DIGICUBE

The programmable virtual Rubik's cube

Version 1.2

Reference Manual

www.softwareandmind.com

Digicube v1.2 and Digicube Reference Manual
© Copyright 2018, 2022 Andrei Sorin

Published by Andsor Research Inc., Toronto, Canada
Information and software download: www.softwareandmind.com

This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

Contents

CHAPTER 1

Concepts and features

Digicube is the implementation of a virtual Rubik's cube: it lets you work with a simulated cube instead of a real, physical one. The program maintains in its memory the cube's current position (the arrangement of pieces), and lets you modify it by specifying the kind of operations that you would perform with a real cube. You can start with the solved position, or with a random position, or by entering a specific position. And, at any time, you can ask the program to solve the current position, fully or partially, and display the required moves. Thus, Digicube lets you solve a scrambled real cube by entering its position and then performing the suggested moves. But it lets you do much more, including feats that are impractical with a real cube (like showing the position reached after thousands of moves). It is an ideal tool for experimenting with positions, move sequences, and solutions.

You use Digicube by entering instructions interactively or by storing them as scripts in a file. Here are some of the operations you can perform: specify sequences of moves or turns, or generate random sequences; modify, swap, or flip individual pieces; store positions in memory and retrieve them later, which also lets you simulate working with several cubes; compare the current position with a stored one; generate a scrambled or partially scrambled position; check the validity of a position; display the current position in various ways; determine the moves needed to reach any position, including partially specified positions.

Several features distinguish Digicube from other cube programs and from the manual solutions. One is the system of notation: instead of the complicated letters and symbols traditionally used to specify colors, faces, and rotations, Digicube uses only the digits 1 to 6. Each face, color, move, and turn is represented by a digit, while the locations and colors of the 26 pieces are represented by numbers composed of these digits. Thus, all you need to know in order to solve a cube is how the six digits correspond to the cube's attributes. This correspondence is simple and logical, and therefore easy to learn.

Another feature is discovering solutions that involve only three faces (*right*, *front*, and *down*, which are the easiest to access): apart from a few moves needed at start to establish the *anchor*, you never rotate the other three faces or the whole cube. Thus, while the solution may require more moves than the traditional methods, it is ultimately easier to apply on a real cube.

Finally, unlike other programs, Digicube doesn't employ mathematical or empirical algorithms. Instead, it discovers solutions simply by trying large numbers of move sequences. Thus, it is unusual in that it shows that it is possible, through programming techniques alone, to reduce the astronomic number of possible positions to subsets that can be analyzed in a few seconds.

Other models

In addition to the classic 3× Rubik's cube, Digicube can simulate two simpler versions: the 2× cube (known as Pocket cube) and the 3× pyramid (known as Pyraminx). The three versions are known as models, and are called for short M1, M2, and M3 (cube 3×, cube 2×, pyramid 3×, respectively). Digicube can simulate one model at a time, and you select the current model with a run option (the default is M1).

To avoid repetition, this manual is devoted to M1, and then, in separate chapters, discusses those features that are different for M2 and M3. Thus, even if you are interested only in M2 or M3, you must start by reading the M1 chapters, because the discussions common to all models are not repeated under M2 and M3.

CHAPTER 2

Installing and running Digicube

Digicube works on PCs running any regular 32-bit or 64-bit version of Microsoft Windows. It needs only about 120 Kb of memory (plus whatever resources Windows needs to run it), and only as much space on the hard drive as you allow for the output log file (typically about 1 Mb). It is, however, cpu-intensive. On a multi-core computer it uses only one core, but multiple instances running simultaneously use different cores. In this manual, the performance times mentioned are for a PC with a 3.4 GHz Intel i5-7500 processor running Windows 10.

To run Digicube, all you need is the program file, **digicube.exe** (and an input file if you write scripts). There is no formal installation procedure; simply copy the program to any convenient folder (directory), and start it by double-clicking on its entry in the folder. It is best, though, to create a new folder (for example, in the Program Files folder), and to extract there all the files from the digicube.zip package. To uninstall, delete this folder. Digicube does not modify the Registry or any other files, and does not create any files apart from the output log file. For convenience in starting Digicube, create a shortcut icon on the desktop (to do this, right-click on the program's entry in the folder).

Digicube is a text-mode console program. When started, it opens a console window, where the keyboard input and the program's output are displayed; the window closes when the program ends. The first time you run Digicube, if you never ran a console program before on that computer, Windows may take a few seconds to install the necessary utility. You can customize the console window by right-clicking on the top bar of its frame and selecting Properties; additional properties are available by right-clicking on the program's shortcut icon or its folder entry. You can change such features as the text and background colors, the font (use a monospaced one, like Courier or Consolas), the window size, and the number of lines that can be recalled by scrolling back. Your choices are stored, and are used whenever you execute Digicube.

Each Digicube execution consists of a series of *runs*. Before a run, it displays "Enter run options (X to end, H for help, I for interactive)". X will end Digicube. H will display the available run options with brief explanations, and will show how to display additional help lists. Otherwise, to start a run, type any needed options, separated by spaces, and press Enter (see chapter 4, "Run options"). With no options entered, default values take effect (specifically, Digicube will read the default input file and execute the first script found there). If you include the option I, an interactive run is started (in which case no input file is needed). It is also possible to specify options later, during the run. You can use the arrow keys and the editing keys to recall and edit options entered in previous runs of the same execution. The runs are independent of one another: apart from the ability to recall previous options, there is no difference between starting a new run and ending Digicube and then restarting it.

You can stop a running script by pressing X (upper or lower case). This is useful when the script contains a group of operations, moves, or turns that are being repeated many times and you don't wish to wait for this to end. In interactive mode, only moves and turns can be repeated many times, and you can stop the repetitions in the same way. Some custom operations also involve many repetitions and can be stopped. Finally, certain solution types, especially if you allow longer move sequences than the default, may take longer than you expected and can be stopped with X. (Sometimes it takes a few seconds for the X to take effect.)

Input and output files

Digicube uses an input file and an output file. These are ordinary text files and can be accessed with any text editor (Notepad, for example). Their default names are **diginp.txt** and **digout.txt**, and their default location is the folder where the program file, digicube.exe, resides. You can specify different names and locations with the run options I and O. For proper alignment of the text data, use a monospaced font (like Courier) in the editor, and no word wrap.

The input file is where scripts are stored. In a regular run, Digicube executes a script; in an interactive run (a run that is entirely interactive), no input file is needed, since all input is then from the keyboard. Digicube reads the input file and never writes to it; it is your responsibility to create and maintain it. An example file diginp.txt, which contains a few scripts, is included in the digicube.zip package.

When you modify the input file with an editor, make sure you close it before starting Digicube (otherwise the changes may not be seen). Avoid modifying the file while Digicube is running, but it is safe to modify it between runs, when the prompt "Enter run options …" is displayed. Multiple instances of Digicube running simultaneously can read the same input file.

In the output file, Digicube writes everything that is displayed in the console window, including the keyboard entries in interactive mode. This file provides, therefore, a log of your activities, which you can review later. But you can also modify it if you wish (after the program ends), since Digicube doesn't read it. If no output file exists, Digicube creates one automatically. When one exists, the new lines are added at the end, unless you instruct Digicube (with the run option N) to delete the file and create a new one.

The output file is updated continuously, in step with the displayed text. Thus, if a run ends abnormally, the file will include everything that was displayed (except perhaps the last, incomplete line). If you open the file with an editor while Digicube is running, you will not see the lines being added at the end; you must close it and reopen it in the editor to bring it up to date. Avoid modifying the file while Digicube is running. If multiple instances of Digicube run simultaneously, make sure you specify different output files (with the run option O), otherwise they would all add lines to the same file at the same time, and the lines would be jumbled up.

When the output file exceeds 1 MB, the message "Reminder: digout.txt is over 1 MB" (with the actual file name, if different) is displayed at the end of each run. This changes to 2 MB, 3 MB, etc. as the file grows. If you no longer need the old contents, use the run option N to re-create the file; or delete it manually, or reduce its size by deleting unwanted portions with the editor, or specify another file. Otherwise, the file will grow indefinitely.

DOS window and 16-bit version

Although there are, generally, no benefits, Digicube can also run in a DOS window under Windows. You can use either command.exe or cmd.exe as DOS emulator (both are in windows\system32\), so choose the one that runs better in your Windows version.

There are a few differences in a DOS window. First, you cannot scroll back the displayed lines as you can in a console window. Second, you start the program by entering its name on the command line, and there is an alternative for specifying the run options: Instead of waiting for the message "Enter

run options …", you can enter the options following the program name when starting it. In this case, that message is omitted and the execution consists of only one run: when the run ends, the program too ends. Third, the default location for the input and output files is the current folder, which may or may not be the one where digicube.exe resides.

D16cube is a 16-bit version of Digicube, identical to it and meant for 16-bit operating systems like MS-DOS (the hardware, though, must still be 32-bit or 64-bit). It also runs under Windows, in a DOS window; set *idle sensitivity* to minimum (in Properties, Miscellaneous), else it runs very slowly. You start it by typing its name at the DOS prompt, like Digicube when running in a DOS window, and you can include the run options with the name. With D16cube, the editing capabilities when recalling run options or entries in interactive mode are limited; and the input and output file and folder names specified with the run options I and O cannot have spaces or more than 8 characters.

CHAPTER 3

Definitions and terminology

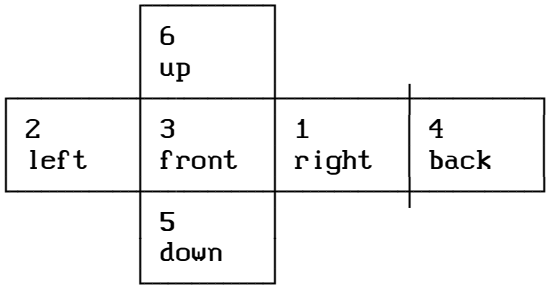
3.1 Sides, faces, and colors

A *piece* is one of the 26 outer parts of the cube; there are 8 corner pieces, 12 edge pieces, and 6 center pieces. The cube has 6 *faces*, each one comprising 9 pieces. (The faces are called sometimes *layers*, to emphasize the pieces and their rotation, as opposed to the outer surface alone.) Each face has its own center piece, but shares the corner and edge pieces with the adjacent faces. This is the arrangement of pieces in each face:

corner	edge	corner
edge	center	edge
corner	edge	corner

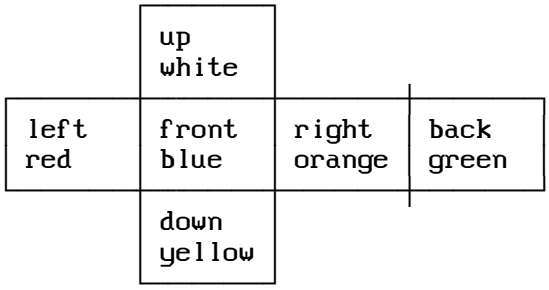
The center piece in each face defines the color of that face. Since the 6 center pieces do not move when the faces are rotated, they can always be used to identify the faces, whether the cube is solved or scrambled. And since their relative position is also fixed, they can be used to identify the orientation of the cube in space (which face is *up*, *front*, etc.). It is these facts that allow us to depict all the cube's features (faces, colors, pieces, rotations, orientation) with the same 6 digits.

The 6 *sides* of a cube refer to the space surrounding it: *right*, *left*, *front*, *back*, *down*, *up*. The sides form a fixed frame of reference: no matter how the whole cube or its faces are rotated, *right* always refers to its right side, *front* to its front side, etc. The sides are defined from the perspective of the cube's user. You can think of this frame of reference as a larger, imaginary cube, within which the real cube rotates. The 6 digits are assigned to the 6 sides as shown in this 2-dimensional schematic of the imaginary cube:

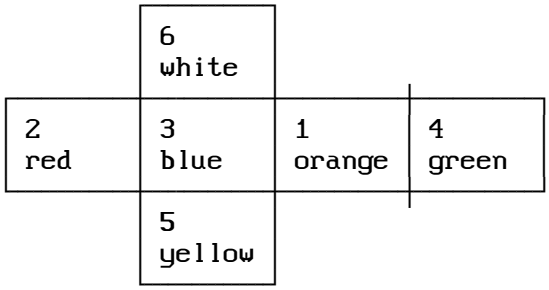


Next, we must assign the 6 digits to the cube's colors. The 6 colors are defined by the center pieces, which determine the cube's orientation; and since we already have digits for the 6 sides, all we have to do is establish a correspondence between the center pieces and the sides. There are 6×4=24 possible orientations (each one of the 6 faces can be *down*, and in each case 4 faces can be *front*), and we select one of them as the *standard orientation*. Which orientation is the standard one is a personal matter:

you must decide, depending on the cubes you are using, which center piece color you want to be *up*, *down*, *front*, etc. For example, a color scheme that matches most cubes currently available is illustrated in this schematic of the cube's faces:



In the standard orientation, then, these will always be the colors of the center pieces, no matter how the faces are rotated. The choice of digits is now obvious: each color inherits the digit of the side where the center piece with that color resides when the cube is in the standard orientation. By combining the previous two schematics, we can depict this correspondence with the following schematic of the center pieces:



The cube's faces are also identified with digits: they are numbered 1 to 6, based on the color of their center piece (a face always has the same center piece). As the cube turns, any face can be at any side. Only in the standard orientation do all the faces correspond to sides: face 1 at side 1 (*right*), face 2 at side 2 (*left*), face 3 at side 3 (*front*), and so on.

