements are thrown to lists such that every list has the same probability of getting the element, then the distribution of list lengths is far from flat. In the case of the $2 \times 2 \times 2$ Rubik's cube, the length of the longest list was more than twice the average length, and three lists contained only one element each, although the average length was about 14. This is not a consequence of bad randomisation. It is a statistical phenomenon, which must be taken into account when designing data structures with good average-case performance.

We also tried to exploit the regularity of the state space of the $2\times2\times2$ Rubik's cube. First, in this case it was possible to base a program on perfect packing of states. Perfect packing leads to extremely good memory consumption and very fast state space manipulation operations. It was thus surprising that the program based on it proved to be somewhat slower than the Rubik hash table program. The reason is that, although perfect packing and unpacking are fast in this case, the very simple packing and unpacking used by the Rubik hash table program are nevertheless so much faster that they compensate for the somewhat slower state set operations.

Second, after a number of experiments (and a suggestion by a reviewer of this paper), a representation of the states

Table 5 Summary of data structures and bounds

Data structure	Mem 10^6	Time laptop	Time workstation	Generality
C++ map	88.2		98	+
nw-bound	14.2			−
Rubik hash table	11.1	105	20	−
Very tight hash table	8.9	410	48	+
Bound $\approx nw' + 1.443n$	6.1			+
Perfect packing	3.7	169	28	−

Table 6 The number of reached states as a function of the number of rotations

Rot.	States	New	Rot.	States	New	Rot.	States	New
0	1	1	5	2,944	2,256	10	1,450,216	930,588
1	7	6	6	11,913	8,969	11	2,801,068	1,350,852
2	34	27	7	44,971	33,058	12	3,583,604	782,536
3	154	120	8	159,120	114,149	13	3,673,884	90,280
4	688	534	9	519,628	360,508	14	3,674,160	276

and a BDD variable ordering were found that led to a small BDD for the set of reachable states. However, unlike hash tables, with BDDs the sizes of the representations of the intermediate sets during the construction of the state space are an important issue. With extreme packing of the BDD, 4.0×10^6 bytes would be needed for the biggest BDD for an intermediate set. This figure is not realistic in practice, though, because it ignores the memory that a practical BDD implementation needs for BDD normalisation and memory management. Furthermore, rotation information was not stored in that BDD. Adding the rotation information made the BDDs grow big. It would also be extremely hard for BDD-based programs to beat the Rubik hash table program in speed.

What can we use the Rubik hash table program for? When the reachable set of states has been constructed, one can feed in any state of the cube, and the program responds with directions for rotating the cube back to the initial state. The reply presents an optimal solution, that is a solution with the smallest possible number of rotations. However, many people solve (although not optimally) the $2 \times 2 \times 2$ cube by hand in less time than it takes to feed in the state.

A more important application is the investigation of the structure of the state space. For instance, Table 6 gives the number of different states that can be reached from the initial state by at most the number of rotations given in the first column and tells how many of them cannot be reached with fewer rotations. From the table we see that no more than 14 rotations are ever needed to solve the cube. On the other hand, more than half the states require more than 10 rotations. A program that solves the cube by doing two breadth-first searches at the same time, one starting from the given state and the other from the initial state, would find optimal solutions, although it would never need to investigate more than 89,942 states, less than 2.5% of the state space.

What next? Finding optimal solutions for the $3 \times 3 \times 3$ Rubik's cube would be a natural challenge. Unfortunately, it has $8! \times 12! \times 2^{12} \times 3^8/12 \approx 4 \times 10^{19}$ reachable states, which would perhaps be too much even for the computers of tomorrow. An optimal solution for just the corner pieces can be found in 20 s, however. This is because the $2 \times 2 \times 2$ cube is the set of the corner pieces of the $3 \times 3 \times 3$ cube.

Acknowledgements Jaco Geldenhuys has provided me with considerable help at various stages of this research. This text is based on [8] (in Finnish), on which Jyrki Lahtonen of the University of Turku and Ari Korhonen of Helsinki University of Technology gave helpful comments.

References

1. Abramowitz, M., Stegun, I. (eds.): Handbook of Mathematical Functions, with Formulas, Graphs, and Mathematical Tables. New York: Dover 1970
2. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Comput. **35**(8), 677–691 (1986)
3. Geldenhuys, J., Valmari, A.: Techniques for smaller intermediary BDDs. In: CONCUR 2001 – 12th International Conference on Concurrency Theory, Aalborg, Denmark, 21–24 August 2001. Lecture Notes in Computer Science, vol. 2154, pp. 233–247. Berlin, Heidelberg, New York: Springer 2001
4. Geldenhuys, J., Valmari, A.: A nearly memory-optimal data structure for sets and mappings. In: Proceedings of the 10th International SPIN Workshop on Model Checking Software, Portland, OR, 9–10 May 2003. Lecture Notes in Computer Science, vol. 2648, pp. 136–150. Berlin, Heidelberg, New York: Springer 2003
5. Knuth, D.E.: The Art of Computer Programming, vol 3: Sorting and Searching. Reading, MA: Addison-Wesley 1973
6. Liaw, H.-T., Lin, C.-S.: On the OBDD-representation of general boolean functions. IEEE Trans. Comput. **41**(6), 661–664 (1992)
7. Myrvold, W., Ruskey, F.: Ranking and unranking permutations in linear time. Inform. Process. Lett. **79**(6), 281–284 (2001)
8. Valmari, A.: Mitä pieni Rubikin kuutio opetti minulle tietorakenteista, informaatioteoriasta ja satunnaisuudesta. Tietojenkäsittelytiede **20**, 53–78 (2003)
9. Wegener, I.: The size of reduced OBDD's and optimal read-once branching programs for almost all boolean functions. IEEE Trans. Comput. **43**(11), 1262–1269 (1994)