To conclude, you must memorize two conversion tables, which are closely related: digits-to-sides and digits-to-colors. Here are the two tables:

1=right, 2=left, 3=front, 4=back, 5=down, 6=up.

1=orange, 2=red, 3=blue, 4=green, 5=yellow, 6=white.

While the first table is always the same, the second one depends on your choice of the standard orientation. The values shown above are for the choice we made earlier (*right*=orange, *left*=red, *front*=blue, and so on). If you choose instead, for example, the orientation where *right*=red, *left*=orange, *front*=green, *back*=blue, *down*=yellow, *up*=white, the second table will be:

1=red, 2=orange, 3=green, 4=blue, 5=yellow, 6=white.

To memorize the digits-to-colors table, it helps if you realize that it too is fixed, in a way, not unlike the digits-to-sides table. The colors reflect the center pieces in the standard orientation (and the whole face when the cube is solved). Therefore, even though the colors depend on the choice of standard orientation, their digits are always the same: color 1 is always *right*, color 2 is *left*, color 3 is *front*, color 4 is *back*, color 5 is *down*, color 6 is *up*.

3.2 Positions

A *position* is a particular arrangement of the cube's pieces, in a particular orientation. The most economical representation of a position is as shown by the run option D and the operation D, or as you specify it in a script with the operation P. It consists of a list of 54 color digits (6 faces of 9 pieces), separated by spaces. The faces are listed in the numerical order of the 6 sides, regardless of the cube's orientation. Thus, the first group of 9 digits is the face at side 1 (*right*), the second group is the face at side 2 (*left*), and so on. The list is broken into two lines for convenience. The solved position looks like this:

```
1 1 1 1 1 1 1 1 1   2 2 2 2 2 2 2 2 2   3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6
```

As a 2-dimensional cube schematic, the same position looks like this:

```
            +---+---+---+
            ¦ 6 ¦ 6 ¦ 6 ¦
            +---+---+---+
            ¦ 6 ¦ 6 ¦ 6 ¦
            +---+---+---+
            ¦ 6 ¦ 6 ¦ 6 ¦
+---+---+---+---+---+---+---+---+---¦---+---+---+
¦ 2 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 3 ¦ 1 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---+---+---+---+---+---+---+
¦ 2 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 3 ¦ 1 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---+---+---+---+---+---+---+
¦ 2 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 3 ¦ 1 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---+---+---+---¦---+---+---+
            ¦ 5 ¦ 5 ¦ 5 ¦
            +---+---+---+
            ¦ 5 ¦ 5 ¦ 5 ¦
            +---+---+---+
            ¦ 5 ¦ 5 ¦ 5 ¦
            +---+---+---+
```

The solved position turned 90 degrees (so that face 1 goes from *right* to *front*) looks like this:

```
4 4 4 4 4 4 4 4 4   3 3 3 3 3 3 3 3 3   1 1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2 2   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6
```

Within each face, the 9 pieces are treated as 3 rows of 3 digits, read top to bottom, and left to right in each row. The top of faces 1, 2, 3, and 4 is defined as their upper edge; for face 5, it is the edge shared with face 3; for face 6, it is the edge shared with face 4. This is also how the faces appear in the 2-dimensional schematic.

Based on this logic, there is another, simpler 2-dimensional display format: rows of color digits. It shows the cube's faces as 3 rows of 3 colors, as they appear in a real cube. The position shown above looks like this:

```
        6 6 6
        6 6 6
        6 6 6

3 3 3   1 1 1   4 4 4   2 2 2
3 3 3   1 1 1   4 4 4   2 2 2
3 3 3   1 1 1   4 4 4   2 2 2

        5 5 5
        5 5 5
        5 5 5
```

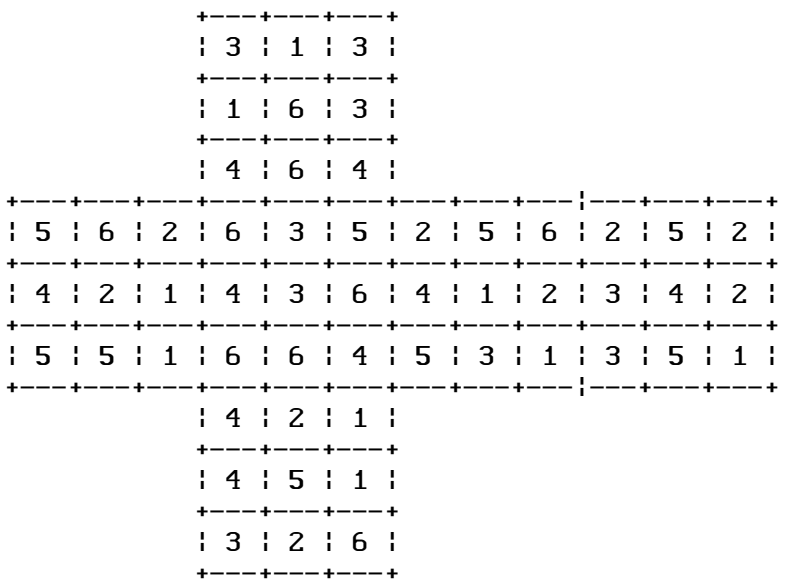
And here is the position of a scrambled cube, shown as list of colors, rows of colors, and schematic:

```
2 5 6 4 1 2 5 3 1   5 6 2 4 2 1 5 5 1   6 3 5 4 3 6 6 6 4
2 5 2 3 4 2 3 5 1   4 2 1 4 5 1 3 2 6   3 1 3 1 6 3 4 6 4

        3 1 3
        1 6 3
        4 6 4

5 6 2   6 3 5   2 5 6   2 5 2
4 2 1   4 3 6   4 1 2   3 4 2
5 5 1   6 6 4   5 3 1   3 5 1

        4 2 1
        4 5 1
        3 2 6
```



Note how the faces and digits correspond in the three formats. Note also how the center digits of the faces in the schematic correspond to the center digits of the faces in the rows of colors, and to the middle digits of the faces in the list of colors. Because this cube is in the standard orientation, the center digits and the middle digits are in numerical order, 1 to 6 (because they correspond to sides 1 to 6, and the displays are in side order).

A position, thus, defines not only the arrangement of pieces but also the cube's orientation: if we alter its orientation, without rotating any faces, its position will change. As example, here is the same cube after turning it 180 degrees, so that face 3 goes from *front* to *back*:

```
5 6 2 4 2 1 5 5 1   2 5 6 4 1 2 5 3 1   2 5 2 3 4 2 3 5 1
6 3 5 4 3 6 6 6 4   6 2 3 1 5 4 1 2 4   4 6 4 3 6 1 3 1 3

      4 6 4
      3 6 1
      3 1 3

2 5 6  2 5 2  5 6 2  6 3 5
4 1 2  3 4 2  4 2 1  4 3 6
5 3 1  3 5 1  5 5 1  6 6 4

      6 2 3
      1 5 4
      1 2 4

          +---+---+---+
          ¦ 4 ¦ 6 ¦ 4 ¦
          +---+---+---+
          ¦ 3 ¦ 6 ¦ 1 ¦
          +---+---+---+
          ¦ 3 ¦ 1 ¦ 3 ¦
+---+---+---+---+---+---+---+---+---¦---+---+---+
¦ 2 ¦ 5 ¦ 6 ¦ 2 ¦ 5 ¦ 2 ¦ 5 ¦ 6 ¦ 2 ¦ 6 ¦ 3 ¦ 5 ¦
+---+---+---+---+---+---+---+---+---+---+---+---+
¦ 4 ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 2 ¦ 4 ¦ 2 ¦ 1 ¦ 4 ¦ 3 ¦ 6 ¦
+---+---+---+---+---+---+---+---+---+---+---+---+
¦ 5 ¦ 3 ¦ 1 ¦ 3 ¦ 5 ¦ 1 ¦ 5 ¦ 5 ¦ 1 ¦ 6 ¦ 6 ¦ 4 ¦
+---+---+---+---+---+---+---+---+---¦---+---+---+
          ¦ 6 ¦ 2 ¦ 3 ¦
          +---+---+---+
          ¦ 1 ¦ 5 ¦ 4 ¦
          +---+---+---+
          ¦ 1 ¦ 2 ¦ 4 ¦
          +---+---+---+
```

To enter the position of a real cube, you use the operation P followed by all 54 digits, or the operations P1 to P6, each one followed by the 9 digits of an individual face.

Generally, a specified position is in the standard orientation; when asking Digicube to solve the position, for example, you will get an error message if it is not. But if your current position is in a different orientation (because of the way it was entered, or because of previous operations), it is easy to correct it: simply specify the one to three turns needed to reorient it; or specify the reset operation R, and Digicube will perform those turns.

Position names

Some positions have special names. *Solved position* is the position where all pieces in each face have the color of the center piece and the cube is in the standard orientation. *Current position* is the position that you are working with, the result of previous operations, moves, or turns. *Initial position* is the position before an operation, typically the position for which a solution is sought. *End position* is the position following an operation, typically after a step in the solution. Thus, the current position could be the end position of one step and the initial position of the next.

Goal position is a partially specified position: some pieces have 9 instead of regular color digits, which means that any colors are acceptable. Goal positions are used as target in partial solutions, where only some of the pieces need to be correct, or in comparisons where only some of the pieces need to match. *Current goal position* is the goal position that you are working with. *Anchor goal position* and *corners goal position* are special positions used in solutions; they are described in section 5.8, "Solutions".

Stored positions

During a run, Digicube stores several positions, which you can access: the current position, the current goal position, and 99 general-purpose positions. At the beginning of a run, they are initialized with some generally useful actual positions (see below), but you can modify them at any time. Changing from script to interactive mode or vice versa (with the operation I) does not alter these positions. They are not stored anywhere when the run ends, so if you want to preserve them you must display them, which records them in the output file. The memory-related operations are discussed in section 5.7, "Memories".

The most important position is the current position: moves, turns, display, solutions, and other operations use the current position. At the beginning of a run, the current position is identical to the solved position.

The current goal position is needed only for those solutions and comparisons that use goal positions. This position will normally use 9's for some of the pieces, as mentioned earlier, but you can also use a fully specified goal position as target. You can enter 9's anywhere in the goal position, but when used as target for a solution, the center pieces cannot be 9 (since they determine the orientation) and there can be no partial 9's (if a piece has 9's, all its faces must be 9); also, the anchor must be correct. At the beginning of a run, the current goal position is identical to the anchor goal position.

The 99 general-purpose positions are called *memories*, and are known by their number, 1 to 99. They are used for saving the current position or the current goal position, if you need them later in the current run. They are also used to store positions that must be compared with the changing current position; for example, you can instruct Digicube to compare the current position with several memories while performing thousands of moves, and display a message when it matches one of those memories.

At the beginning of a run (or when changing the current model to M1), memories 1 to 96 have the solved position, memory 97 has the corners goal position, and memory 98 has the anchor goal position. Memory 99 has correct center pieces and 9's for all edge and corner pieces. Apart from their initial contents, all memories are identical.

3.3 Pieces

The two conversion tables we discussed earlier, digits-to-sides and digits-to-colors, can also be used to assign the 6 digits to the cube's 26 pieces. This is needed only for operations that deal with individual pieces, not for basic operations like entering or solving a position.

Pieces have their own small faces, which have a color and are part of the cube's faces: center pieces have one face, edge pieces have two, and corner pieces have three. We can identify a piece by its location and its colors; and both can be expressed as unique numbers of one, two, or three digits, depending on the piece's number of faces.

Starting with locations, we use the cube's sides. Recall the digits-to-sides conversion table:

1=right, 2=left, 3=front, 4=back, 5=down, 6=up.

Like the cube's sides, the 26 piece locations are fixed numbers: they express fixed places within the frame of reference formed by the 6 sides (the imaginary cube we discussed earlier, within which the real cube rotates). So they do not depend on the cube's orientation, or on whether the cube is solved or scrambled. A location fit for a center piece can be occupied at various times by any one of the 6 center pieces; one fit for an edge piece by any one of the 12 edge pieces; and one fit for a corner piece by any one of the 8 corner pieces.

The location of a center piece is a one-digit number – the side where the piece resides: 1, 2, 3, 4, 5, 6. An edge piece has two faces, so it involves two sides and its location is a number made up of the two digits that identify the sides: 13, 14, 15, 16, 23, 24, 25, 26, 35, 36, 45, 46. A corner piece has three faces, so it involves three sides and its location is a three-digit number: 135, 136, 145, 146, 235, 236, 245, 246.

Note that the numbers are formed by using the digits in ascending order; thus, 13 (not 31), 46 (not 64), 135 (not 153, 315, 531, etc.), 246 (not 426, 462, 624, etc.). In other words, we form a location number by looking at its one or two or three sides in a precise, numerical order. So there is a unique correspondence: for a given location we can recall its number by converting sides to digits, and for a given number its location by converting digits to sides.

Next, we must assign numbers to the combinations of colors of the 26 pieces. Recall the digits-to-colors conversion table, which depends on the choice of the standard orientation. The choice we made was:

right=orange, left=red, front=blue, back=green, down=yellow, up=white.

And the resulting digits-to-colors conversion table (due to this choice and the fixed digits-to-sides table shown earlier) was:

1=orange, 2=red, 3=blue, 4=green, 5=yellow, 6=white.

The color number of a piece is then simply a combination of these color digits: one-digit numbers for the center pieces, two-digit numbers for the edge pieces, and three-digit numbers for the corner pieces. For example, 3 for the center piece blue, 25 for the edge piece red-yellow, 146 for the corner piece orange-green-white. The numbers will be unique, because the 26 pieces have unique combinations of colors.

However, unlike the location numbers, the color numbers are not restricted to one arrangement of their digits. They must also identify the orientation of the piece, and it is through the order of digits that they do it. When the cube is solved and in the standard orientation, the color number of each piece is the same as its location number (because of the relationship between the two tables, digits-to-sides and digits-to-colors). Otherwise, both the location and the color numbers are needed to describe a piece.

A center piece, in a given location, can have only one orientation. An edge piece can have 2 orientations. For example, the piece with color 35, when in location 16 (*right*, *up*), can have: 3 *right*, 5 *up*, or 5 *right*, 3 *up*; so we need two numbers, 35 and 53, to express this. A corner piece can have 6 orientations, but in any one location only 3 of them are possible with a real cube. For example, the piece with color 236, when in location 136 (*right*, *front*, *up*), can have: 2 *right*, 6 *front*, 3 *up*, or 6 *right*, 3 *front*, 2 *up*, or 3 *right*, 2 *front*, 6 *up*; so we need 3 numbers, 263, 632, and 326, to express this. In location 135, the same corner piece can have its other 3 orientations, and the color numbers are 236, 362, and 623.

In practice, the color numbers are easy to determine: like the location numbers, we express the colors of a piece by looking at its one or two or three sides in a precise, numerical order. But now we use the color digits of those sides, instead of the side digits, to form the number.

For example, when we say that the center piece at location 1 has color 4, we mean that color 4 is at side 1 (green *right*); when we say that the edge piece at location 23 has colors 16, we mean that color 1 is at side 2, and color 6 is at side 3 (orange *left*, white *front*); when we say that the corner piece at location 246 has colors 315, we mean that color 3 is at side 2, color 1 is at side 4, and color 5 is at side 6 (blue *left*, orange *back*, yellow *up*).

To describe a piece, we use the expression **^loc=col** (location and colors). Thus, the three examples above are written as ^1=4, ^23=16, ^246=315. As mentioned, for the solved cube in the standard orientation the location number is the same as the color number for all pieces, so the position, expressed as a list of pieces, looks like this:

```
^135=135 ^136=136 ^145=145 ^146=146
^235=235 ^236=236 ^245=245 ^246=246
^13=13 ^14=14 ^15=15 ^16=16
^23=23 ^24=24 ^25=25 ^26=26
^35=35 ^36=36 ^45=45 ^46=46
^1=1 ^2=2 ^3=3 ^4=4 ^5=5 ^6=6
```

And here is the position of a scrambled cube in the standard orientation, shown as list of pieces, list of colors, rows of colors, and schematic, so you can see how the pieces correspond in the four formats:

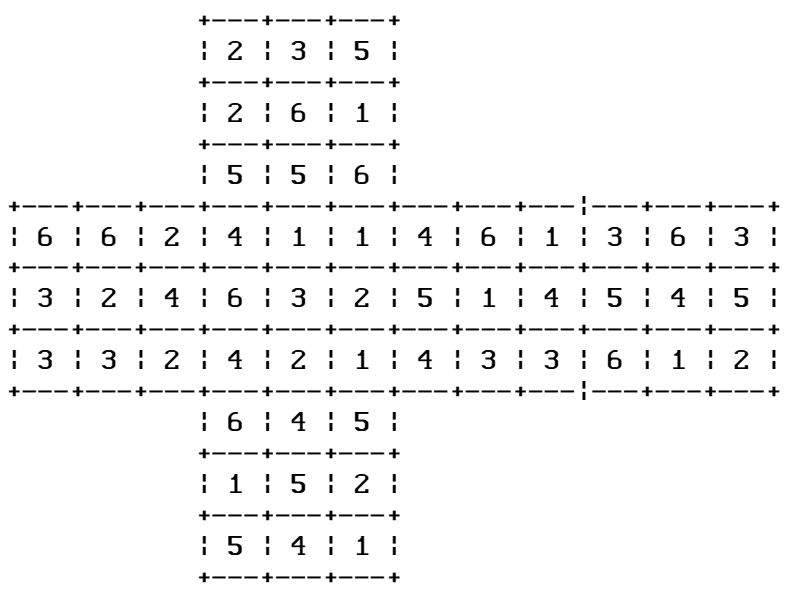
```
^135=415 ^136=416 ^145=361 ^146=135
^235=246 ^236=245 ^245=325 ^246=632
^13=52 ^14=45 ^15=32 ^16=61
^23=46 ^24=35 ^25=31 ^26=62
^35=24 ^36=15 ^45=14 ^46=63
^1=1 ^2=2 ^3=3 ^4=4 ^5=5 ^6=6

4 6 1 5 1 4 4 3 3   6 6 2 3 2 4 3 3 2   4 1 1 6 3 2 4 2 1
3 6 3 5 4 5 6 1 2   6 4 5 1 5 2 5 4 1   2 3 5 2 6 1 5 5 6
```

```
        2 3 5
        2 6 1
        5 5 6

6 6 2   4 1 1   4 6 1   3 6 3
3 2 4   6 3 2   5 1 4   5 4 5
3 3 2   4 2 1   4 3 3   6 1 2

        6 4 5
        1 5 2
        5 4 1
```



The list of pieces is the most convenient format when we study individual pieces. The schematic is the most useful one when we must visualize the whole cube. The list of colors is the most economical one: the simplest way to enter, display, store, and compare positions. The rows of colors show the individual faces well, like the schematic but in a more compact way.

The rows and the schematic are strictly output formats, but the list of colors and the list of pieces are used for both input and output. The list of colors, when used as input, is specified in its entirety in one operation (in scripts), or is broken down into 6 operations with shorter lists, one for each face (in scripts or interactively). The list of pieces, when used as input, is specified as up to 26 operations (one for each piece that must be set up, in any order). For both types of lists, the lines added to the output file by a display operation (as seen in the examples above) can be copied directly into the input file and used to specify that position in a script. Once specified, the position can be displayed in a different format.

3.4 Moves and turns

The rotations (moves and turns) specified in a script or interactively modify the current position, thus simulating the way a rotation would modify a real cube. Rotations can be specified one at a time or several in a sequence.

A *move* is the rotation of a face (layer) 90 degrees relative to the rest of the cube. It is identified by a digit, which refers to the side where the rotated face resides, as discussed earlier: 1=*right*, 2=*left*, 3=*front*, 4=*back*, 5=*down*, 6=*up*. The cube can be in any orientation. To emphasize, move 1 rotates the face at side 1, regardless of which face is there (face 1 sometimes, otherwise another face); move 2 rotates the face at side 2; and so on.

A move can be clockwise (cw) or counterclockwise (ccw). When ccw, the digit is prefixed with "-", conveniently looking like a negative number. The direction cw or ccw is from the perspective of the rotated face, not the cube's user. The *type* of a move refers to both directions; thus, there are 6 move types, 1 to 6, and therefore 12 possible moves.

Digicube has only 90-degree moves; a 180-degree rotation is treated as 2 identical consecutive moves. A move affects only 4 corner pieces and 4 edge pieces, and does not affect the center piece of the rotated face; thus, it does not alter the cube's orientation. As example of moves, the following sequence will cause the cube to return to the position it starts with:

```
3 5 -1 -5 1 3 -5 -3 5 1 -3 -1
```

A *turn* is the rotation of the whole cube 90 degrees, cw or ccw. It can be described as a move where the rest of the cube rotates together with the face, and is identified by the digit (and "-" if ccw) that such a move would have, enclosed in parentheses.

Digicube has only 90-degree turns; a 180-degree rotation is treated as 2 identical consecutive turns. A turn does not affect the arrangement of the cube's pieces, but affects 4 center pieces, and hence alters the cube's orientation. As example of turns, the following sequence will cause the cube to return to the position it starts with:

```
(2) (5) (-4) (-5) (-3) (6) (6) (3)
```

Note that a turn is equivalent to the reverse turn of the opposite side: (1) is the same as (-2), (-1) is the same as (2), and similarly for sides 3/4 and 5/6.

Moves and turns can be mixed in a sequence, and sequences in a script can span multiple lines and can have any length. They are limited to 30 moves and/or turns per line in interactive mode. Digits must be separated by one or more spaces, but spaces are optional around parentheses. The only restriction is that a turn and its enclosing parentheses be together on the same line of a script. All elements in the following sequence are valid:

```
( 1 ) ( -3 ) -5 (1) (-3) (1)(-3)(5) 1 1(3)(5)3 (-5)-3 1(-3)-1(5)3(1)(-5)
```

When a long sequence consists of a repeated pattern of moves or turns, it can be specified as a *loop*, by enclosing it in square brackets and preceding it with a repeat count (see chapters 6, "Interactive mode" and 7, "Scripts"). The following example shows a sequence of moves and turns which, repeated 120 times, causes the cube to return to the position it starts with:

```
[120 (5) 3 (-5) 3 (-3) 5 (3) 5 3]
```

Digicube uses only moves of type 1, 3, and 5 in solutions, but your sequences can have moves and turns of all types, 1 to 6. Also, you can rotate the middle layers too, by combining appropriate moves and turns. For example, to rotate the horizontal middle layer 90 degrees clockwise (as seen from above), rotate the whole cube, then rotate back the faces at sides 5 and 6. This sequence is shown below, together with the end position in two display formats (if starting from the solved position):

```
(6) -6 5

  1 1 1 4 4 4 1 1 1   2 2 2 3 3 3 2 2 2   3 3 3 1 1 1 3 3 3
  4 4 4 2 2 2 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  3 3 3  1 1 1  4 4 4  2 2 2
  2 2 2  3 3 3  1 1 1  4 4 4

        5 5 5
        5 5 5
        5 5 5
```

If you specify the digit 0 for a move or a turn, Digicube generates a random move or turn (see section 5.9, “Random moves and turns”). This is useful in experiments, or to scramble the cube. You can select moves of either 3 or 6 types with the operation N3/N6. The actual generated values can be displayed, if you want, using the operation ON/OF.

CHAPTER 4

Run options

4.1 Introduction

The run options are depicted with letters (sometimes followed by a number), separated by spaces. You can enter the letters in either upper or lower case. (In this manual they are in upper case, for clarity, when mixed with text.) For ease of recall, the letters match, as far as possible, the initial letter of a word describing the option (D for Display, R for Reverse, etc., and that word is capitalized in the option's description). The options can be entered in any order. The options list must not exceed 125 characters. A message is displayed if there is an error (see chapter 12, "Error messages").

You specify the options before a run, at the prompt "Enter run options (X to end, H for help, I for interactive)", and they apply only to that run. Many runs need only one option, or no options at all. When you don't specify an option, its default value takes effect. For convenience, you can use the arrow keys and the editing keys to recall and edit options entered in previous runs of the same execution (that is, since you started Digicube). Most options can also be set and reset during the run, through the operations "+" and "–": setting an option is equivalent to specifying it, and resetting it restores the default value.

So, depending on your needs and preferences, you can specify an option before the run, or set and reset it during the run. For example, if you execute a certain script repeatedly, it is simpler to set the options in the script; but if you are experimenting and modifying the options frequently, it may be simpler to specify them before each run than to modify the script. Here are a few examples of run options, as they might be entered before a run:

```
m2 d1
d w
s6
iscript-1 oresults-1
```

The options file digopt.txt lets you store sets of run options, so you don't have to type them every time (see run option G). This is useful when there are long input and output file names, or if you use the same sets of options repeatedly.

The next section is a summary of the run options; the following section describes them in detail. Certain options, listed separately at the end, are special; they are more akin to functions, and have priority over the regular options (see section 4.4, "Priority").

4.2 Summary of options

M1, **M2**, **M3**: Specify Model.
D: Display the initial position in solutions as a list of colors.
D1, **D2**: Display also the intermediate positions in solutions.
J: Display the initial position in solutions as rows of colors.
U: Display the initial position in solutions as a list of pieces.
A: Display the initial position in solutions as a cube schematic.
R: Show also the Reverse moves in solutions.
W: Show also Widely spaced moves in solutions.
N: Create New output file.
K: Cancel writing to output file.
I: Specify Input file or Interactive run.
S1–**S999**: Start reading the input file at label #1–#999.
O: Specify Output file.

G1–**G999**: Get the run options at label #1–#999 in the options file.
C1–**C99**: Perform Custom operation 1–99.
C: List the Custom operations that are currently implemented.
H: Help – display the run options with brief descriptions.
L: List the operations with brief descriptions.
Z: Display Digicube license information.
X: End Digicube.

4.3 Options

M1, **M2**, **M3**: Specify Model. M1 is the classic 3× Rubik's cube, M2 is the 2× cube (known as Pocket cube), M3 is the 3× pyramid (known as Pyraminx). The default, if you omit the option, is M1. The choice of model affects several features; for example, some operations are valid only for M1, and others only for M1 and M2. The error message "Invalid for current model" is caused by such an operation, or if using in M3 the digit 5 or 6 for a color, a move, or a turn.

D: Display the initial position in solutions as a list of colors. The default is No. D will display, before the move sequences of a solution, the position being solved (which was specified in a script, was entered interactively, or is the result of previous operations), with the title "Initial". The display is the list of color digits (see section 3.2, "Positions"). Here is an example:

```
Initial:  6 1 4 1 1 1 3 5 2   5 2 2 6 2 2 6 2 5   6 2 4 3 3 5 3 3 6
          1 6 2 4 4 3 5 4 3   1 1 2 5 5 3 1 5 4   3 4 5 4 6 6 4 6 1
```

Options J, U, and A, described below, are similar to D, but the four are independent of one another: you can specify any combination of them.

D1, **D2**: Display also the intermediate positions in solutions. The default is No. These options are an extension of D: D, D1, and D2 are mutually exclusive, as D1 includes the D display, and D2 includes the D1 display. D1 will display, in addition to the initial position, the position reached after each step of the solution, with the title "End". D2 will display, in addition to this, before each step, the goal position of that step (what position the step will try to reach), with the title "Goal". Here is how the first two steps of a solution are displayed with D2:

```
Initial:  4 5 6 2 1 6 4 6 4   6 6 2 4 2 3 5 5 5   3 3 1 1 3 4 3 1 5
          4 5 1 2 4 6 5 1 3   1 4 2 4 5 3 2 5 1   3 2 2 1 6 3 6 2 6

Goal:     9 9 9 9 1 9 9 9 9   2 2 9 2 2 9 9 9 9   9 9 9 9 3 9 9 9 9
          9 4 4 9 4 4 9 9 9   9 9 9 9 5 9 9 9 9   6 6 9 6 6 9 9 9 9
1. Anchor: ..... .. -3 -6 4 -5 -6 -1 4
End:      6 5 2 6 1 4 6 4 2   2 2 2 2 2 2 4 1 3   4 5 4 3 3 1 1 6 3
          5 4 4 5 4 4 6 5 1   5 3 1 3 5 1 5 3 3   6 6 3 6 6 1 5 2 1

Goal:     1 9 1 9 1 9 1 9 1   2 2 2 2 2 9 2 9 2   3 9 3 9 3 9 3 9 3
          4 4 4 9 4 4 4 9 4   5 9 5 9 5 9 5 9 5   6 6 6 6 6 9 6 9 6
2. Corners: ..... ..... . -1 3 5 1 -5 1 1 3 -1 -5 -3
End:      1 1 1 3 1 2 1 5 1   2 2 2 2 2 3 2 5 2   3 1 3 6 3 2 3 4 3
          4 4 4 5 4 4 4 1 4   5 1 5 4 5 3 5 3 5   6 6 6 6 6 5 6 6 6
```

J: Display the initial position in solutions as rows of colors. The default is No. J is like D, but displays the colors of each face as 3 rows of 3 pieces (see section 3.2, "Positions"). Here is how the position shown above for D is displayed:

```
Initial:
          3 4 5
          4 6 6
          4 6 1

  5 2 2   6 2 4   6 1 4   1 6 2
  6 2 2   3 3 5   1 1 1   4 4 3
  6 2 5   3 3 6   3 5 2   5 4 3

          1 1 2
          5 5 3
          1 5 4
```

U: Display the initial position in solutions as a list of pieces. The default is No. U is like D, but displays the location and colors of the pieces in the form **^loc=col** (location and colors, as explained in section 3.3, "Pieces"). Here is how the position shown above for D is displayed:

```
Initial:  ^135=362 ^136=641 ^145=254 ^146=415
          ^235=531 ^236=264 ^245=631 ^246=523
          ^13=15 ^14=14 ^15=53 ^16=16
          ^23=23 ^24=63 ^25=25 ^26=24
          ^35=31 ^36=26 ^45=45 ^46=64
          ^1=1 ^2=2 ^3=3 ^4=4 ^5=5 ^6=6
```

A: Display the initial position in solutions as a cube schematic. The default is No. A is like D, but displays the 2-dimensional cube schematic. Here is how the position shown above for D is displayed:

```
Initial:
                +---+---+---+
                ¦ 3 ¦ 4 ¦ 5 ¦
                +---+---+---+
                ¦ 4 ¦ 6 ¦ 6 ¦
                +---+---+---+
                ¦ 4 ¦ 6 ¦ 1 ¦
  +---+---+---+---+---+---+---+---+---+---+---+---+
  ¦ 5 ¦ 2 ¦ 2 ¦ 6 ¦ 2 ¦ 4 ¦ 6 ¦ 1 ¦ 4 ¦ 1 ¦ 6 ¦ 2 ¦
  +---+---+---+---+---+---+---+---+---+---+---+---+
  ¦ 6 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 5 ¦ 1 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦ 3 ¦
  +---+---+---+---+---+---+---+---+---+---+---+---+
  ¦ 6 ¦ 2 ¦ 5 ¦ 3 ¦ 3 ¦ 6 ¦ 3 ¦ 5 ¦ 2 ¦ 5 ¦ 4 ¦ 3 ¦
  +---+---+---+---+---+---+---+---+---+---+---+---+
                ¦ 1 ¦ 1 ¦ 2 ¦
                +---+---+---+
                ¦ 5 ¦ 5 ¦ 3 ¦
                +---+---+---+
                ¦ 1 ¦ 5 ¦ 4 ¦
                +---+---+---+
```

R: Show also the Reverse moves in solutions. The default is No. The reverse moves are the moves that, if applied to the final, solved position, will result in the initial position (the one before the solution). The display consists of the solution steps listed in reverse, and the moves in each step reversed and listed backwards. The reverse moves are shown following the regular display of the solution steps. Here is an example:

```
1. Anchor: ..... . 5 5 -3 1 -2 -6
2. Corners: ..... ..... .. 1 5 -1 3 3 -1 3 -1 -5 1 -3 -5
3. Edges: ..... ..... 1 -3 -1 -3 -1 -3 1 3 1 3  >> 25 36
4. Edges: ..... ..... 1 3 1 5 3 -5 -1 -3 -1 -5  >> (15 45)
5. Edges: ..... ..... 3 1 3 -1 -3 -1 -3 -1 3 1  >> 14 23
6. Flip 15 45:  1 5 1 -5 -1 -5 3 1 5 5 -1 -5 -3 -5 -1 5
Moves: 64    Time: 1.8 sec    Sequences tried: 36,861,360

Reverse moves
1.  -5 1 5 3 5 1 -5 -5 -1 -3 5 1 5 -1 -5 -1
2.  -1 -3 1 3 1 3 1 -3 -1 -3
3.  5 1 3 1 5 -3 -5 -1 -3 -1
4.  -3 -1 -3 -1 3 1 3 1 3 -1
5.  5 3 -1 5 1 -3 1 -3 -3 1 -5 -1
6.  6 2 -1 3 -5 -5
```

R lets you re-create the initial scrambled position with a real cube after solving it. In general, it lets you create with a real cube any valid position, by specifying that position with the operation P and then solving it and applying the reverse moves to a solved cube. You can use this method, for example, to create interesting color patterns.

R also lets you reach with a real cube any position from any other position, by using the solved position as intermediary: enter and solve the first position, then enter and solve the target position showing the reverse moves, and finally apply these moves to the solved cube.

W: Show also Widely spaced moves in solutions. The default is No. W repeats the regular solution display while adding extra space between lines and between moves. This makes it easier to read in a small window, when using it to solve a real cube. W applies only to the display produced by solutions (in scripts or interactively), not to everything displayed by the program.

N: Create New output file. N deletes the existing output file and creates a new one at the beginning of the current run. The default is to add the new lines at the end of the existing file. The file name is assumed to be digout.txt, unless the option O is used to specify a different name. This option is useful when the file has become too large, or when you no longer need the past results and want the new ones to start at the beginning of the file. You can also delete the output file yourself before starting Digicube, and a new one will be created automatically.

K: Cancel writing to output file. K prevents Digicube from adding the displayed lines to the output file. The default is to add the lines to the file. This option is useful when you expect a large number of displayed lines in the following operations and don't wish to save them.

I: Specify Input file name or Interactive run. I is used for two options, but there is no conflict, since the options are mutually exclusive. If specified alone, I makes the current run an interactive run: the program begins interactive mode from the start, and when you type I to end interactive mode, the run too ends (see chapter 6, "Interactive mode"). In such a run there is no need for a script, and hence for an input file, since all input is from the keyboard. (In a regular run, the program begins interactive mode from a script, and returns to the script when interactive mode ends.)

If I is followed by a file name, that file will be used as input file in the current run; the run is then a regular run, and Digicube will execute a script from the file. The name can include a folders path, but must have no extension (the extension is always **.txt**). The whole name string can be up to 100 characters long, and must follow the "I" with no intervening spaces. You can enclose the string in double quotation marks if you wish; if the names in the string contain spaces, the quotation marks are mandatory. Here are some examples:

```
iinpfl12
i\scripts\inpfl12
i"c:\digicube\script files\script_1"
```

To avoid entering long file names repeatedly, you can store sets of run options in a file (see run option G).

The default, if I is omitted, is to use an input file called **diginp.txt**, located in the same folder as the program file, digicube.exe. Also, if you use I and specify a file but no path, the file is assumed to be located in that folder.

Apart from allowing you to override the default file name and location, the option I lets you execute scripts stored in several files.

If you have no script, you must use the option I so as to start an interactive run; otherwise Digicube will start a regular run and attempt to execute whatever it finds in diginp.txt (or display an error message if the file does not exist).

S1–S999: Start reading the input file at label #1–#999. The default is to read the file starting at the beginning. But if you want to bypass a portion at the beginning, or if there are several scripts in the file, insert a label where the required script starts, and use the option S with the same number. The label is specified as #1, #23, etc. It must start in the first position on a line, and the script can start on the same line (after one or more spaces) or on the following line.

If you want to read the file from the beginning and the first script has a label, you must use S and specify that label, else the label will be seen as an operation and will cause an error message. If you want to omit the S, delete the label.

Labels have no numeric significance, and need not be numerically consecutive: Digicube simply looks, starting at the beginning of the file, for the first line with the given label. (See chapter 7, "Scripts", for examples of labels and scripts.)

O: Specify Output file name. O must be followed by a file name, and that file will be used as output file in the current run. The name can include a folders path, but must have no extension (the extension is always **.txt**). The whole name string can be up to 100 characters long, and must follow the "O" with no intervening spaces. You can enclose the string in double quotation marks if you wish; if the names in the string contain spaces, the quotation marks are mandatory. Here are some examples:

```
ooutfl12
o\logs\outfl12
o"c:\digicube\output files\ofile_1"
```

To avoid entering long file names repeatedly, you can store sets of run options in a file (see run option G).

The default, if O is omitted, is to use an output file called **digout.txt**, located in the same folder as the program file, digicube.exe. Also, if you use O and specify a file but no path, the file is assumed to be located in that folder. With a specified name or with the default, if the file does not exist, Digicube will create it.

Apart from allowing you to override the default file name and location, the option O is useful if you want the lines displayed in the current run to be added to a separate file. The option is essential if you execute several instances of Digicube simultaneously, otherwise they would all add lines to the same file at the same time, and the lines would be jumbled up.

Special options

G1–G999: Get the run options at label #1–#999 in the options file. G lets you store sets of run options in a file. This is useful when you need the same set of options on different occasions, when there are many options, when the I and O options have long file names, and so on. The file, called options file, must have the name **digopt.txt**. It is an ordinary text file and can be accessed with any

text editor (Notepad, for example). It must reside in the same folder as the program file, digicube.exe, and is needed only if you use the option G.

In the file, the label is specified as #1, #23, etc. (the number you use with G). It must start in the first position on a line, and the options are listed on the same line (after one or more spaces). The actual options list must not exceed 125 characters. Labels have no numeric significance, and need not be numerically consecutive. Also, any number of lines, blank or not, may exist between the option lines. Digicube simply looks, starting at the beginning of the file, for the first line with the given label. Here is an example of an options file:

```
#1 m2 d2 ooutfl21
#5 c1 d u w
use this for m3 trials:
#23 m3 iscriptfile1 o"c:\text files\log file1"
```

The list of options in the options file must not include the option G. If it does, the error message is "Redirection invalid in options file".

C1-C99: Perform Custom operation 1-99. The custom operations are functions that cannot be conveniently specified through a script or interactively (for example, they require several values to be entered). They are also the means to add new functions to Digicube (through programming). They are numbered 1 to 99, but only a few are currently implemented. Depending on its design, a custom operation may display lines in the output file, read scripts in the input file, and use run options, similarly to regular operations. (See chapter 11, "Custom operations".)

C: List the Custom operations that are currently implemented.

H: Help – display the run options with brief descriptions.

L: List the operations with brief descriptions. These are the operations used in scripts and in interactive mode (see chapter 5, "Operations").

Z: Display Digicube license information.

X: End Digicube.

4.4 Priority

The options G, C, H, L, Z, and X are special. They are more akin to functions, and have priority over the regular options. That is, if you include one of them in a list of options, *it* is executed and the regular options are ignored. For example, X will end the program, H will display the help list, and G will redirect to the options file, regardless of any regular options included. C1-C99 have no priority, and can be used together with regular options.

If you include several special options, their priority is: X H L C Z G. Thus, X will end the program even if another special option is included; H will display the help list even if L, C, Z, or G is included; and so on. G is executed only if it is the only special option. Also, with G, the priority rules described here apply equally to the list found in the options file. In all cases, the order in which the options are specified in a list is irrelevant.

CHAPTER 5

Operations

5.1 Introduction

The operations are depicted with one or two letters, sometimes followed by a number. You can enter the letters in either upper or lower case. (In this manual they are in upper case, for clarity, when mixed with text.) For ease of recall, the letters match, as far as possible, the initial letter of a word describing the operation (D for Display, S for Solve, etc., and that word is capitalized in the operation's description). Apart from I (for interactive) and D, J, U, and A (for display format), an operation is unrelated to the run option that has the same letter. A few operations are depicted with a symbol and, for clarity, are enclosed in quotation marks in their description: "^", "/", "\", "+", "–", "*". Omit the quotation marks when using the operation.

The option L, at the prompt "Enter run options …", displays a list of the available operations with brief descriptions, as a handy reminder. If you need it while past that prompt (in interactive mode, for example), start a second instance of Digicube. (But if you intend to perform operations there too, make sure you specify a different output file, else the display lines of the two instances will be jumbled up.)

The operations can be performed from a script or in interactive mode. With a few exceptions, they are identical in scripts and interactively. In a script, operations are entered consecutively, separated by one or more spaces. There can be any number of operations on a line, and a sequence of operations can span any number of lines. Sequences of moves and turns can be mixed with operations. Square brackets are used to define a loop (a group of elements that is to be repeated). Interactively, operations are entered and executed one at a time; only moves and turns can be entered several at a time and as a loop.

Interactively, an error message ends the operation that caused it and returns to the prompt, where you can enter the next operation. In a script, an error message usually ends the run (so you must correct the script and re-execute it); for some errors, a message is displayed with the choice to end the run or to skip the operation that caused it and continue with the next one.

The next section is a summary of the operations, grouped by function in several categories. Each category is then discussed in a separate section.

5.2 Summary of operations

Positions

P: Specify the current Position.
P1–**P6**: Specify side 1–6 for the current Position.
PG: Specify the current Goal Position.
PG1–**PG6**: Specify side 1–6 for the current Goal Position.
E: Exchange the current position with the current goal position.
V: Verify the current position.
VG: Verify the current Goal position.
R: Reset the current position to the standard orientation.
RG: Reset the current Goal position to the standard orientation.
Y: Generate a scrambled position by performing moves and turns.
YP: Generate a scrambled position by rearranging the Pieces.
YT: Generate a scrambled position by using the current goal position as Template.

Position display

D: Display the current position as a list of color digits.
J: Display the current position as rows of color digits.
U: Display the current position as a list of pieces.
A: Display the current position as a cube schematic.
DG: Display the current Goal position as a list of color digits.
JG: Display the current Goal position as rows of color digits.
UG: Display the current Goal position as a list of pieces.
AG: Display the current Goal position as a cube schematic.

Miscellaneous display

L: Display Label.
"*": Display comment.
ON, **OF**: Switch On/Off the display of input moves and turns.

Counts

Q: Display the current counts of moves, turns, and cycles.
Q0: Clear the counts of moves, turns, and cycles.
QC: Display the current cycle.

Memories

T1–**T99**: Copy the current position To memory 1–99.
F1–**F99**: Copy the current position From memory 1–99.
TG1–**TG99**: Copy the current Goal position To memory 1–99.
FG1–**FG99**: Copy the current Goal position From memory 1–99.
C1–**C99**: Compare the current position with memory 1–99.
CC1–**CC99**: Compare Constantly the current position with memory 1–99.
CC: Display the CC list.
CC0: Clear the CC list.
G1–**G99**: Compare the current position with memory 1–99 as Goal.
GG1–**GG99**: Compare constantly the current position with memory 1–99 as Goal.
GG: Display the GG list.
GG0: Clear the GG list.

Solutions

S: Solve the current position.
SA: Solve the current position for Anchor.
SC: Solve the current position for Corners.
SE: Solve the current position for Edges.
SD: Solve the current position Directly.
SG: Solve the current position for Goal.
M: Choose order of trial Moves in solutions.
K1–**K20**: Stop after trying sequences of 1–20 moves in solutions.

Random moves and turns

N3, **N6**: Use 3 or 6 faces in random moves.
NS, **NR**: Save/Restore the program's random state.

Individual pieces

"^": Set the colors of one piece in the current position.
"/", "\": Flip clockwise or counterclockwise one piece in the current position.

Miscellaneous operations

"+", "–": Set/reset the current run options.
I: Begin or end Interactive mode.
Z: End script in custom operation.
X: End current run.

5.3 Positions

P: Specify the current Position. (P is valid only in scripts; in interactive mode you must use P1–P6, described later.) P must be followed by a list of 54 color digits, depicting the pieces of a position (see section 3.2, "Positions"). The list is similar to the list displayed by the operation D. (Thus, to re-create a position displayed in a previous run, copy those lines from the output file into a script in the input file, and add the P.)

To specify the position of a real cube, enter first the 9 digits of the face at side 1 (*right*), then those of the face at side 2 (*left*), then 3 (*front*), 4 (*back*), 5 (*down*), and 6 (*up*). Normally, the position is in the standard orientation, so faces correspond to sides: face 1 is at side 1, face 2 is at side 2, and so on.

Within each face, the pieces are specified as 3 rows of 3 digits, top to bottom, and left to right in each row. The top of faces 1, 2, 3, and 4 is defined as their upper edge; for face 5, the top is the edge shared with face 3; for face 6, it is the edge shared with face 4. This is also how the faces appear in the 2-dimensional cube schematic displayed by the operation A.

In practice, you first turn the cube so that it is in the standard orientation (by watching the center pieces); face 3 will be *front*, and face 6 will be *up*. Then, with face 5 staying *down*, you turn it so as to see faces 1, 2, 3, and 4, in this order, and enter the 9 colors of each one as explained above. Then, with face 3 again as *front*, you tilt the cube back so as to expose face 5, and enter its 9 colors. Then you restore face 6 as *up*, and enter its 9 colors. At all times, you use the center pieces to identify the faces and the cube's orientation.

As example, here is how you might specify the position of a scrambled cube:

```
p  3 5 2 2 1 6 5 5 5   5 3 1 6 2 2 3 1 6   4 2 5 3 3 6 1 4 4
   6 2 3 1 4 4 4 3 6   3 1 1 5 5 3 2 6 2   1 4 4 1 6 4 6 5 2
```

The digits must be separated by one or more spaces, and can span one or more lines. They can be grouped in any pattern, but it is a good idea to be consistent, and to group them as faces, resembling the way they are displayed by the operation D. This helps if you need to compare positions, or to detect visually such errors as entering too few or too many digits for a certain face. If you enter more than 54 digits, the superfluous ones will be interpreted as moves. If you enter fewer, the following element in the script will likely cause an error message (but if it is a move, it will be interpreted as a color digit belonging to the P specification).

At this point, Digicube only checks that the digits are valid colors, 1 to 6, and will point out an invalid one with an error message. The digit 9 (for partially specified positions), normally used with goal positions, is also valid with P; this provides the flexibility to create and modify a goal position as the current position (if you need the operations "^", "/", or "\", possible only with the current position), and move it later to the current goal position.

Even with correct digits, however, the position may be invalid. (A position is deemed invalid if it cannot be reached starting from the solved position and using only moves and turns.) A complete check is performed by V and S (and its variants); operations like displaying, storing, and comparing, as well as moves and turns, do not require a valid position.

Generally, you enter a position in the standard orientation; there rarely is a reason to enter it in a different orientation and, in any case, if you need that, you can alter its orientation with turns after

entering it. (Keep in mind that when you specify the position with P, the first face is always the one at side 1, the second is the one at side 2, and so on, regardless of the orientation you choose.) S and its variants, for example, require a position in the standard orientation. If you entered a position incorrectly, you can reorient it with the operation R (or by specifying the appropriate turns).

It is easy to confirm that a position is in the standard orientation, by noting the middle digits in each face (the 5th digit in each group of 9 digits). These digits correspond to the center pieces, whose color numbers in the standard orientation match the side numbers. Thus, since the 6 faces are always ordered by side number, 1 to 6, the middle digits in the standard orientation are also ordered as 1 to 6.

P1–**P6**: Specify side 1–6 for the current Position. These variants of P modify only one face, leaving the other faces unchanged; so they must be followed by only 9 digits. The number, 1 to 6, identifies the side: 1=*right*, 2=*left*, 3=*front*, 4=*back*, 5=*down*, 6=*up*. Thus, in addition to specifying a complete position, P1–P6 let you modify an existing one without having to enter all 54 digits, if only a few must change. As with P, the digits can be listed freely, but a consistent pattern is helpful.

Comparing these variants with P, P1 refers to the first group of 9 digits, P2 to the second group, and so on. For a complete position you must specify all sides, but you can list them in any order, since each side is a separate operation. Here are two ways the position shown earlier for P might be specified in a script:

```
p1 3 5 2 2 1 6 5 5 5  p2 5 3 1 6 2 2 3 1 6  p3 4 2 5 3 3 6 1 4 4
p4 6 2 3 1 4 4 4 3 6  p5 3 1 1 5 5 3 2 6 2  p6 1 4 4 1 6 4 6 5 2

p1  3 5 2 2 1 6 5 5 5
p2  5 3 1 6 2 2 3 1 6
p3  4 2 5 3 3 6 1 4 4
p4  6 2 3 1 4 4 4 3 6
p5  3 1 1 5 5 3 2 6 2
p6  1 4 4 1 6 4 6 5 2
```

In interactive mode, these variants of P are the only way to specify a position. Each side is entered as a separate operation. If an error is discovered later, you only need to re-enter the incorrect side.

When working interactively, remember that the only record of the position is in the output file (as all keyboard entries are added to the file). So if you need the position again later in the current run, you will have to re-enter it (by reading it in the output file). It is a good idea, therefore, to copy it to a memory as soon as you enter it, from where you can easily restore it.

PG: Specify the current Goal Position. (PG is valid only in scripts; in interactive mode you must use PG1–PG6, described below.) PG is like P, but affects the current goal position. Here is how you might use it (this is the corners goal position):

```
pg  1 9 1 9 1 9 1 9 1   2 2 2 2 2 9 2 9 2   3 9 3 9 3 9 3 9 3
    4 4 4 9 4 4 4 9 4   5 9 5 9 5 9 5 9 5   6 6 6 6 6 9 6 9 6
```

PG1–**PG6**: Specify side 1–6 for the current Goal Position. These variants of PG are like P1–P6, but affect the current goal position. Here is one way the position shown with PG might be specified:

```
pg1  1 9 1 9 1 9 1 9 1
pg2  2 2 2 2 2 9 2 9 2
pg3  3 9 3 9 3 9 3 9 3
pg4  4 4 4 9 4 4 4 9 4
pg5  5 9 5 9 5 9 5 9 5
pg6  6 6 6 6 6 9 6 9 6
```

In interactive mode, these variants of PG are the only way to specify a position; each side is entered as a separate operation.

E: Exchange the current position with the current goal position. E is useful if you want to work with the current goal position in the current position: one E makes it the current position while saving the latter as the current goal position, and a second E later restores them. You may need this maneuver in order to perform the operations "^", "/", or "\", possible only with the current position. For example, to flip two corners of the current goal position without modifying the current position, use this sequence:

```
e /135 \235 e
```

Another way to copy positions between the current and current goal positions is through memories, using the operations T, TG, F, and FG.

V: Verify the current position. V checks the current position and, if invalid, displays an error message describing the problem (see chapter 12, "Error messages"). A position is deemed invalid if it cannot be reached starting from the solved position and using only moves and turns. With Digicube, a position may be invalid due to incorrect specification with P and its variants, or incorrect manipulation of the individual pieces with "^", "/", or "\". Also, 9's found in the position are now considered invalid.

In addition, the position must be in the standard orientation (if it is not, the error message is "Cube incorrectly oriented"). A valid cube in a different orientation is still valid, of course, but this condition was added because in most situations where you may use V you also need the standard orientation. In particular, S and its variants require it. By making the V and S checks identical, you can be sure that a position accepted by V will also be accepted by S later. If the current position is not in the standard orientation, use R before V to reset it. If you want to verify it but keep its non-standard orientation, save it in a memory, perform R and V, and then restore it, as in this sequence:

```
t1 r v f1
```

If all checks are successful, V displays the message "Cube OK".

VG: Verify the current Goal position. VG is like V, but checks the current goal position. Here are the differences:

9's found in the position are valid, but there can be no partial 9's; that is, if a piece has 9's, all its faces must be 9. Also, the center pieces cannot be 9 (since they are needed to determine the orientation).

The anchor must be correct (see section 5.8, "Solutions"). The reason is that the current goal position is often used as target for SG (solve for goal), which also checks the anchor. (Because moves are restricted to three faces, with a wrong anchor it would be impossible to find a solution.) By making the VG and SG checks identical, you can be sure that a position accepted by VG will also be accepted by SG later. The message if wrong is "Goal anchor wrong".

If all checks are successful, the message is "Goal OK" if the current goal position has no 9's, and "Goal appears OK" if it has 9's. The reason for the qualified "OK" is that a partially specified position cannot be fully verified; its ultimate validity depends on the missing pieces.

R: Reset the current position to the standard orientation. Resetting the position means to reorient it if necessary (by performing one to three turns) so as to place it in the standard orientation. Digicube needs the center pieces to determine the orientation, and if these pieces are incorrect it displays the message "Invalid cube, cannot reset position". This can happen if you perform R before V or S and its variants (which would discover the problem through their validity checks). R only needs correct center pieces, and will reset the position even if it is otherwise invalid. If successful, R displays the message "Reset position".

The center pieces are incorrect if their colors are wrong, or if their relative arrangement is wrong. The following position, for example, where faces 1 and 2 were reversed, is not a solved cube in a non-standard orientation but an invalid cube: it is impossible to reach this position with a real cube, or to modify it with turns so as to reach the standard orientation.

```
2 2 2 2 2 2 2 2 2   1 1 1 1 1 1 1 1 1   3 3 3 3 3 3 3 3 3
4 4 4 4 4 4 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6
```

RG: Reset the current Goal position to the standard orientation. RG is like R, but affects the current goal position. Since the center pieces are needed to determine the orientation, they cannot be 9. If they are incorrect, the message is "Invalid goal, cannot reset position". If RG is successful, the message is "Reset goal position".

Y: Generate a scrambled position by performing moves and turns. The message "Scramble" is displayed. The current position is replaced with a randomly scrambled one, as follows. Starting with the solved position, Digicube performs, a number of times (the number is random in the range 190–210), a pair consisting of a random turn followed by a random move. The turns and moves involve all 6 faces, clockwise and counterclockwise (even when the operation N3 is in effect). The final position is then reset (placed in the standard orientation). Randomly scrambled positions are useful in experiments.

Each time it is used, Y generates a different position. See the operations NS/NR if you need to generate a random position identical to the one generated on a previous occasion.

If you prefer a different method of scrambling, you can design your own sequence of random moves and turns (see the examples in section 5.9, "Random moves and turns").

YP: Generate a scrambled position by rearranging the Pieces. The message "Scramble pieces" is displayed. The current position is replaced with a randomly scrambled one, as follows. Starting with the solved position, Digicube rearranges randomly the 12 edge pieces and the 8 corner pieces. Then it checks the final position and, if invalid, corrects it. Since the center pieces are not altered, the final position is in the standard orientation.

Each time it is used, YP generates a different position. See the operations NS/NR if you need to generate a random position identical to the one generated on a previous occasion.

YP is equivalent to disassembling a real cube and then reassembling the 20 pieces randomly; then, it may be necessary to physically flip or swap one or two pieces, to make the cube valid. YP is an alternative to Y, another way to generate a scrambled position in experiments. There is no practical difference between the results, and the choice is a matter of personal preference.

YT: Generate a scrambled position by using the current goal position as Template. The message "Scramble template" is displayed. The current position is replaced with a randomly scrambled one, as follows. First, the pieces with actual colors in the current goal position are copied into the corresponding locations in the current position. Then, the missing pieces (which have 9's in the current goal position) are rearranged randomly in the remaining locations. The final position is checked and, if invalid, corrected.

The current goal position is verified before the operation, as described for VG. However, for YT the anchor need not be correct, and the anchor pieces can even have 9's. Any pieces except the center pieces can have 9's. Partially specified goal positions cannot be fully verified; thus, even if no errors are discovered before the operation, some may be discovered once the missing pieces are replaced with actual ones. In this case the final position cannot be corrected, a message is displayed, and the current position remains unchanged.

YT is like YP, but is restricted by the pieces that have actual colors from the start. When all edge and corner pieces have 9's (like the position stored in memory 99 at the beginning of a run), YT is equivalent to YP. When none have 9's, YT generates a current position identical to the current goal position. Each time it is used (with 9's), YT generates a different position, where the pieces that have actual colors from the start are constant. Thus, YT is useful when you need a partially solved or partially scrambled position; for example, one face correct and the rest irrelevant, or some pieces forming a certain pattern and the rest irrelevant.

In the following script, YT is used to generate positions where the edge and corner pieces in the top layer are solved. It starts with the position in memory 99 (all 9's) and uses the operation "^" to assign to the top pieces the solved colors. It makes this the current goal position, displays it, and executes YT and J three times. The resulting display is shown following the script.

```
f99 ^16 ^26 ^36 ^46 ^136 ^236 ^146 ^246 e
jg"Goal" [3 yt j] x

Goal:
        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  9 2 9  9 3 9  9 1 9  9 4 9
  9 9 9  9 9 9  9 9 9  9 9 9

        9 9 9
        9 5 9
        9 9 9

Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  5 2 5  3 3 2  4 1 1  4 4 4
  5 5 5  3 1 5  4 5 4  5 3 3

        1 3 2
        2 5 1
        2 2 1

Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  3 2 5  4 3 2  5 1 4  2 4 5
  5 5 2  5 3 1  4 2 3  5 4 1

        4 1 5
        1 5 3
        3 1 2

Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  2 2 1  3 3 1  4 1 3  2 4 4
  5 5 5  2 5 3  1 5 4  2 5 4

        3 4 5
        3 5 2
        1 1 5
```

In the next script, YT is used to generate positions where the edge and corner pieces in the top layer, as well as the edge pieces in the middle layer, are solved. So the script starts with the solved position and uses "^" to assign 9's to the bottom locations; then it continues like the previous script.

```
^15=99 ^25=99 ^35=99 ^45=99 ^135=999 ^235=999 ^145=999 ^245=999 e
jg''Goal'' [3 yt j] x

Goal:
        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  2 2 2  3 3 3  1 1 1  4 4 4
  9 9 9  9 9 9  9 9 9  9 9 9

        9 9 9
        9 5 9
        9 9 9

Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  2 2 2  3 3 3  1 1 1  4 4 4
  3 3 5  1 5 4  2 5 1  5 4 2

        4 2 5
        5 5 1
        5 5 3

Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  2 2 2  3 3 3  1 1 1  4 4 4
  1 1 4  2 5 5  2 5 5  3 4 5

        5 3 3
        5 5 2
        4 5 1
```

```
Scramble template

        6 6 6
        6 6 6
        6 6 6

  2 2 2  3 3 3  1 1 1  4 4 4
  2 2 2  3 3 3  1 1 1  4 4 4
  5 4 4  5 5 5  3 5 5  2 1 2

        1 2 1
        5 5 3
        4 5 3
```

5.4 Position display

D: Display the current position as a list of color digits. (See section 3.2, "Positions".) Here is an example:

```
  2 2 3 3 1 5 4 3 6   2 4 3 1 2 6 4 4 1   6 1 5 2 3 5 5 3 1
  1 3 3 4 4 6 4 5 2   3 2 5 6 5 1 6 2 1   5 6 6 1 6 4 2 5 4
```

In a script (but not in interactive mode), you can optionally add a short title, to be displayed with the position. The title is a string of characters enclosed in double quotation marks (no spaces between D and the quotation mark). The string can have any length, but only the first 7 characters are displayed. Here is an example of a script and the resulting display:

```
d''Pos. 4''

Pos. 4:   2 2 3 3 1 5 4 3 6   2 4 3 1 2 6 4 4 1   6 1 5 2 3 5 5 3 1
          1 3 3 4 4 6 4 5 2   3 2 5 6 5 1 6 2 1   5 6 6 1 6 4 2 5 4
```

Note the difference between the operation D and the run option D: the option D instructs Digicube to display the initial position whenever S or its variants is executed, while the operation D simply displays the current position, at any time. Thus, D, if executed just before S, will display the same position as the option D.

J: Display the current position as rows of color digits. (See section 3.2, "Positions".) Apart from the different display format, J is like D, including the optional title in quotation marks. Here is how the position shown for D is displayed:

```
        5 6 6
        1 6 4
        2 5 4

2 4 3   6 1 5   2 2 3   1 3 3
1 2 6   2 3 5   3 1 5   4 4 6
4 4 1   5 3 1   4 3 6   4 5 2

        3 2 5
        6 5 1
        6 2 1
```

U: Display the current position as a list of pieces. (See section 3.3, "Pieces".) Apart from the different display format, U is like D, including the optional title in quotation marks. Here is how the position shown for D is displayed:

```
^135=415 ^136=254 ^145=641 ^146=316
^235=153 ^236=362 ^245=426 ^246=235
^13=35 ^14=54 ^15=31 ^16=24
^23=62 ^24=16 ^25=46 ^26=41
^35=32 ^36=15 ^45=52 ^46=36
^1=1 ^2=2 ^3=3 ^4=4 ^5=5 ^6=6
```

A: Display the current position as a 2-dimensional cube schematic. (See section 3.2, "Positions".) Apart from the different display format, A is like D, including the optional title in quotation marks. 9's in the position are displayed as "-". Here is how the position shown for D is displayed:

```
            +---+---+---+
            | 5 | 6 | 6 |
            +---+---+---+
            | 1 | 6 | 4 |
            +---+---+---+
            | 2 | 5 | 4 |
+---+---+---+---+---+---+---+---+---+---+---+---+
| 2 | 4 | 3 | 6 | 1 | 5 | 2 | 2 | 3 | 1 | 3 | 3 |
+---+---+---+---+---+---+---+---+---+---+---+---+
| 1 | 2 | 6 | 2 | 3 | 5 | 3 | 1 | 5 | 4 | 4 | 6 |
+---+---+---+---+---+---+---+---+---+---+---+---+
| 4 | 4 | 1 | 5 | 3 | 1 | 4 | 3 | 6 | 4 | 5 | 2 |
+---+---+---+---+---+---+---+---+---+---+---+---+
            | 3 | 2 | 5 |
            +---+---+---+
            | 6 | 5 | 1 |
            +---+---+---+
            | 6 | 2 | 1 |
            +---+---+---+
```

DG: Display the current Goal position as a list of color digits. DG is like D, but displays the current goal position.

JG: Display the current goal position as rows of color digits. JG is like J, but displays the current goal position.

UG: Display the current Goal position as a list of pieces. UG is like U, but displays the current goal position.

AG: Display the current Goal position as a cube schematic. AG is like A, but displays the current goal position.

5.5 Miscellaneous display

L: Display Label. L is followed by a string of characters enclosed in double quotation marks (no spaces between L and the quotation mark). When the script is executed, the string is displayed at that point (as a separate line). This lets you insert a short title or note among the lines displayed by the script, which is also useful if reading the output file later. The string can have any length, but only the first 20 characters are displayed. L with no string simply inserts a blank line at that point. Here is an example:

```
l''Test 3''
```

In interactive mode, the string needs no quotation marks, and all characters to the end of the line are added to the output file; so L can be used for any comments.

"*": Display comment. "*" is valid only in interactive mode, where it is identical to L: you can enter any text following it, and the text to the end of the line is added to the output file.

In a script, "*" is not an operation. It can be used only in the first position on a line, and marks the whole line as a comment. That line is ignored when running the script.

ON, **OF**: Switch On/Off the display of input moves and turns. The moves and turns specified in a script are normally executed without being displayed. This means that they are not recorded in the output file either. This operation lets you display them selectively, by switching the display on and off. The display is off at the beginning of a run (or if changing the current model). When ON is executed, if the display was off, the message "Disp on" is displayed; with OF, if the display was on, the message is "Disp off".

This feature is especially useful with random moves and turns (those you specify as 0), as it lets you see what values were actually generated and executed. In the following example, the display at start is assumed to be off; the first line is the script and the second one is the resulting display (note that the random move and turn are repeated 3 times):

```
of 3 -1 (5) on 2 1 [3 0 (0)] (5) -2 1 of 3 -1 5 x

Disp on 2 1 -3 (3) -1 (1) 5 (3) (5) -2 1 Disp off
```

In interactive mode, the moves and turns you enter are written to the output file as keyboard entries, even with the display off; when ON is in effect, they are simply repeated on the next line. Thus, ON is important only for random moves and turns, to show the actual values.

5.6 Counts

Introduction

The moves and turns you specify in a script or interactively are counted as they are executed, and the current counts can be displayed. Also counted is the number of cycles (iterations) of loops. The counts do not include any moves or turns generated by the program (in solutions, for example, and in operations like Y and R). Changing from script to interactive mode or vice versa (with the operation I) does not alter the counts.

The counts are 0 when a run starts, and you can clear them at any time. They are usually meaningful only for a specific set of operations, so you normally clear them before, and display them after, those operations. The counts are correct up to about 2.1 billion; then you must clear them.

Operations

Q: Display the current counts of moves, turns, and cycles. Each count is shown only if non-zero (unless all three are 0). In a script (but not in interactive mode), you can optionally add a short title, to be displayed with the count. The title is a string of characters enclosed in double quotation marks (no spaces between Q and the quotation mark). The string can have any length, but only the first 20 characters are displayed. Here is an example of a script and the resulting display:

```
q"Check point 1"
```

```
Check point 1: 23 moves, 1 turn
```

Q0: Clear the counts of moves, turns, and cycles ("0" is the digit 0). The message "Clear count" is displayed. The counts are automatically cleared at the beginning of a run (or if changing the current model).

QC: Display the current cycle. QC can be used within the brackets defining a loop, if you want to add the cycle number to whatever is displayed in each cycle. Outside the brackets of a loop, QC always displays "#0". In interactive mode, a loop contains only moves and turns, so you cannot use QC.

The following example illustrates the use of Q, Q0, and QC with a group of elements repeated 4 times. The first line is the script, and the rest are the resulting display.

```
d q0 [4 qc 1 3 -3 -1 (5) (5) (5) q d] x


  1 1 1 1 1 1 1 1 1   2 2 2 2 2 2 2 2 2   3 3 3 3 3 3 3 3 3
  4 4 4 4 4 4 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

Clear count
#1
4 moves, 3 turns, 1 cycle

  4 4 4 4 4 4 4 4 4   3 3 3 3 3 3 3 3 3   1 1 1 1 1 1 1 1 1
  2 2 2 2 2 2 2 2 2   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

#2
8 moves, 6 turns, 2 cycles

  2 2 2 2 2 2 2 2 2   1 1 1 1 1 1 1 1 1   4 4 4 4 4 4 4 4 4
  3 3 3 3 3 3 3 3 3   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

#3
12 moves, 9 turns, 3 cycles

  3 3 3 3 3 3 3 3 3   4 4 4 4 4 4 4 4 4   2 2 2 2 2 2 2 2 2
  1 1 1 1 1 1 1 1 1   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

#4
16 moves, 12 turns, 4 cycles

  1 1 1 1 1 1 1 1 1   2 2 2 2 2 2 2 2 2   3 3 3 3 3 3 3 3 3
  4 4 4 4 4 4 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6
```

5.7 Memories

Introduction

The 99 general-purpose positions, called memories, are used to save the current position or the current goal position, if you need them later in the current run, or to store positions to be compared with the changing current position. They are identified by the number 1 to 99, which is attached to the operation letter(s). In this manual, identifiers like "T1-T99" and "CC1-CC99" are sometimes abbreviated in their description as T, CC, etc.

At the beginning of a run (or when changing the current model), memories 1 to 96 contain the solved position; so whenever you need the solved position during a run, you can simply copy it from any one of these memories that has not been modified during that run. Memory 97 contains the corners goal position, and memory 98 the anchor goal position (see section 5.8, "Solutions"). Memory 99 has correct center pieces and 9's for all edge and corner pieces; it is thus a generic goal position, useful as a starting point when creating positions with many 9's. Apart from their initial contents, all memories are identical. Changing from script to interactive mode or vice versa (with the operation I) does not alter the 99 memories.

Operations

T1-**T99**: Copy the current position To memory 1-99.

F1-**F99**: Copy the current position From memory 1-99.

TG1-**TG99**: Copy the current Goal position To memory 1-99.

FG1-**FG99**: Copy the current Goal position From memory 1-99.

C1-**C99**: Compare the current position with memory 1-99. The two positions need not be valid, and may contain 9's. Thus, you can also use C to compare goal positions.

The comparison is successful when all 54 digits of one position are the same as the corresponding digits of the other. When successful, the message "Match mem" is displayed, showing the memory number; nothing is displayed if the comparison failed. Since the counts of moves, turns, and cycles discussed earlier are often needed in conjunction with comparisons, they are displayed following the message, if non-zero. Here is a short script using T1 and C1 (with any initial position), and the resulting display.

```
q0 t1 1 3 (5) (-5) -3 -1 c1 x

Clear count
Match mem 1: 4 moves, 2 turns
```

CC1-**CC99**: Compare Constantly the current position with memory 1-99. CC compares the current position with a memory, just like C, but the comparison is performed later. When CC is executed, Digicube only adds the memory number to a list (known as the CC list). Then, it performs the comparison automatically with every memory in the list, every time the current position changes (after every move or turn, and after the operations F, E, Y, YP, YT, R, all S and P variants, "^", "/", "\"), until you clear the list. The list is cleared with CC0, and can be displayed with CC. It can hold up to 99 memories.

The list is updated and used the same way in a script and interactively. Changing from script to interactive mode or vice versa (with the operation I) does not alter it.

When a comparison is successful, a message is displayed, as shown for C. If you modify a memory after adding it to the list, a comparison based on that number will no longer be the same. If several memories in the list contain the same position, a successful comparison ends after the first one (so there is only one message). In this case, if it is important that another memory be shown with the message, copy some other position to the first memory, or clear the list and re-create it without it.

In the following script, T10 stores the current position, then CC10 is used to confirm that the repeated moves cause the cube to return to this position every 84 cycles:

```
t10 cc10 q0 [300 2 3 5] x

Clear count
Match mem 10: 252 moves, 84 cycles
Match mem 10: 504 moves, 168 cycles
Match mem 10: 756 moves, 252 cycles
```

CC: Display the CC list. CC displays the list of memories defined with CC operations. This list is useful for documentation in the output file, or as reminder when using these operations interactively. The list is in the order in which the memories were added. Here is an example of the display:

```
CC: 1 3 11 12
```

CC0: Clear the CC list ("0" is the digit 0). Clearing the list ends the automatic comparisons. The message "Clear CC list" is displayed. The list is automatically cleared at the beginning of a run (or if changing the current model).

G1–G99: Compare the current position with memory 1–99 as Goal. G is like C, but the comparison treats the memory as a goal position, rather than a regular position. Thus, 9's in that position will match any value in the corresponding digits in the current position. (If there are no 9's, the G comparison is identical to C.) The message, when the comparison is successful, is "Match goal mem".

Recall that memories with 9's can be compared also using C. But that would be a regular comparison, so those 9's would require 9's in the corresponding digits in the current position for the comparison to be successful.

GG1–GG99: Compare constantly the current position with memory 1–99 as Goal. GG is like CC, but the comparisons treat the memory as a goal position, rather than a regular position (see the descriptions of G and CC earlier). The list defined with GG operations (known as the GG list) is separate from the CC list, and it too can hold up to 99 memories.

CC and GG definitions can coexist, and the automatic comparisons are performed for both lists. A particular comparison may then be successful for a CC memory, or a GG memory, or both; in the latter case, two messages are displayed.

GG: Display the GG list. GG is like CC, but displays the GG list.

GG0: Clear the GG list. GG0 is like CC0, but affects the GG list and comparisons. The message "Clear GG list" is displayed.

5.8 Solutions

Introduction

Digicube solves cube positions by relying on programming techniques alone, without any mathematical or empirical algorithms. (Only when verifying the correctness of a position does it use a mathematical concept – the principle of parity.) It simply tries move sequences of increasing length, where the sequences consist of combinations of 3 move types, 1, 3, 5 (*right*, *front*, *down*, clockwise and counterclockwise). A high degree of optimization, applied at all levels of the implementation, makes this idea feasible.

For example, not all possible combinations need to be tried. To save time, Digicube ignores sequences containing moves that do nothing (consecutive opposites like "1 –1" and "–1 1") or have the same effect as others (3 identical moves like "1 1 1", which is the same as "–1", and reverse pairs like "–1 –1", which is the same as "1 1"). This reduces from 6 (1, –1, 3, –3, 5, –5) to about 4.45 the average number of moves tried for each move added to the sequence length, and reduces therefore to 4.45 the factor by which the processing time increases with each extra move. For 12-move sequences, a full search is about 36 times faster.

Each sequence is applied to an initial position until one is found that leads to the required end position. Ideally, the initial position would be the one that must be solved, and the end position would be the solved position. But this is practical only for simple positions that can be solved in a few moves. Even solutions as short as 15 moves take a few minutes to discover in this fashion, and typical positions may need over 30 moves (because of the restriction to three faces), which would take millions of years.

To make this method practical, Digicube divides the solution into several steps, each one with its own initial and end positions. The move sequences needed in each step are then short enough to be discovered fairly quickly, resulting in an average time of 3.3 seconds to reach the final, solved position. (The times mentioned in this manual are for a PC with a 3.4 GHz Intel i5-7500 processor running Windows 10.)

There are several solution types, implemented through the operation S and its variants, SA, SC, SE, SD, and SG. S uses the standard solution method, which requires several steps to reach the solved position; this method can solve any position. SA, SC, and SE also use the standard method, but they stop after the first step (anchor), or the second step (corners), or the other steps (edges), before reaching the solved position. They are useful in experiments, or if you wish to combine methods.

SD attempts to reach the solved position in one step; this is practical only if the solution involves a relatively small number of moves. SG attempts to reach in one step a position that matches the current goal position; that is, any color digit is accepted if the corresponding digit in the goal position is 9. This can usually be achieved in a relatively small number of moves if more than about 6 pieces in the goal position have 9's. Goal solutions are useful when the end position is a partial solution (as is the case in the individual steps of the standard solution), or when it is a pattern of colors that involves only some of the pieces.

All S variants use the current position as the initial position; so they first verify it, and end with an error message if it is invalid. If this is the position of a valid real cube, an error usually means that you made a mistake when entering it. SG also verifies the current goal position. Both positions must be in the standard orientation.

Although Digicube doesn't seek solutions with the smallest number of moves, it must be noted that the one-step solutions (including the solutions for models M2 and M3), as well as the individual steps in the standard solution, do discover, in fact, the shortest sequence (allowing for the restriction to three faces). This is due simply to the search method: it starts with one move, then tries two, three, and so on, and stops at the first successful sequence. There are usually additional sequences of the same length that lead to the end position. Which sequence is discovered first depends on the order in which move types are tried. The order can be modified with the operation M (the normal order is 1 3 5).

To solve a real cube, you first enter its position (with the operation P or its variants, P1–P6); that becomes the current position, and you can execute one of the S operations. For SG, you must also specify the current goal position (with PG or its variants, PG1–PG6). The solution process will display the sequence of moves needed to reach the solved position, and you apply these moves to the real cube, reading the displayed lines directly (or, if you prefer, reading them later from the output file, where they were added at the same time). With the standard method there is a move sequence for each step, and you apply them in the order shown.

The initial position is entered with the cube in the standard orientation (see section 3.2, "Positions", and the operation P), and you must hold it in this orientation for each move, throughout the solution process. (Watch the center pieces: color 3 *front*, color 6 *up*.) It is a good idea to check the cube visually after each step, to confirm that the new position is what that step was meant to accomplish. If it is not, it means that you made a mistake when applying the moves, and you should stop, since the following moves would be useless. You must enter then the current position and treat it as the initial position for a new solution.

Since only three faces are involved, and these faces (*right*, *front*, *down*) are easily accessible, you should be able, with practice, to apply the moves quickly and without even watching the cube.

You can stop a solution before its normal end by pressing X (both in interactive mode and with a script); if you do, the current position remains unchanged.

If the initial position is the solved position, or if a particular step is already solved, "---" is displayed instead of a move sequence. Here are two examples:

```
Solution: ---
Moves: 0    Time: 0.0 sec    Sequences tried: 1

1. Anchor: ---
2. Corners: ..... ..... .. 1 3 3 -1 3 -1 5 3 -5 1 -3 -1
Moves: 12    Time: 1.3 sec    Sequences tried: 28,110,832
```

The standard solution

Here is a typical standard solution, showing also the initial position:

```
Initial:  1 3 5 4 1 1 6 3 1   3 6 6 2 2 6 2 4 2   4 5 4 2 3 1 5 5 3
          1 3 5 6 4 4 6 5 3   4 1 1 6 5 1 6 4 4   2 2 3 3 6 5 2 2 5
1. Anchor: ..... .. 5 3 1 3 6 -3 6
2. Corners: ..... .... 1 3 -1 -3 -1 3 -5 -1 -5
3. Edges: ..... ..... -1 -3 5 1 3 5 1 -3 -5 -3  >> 13 36
4. Edges: ..... ..... -5 -1 -5 1 5 1 5 1 -5 -1  >> 16 25
5. Edges: ..... ..... 1 3 1 5 3 -5 -1 -3 -1 -5  >> (14 23)
6. Edges: ..... ..... .. 3 -5 3 5 3 5 3 -5 -3 -5 3 3  >> (15 45)
7. Flip 14 15 23 45:  1 5 5 -1 -3 5 -3 -1 -5 1 1 5 3 -1 3 5
Moves: 74    Time: 3.0 sec    Sequences tried: 59,119,446
```

The lines are displayed one at a time, as the solution progresses. In each step, the number of dots shown at a given moment is the length of the move sequence currently being tried; this starts with one dot, more are added as the sequence length increases, and ends when a successful sequence is discovered and displayed. (For the first few dots this occurs so fast that they all seem to be displayed at once.) In the edge steps, the program skips sequences with an odd number of moves (to save time, since a successful sequence always has an even length). Once the sequence for a step is displayed, the dots simply show the number of moves in that sequence.

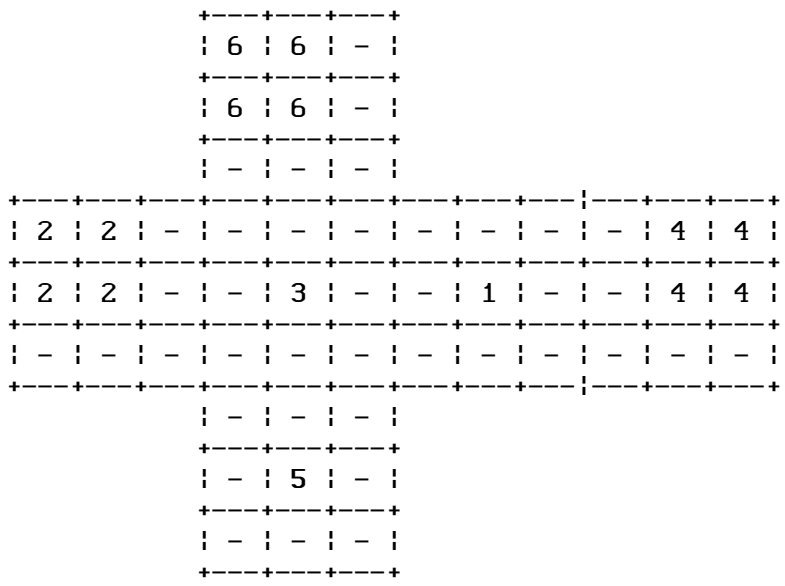
In the following discussion, refer to the solution displayed above. This discussion is important only if you are interested in the logic behind the solution steps; you don't have to read it if all you want is to use the S operations.

The first step solves the set of 4 pieces known as *anchor*: corner piece 246 (*left-back-up*) and the adjacent edge pieces, 24 (*left-back*), 26 (*left-up*), 46 (*back-up*). These piece numbers are their colors; but, since the cube is in the standard orientation, these numbers are also the locations of these pieces when solved (see section 3.3, "Pieces").

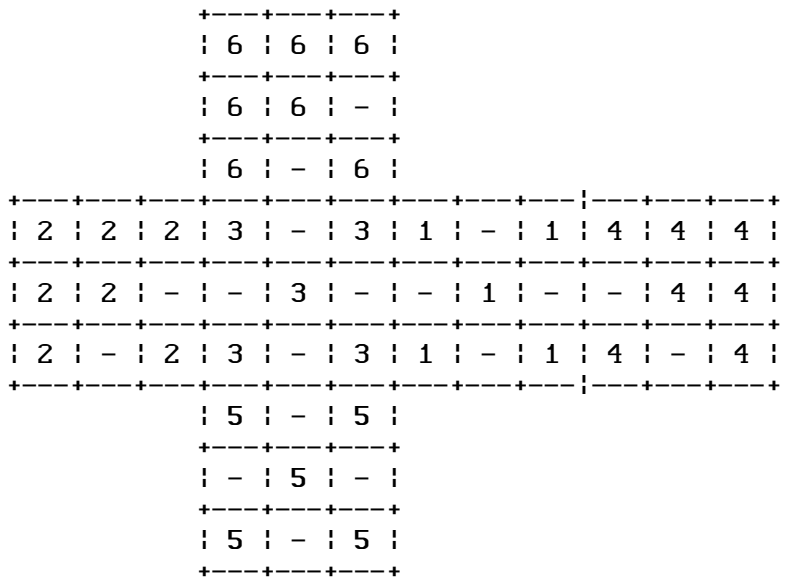
This is the only step where all 6 faces are involved. Once the anchor pieces are in their solved locations, we only need moves involving 3 faces (1, 3, 5, which are at sides 1, 3, 5 – *right*, *front*, *down*), since these moves can access all remaining pieces. Longer sequences are needed in each step when restricted to 3 faces, but this restriction also makes them easier to apply on a real cube. Note that these faces were chosen specifically because they are the easiest to access. Moreover, face 1 is somewhat easier than face 3, and face 3 is easier than face 5; so the order in which moves are tried is 1 3 5 so as to generate, on average, more moves 1/–1 than 3/–3, and more 3/–3 than 5/–5 (see operation M).

The anchor sequence itself demonstrates this principle: because it uses 6 faces, it is shorter than the others; but it is a little more difficult to apply on a real cube.

The anchor step is, in effect, a solution of type SG where the goal position has 9's for all pieces except the center pieces and the 4 anchor pieces. This is known as the anchor goal position, and is what the current goal position and memory 98 are initialized with at the beginning of a run. Here is what it looks like as a 2-dimensional cube schematic (the 9's are shown as dashes in this display format):



The second step solves the remaining 7 corner pieces. It is, in effect, a solution of type SG where the goal position has correct center, anchor, and corner pieces, and 9's for the 9 edge pieces still unsolved: 13, 14, 15, 16, 23, 25, 35, 36, 45. This is known as the corners goal position, and is what memory 97 is initialized with at the beginning of a run. Here is what it looks like as a cube schematic:



The next steps solve the remaining 9 edge pieces. They are solved in pairs, so up to four steps are required. (The 9th piece is solved automatically during this process, because a valid cube cannot have a single incorrect piece). It would take too long to solve specific pieces with an SG solution, as was done in the previous steps (because there are fewer and fewer 9's in the goal position as we approach the end). Instead, a modified version of the SG solution is used, which takes advantage of the fact that it doesn't matter in what order the remaining edge pieces are solved, or how they are combined into pairs. Thus, instead of searching for a sequence that solves specific combinations of two pieces, a sequence is deemed successful if it solves *any* two pieces. Such sequences are more common, so they are discovered sooner (and consequently they are also shorter).

After solving two pairs, even this kind of search takes too long. So, for the last two pairs, a sequence is deemed successful also if the two edge pieces in the pair are in their correct locations but flipped. A solved pair may still be found, but this is not a requirement. In the end, either all four pairs are solved, or three are solved and one is flipped, or two are solved and two are flipped.

The reason flipped edge pieces are accepted is that they are easy to correct: there exist sequences of moves that flip (and hence correct) 2 or 4 edge pieces at once. There are 36 possible combinations of 2 pieces out of the original 9, and 126 combinations of 4 pieces. The 162 different sequences needed to flip them are stored in the program, and the appropriate one is used directly as the last step, *Flip*. Since this step involves no search, it takes no significant time. In the rare cases where all four pairs are solved and there are no flipped pieces, this step is omitted.

Each edge step displays, following the move sequence, the pair or pairs that were found; a flipped pair is enclosed in parentheses. Then, the flip step shows the two or four pieces that must be corrected. These are the pieces previously shown in parentheses, now sorted numerically.

At the end, Digicube displays the total number of moves required for the solution, the time it took to discover it, and the number of move sequences tried. The time shown is the elapsed time, so it is a true measure of the solution time only if no other programs were running in the same core, and if there were no interruptions. The sequence total includes every sequence tried, in every step, from one move in length to the final number of moves.

The average total moves is 73.6, about one in ten solutions needs more than 80, and the highest is 90 (as seen in over 200,000 solutions of random positions). The average time is 3.3 seconds (see operation K).

While four edge steps with one pair per step is the most common solution for random positions, other situations can occur. For example, two and even three pairs may be discovered in one step, solved or flipped; or one, two, and even three solved or flipped pairs may exist even before the first edge step, so they are not shown in any step. Very rarely, a pair is included in the flip step without being shown earlier in parentheses; this is a flipped pair found together with two solved pairs before the first edge step. Thus, rather than four edge steps, a solution may have only three, two, or even one. Here are various combinations of edge and flip steps seen in the solution of random positions:

```
3. Edges: ..... ..... 1 3 5 3 -1 -5 -3 -1 -5 3  >> 16 36
4. Edges: ..... ..... .. 1 1 5 1 3 1 -5 -3 -1 -5 -3 -1  >> 13 23
5. Edges: ..... ..... .. 1 -3 -5 1 3 -1 -3 5 3 -5 -1 5  >> 15 45
6. Edges: ..... ..... -1 -5 -1 -5 1 5 1 5 1 -5  >> (14 35)
7. Flip 14 35:  -1 -3 -1 3 1 5 1 3 1 1 -3 -5 1 3 1 -3

3. Edges: ..... ..... 1 3 5 -3 -1 -3 -5 -1 5 3  >> 23 45
4. Edges: ..... ..... -1 -3 -5 -3 -1 3 5 3 1 5  >> 15 25
5. Edges: ..... ..... .. 1 1 -5 -1 -3 -5 3 3 5 1 3 5  >> 13 36
6. Edges: ..... ..... -1 3 1 3 1 3 -1 -3 -1 -3  >> 14 16

3. Edges: ..... ..... 1 3 1 3 1 -3 -1 -3 -1 -3  >> 13 16
4. Edges: ..... ..... 3 5 -3 -5 -3 -5 -3 5 3 5  >> 25 45, (14 15)
5. Edges: ..... ..... .. 5 -1 -3 -1 -3 1 3 1 3 1 -3 -5  >> (35 36)
6. Flip 14 15 35 36:  1 1 5 3 -1 3 5 1 5 5 -1 -3 5 -3 -1 -5

3. Edges: ..... ..... 1 5 1 5 -1 -5 -1 -5 -1 5  >> 35 45
4. Edges: ..... ..... 3 -5 -3 -5 -3 -5 3 5 3 5  >> 13 25
```

```
3. Edges: ..... ..... 1 3 1 5 3 -5 -1 -3 -1 -5  >> 13 15, 35 45
4. Edges: ..... ..... 1 3 -5 -1 -5 -3 -5 1 3 5  >> (14 23), (25 36)
5. Flip 14 23 25 36:  1 3 1 3 5 -1 5 3 1 1 -3 -1 -5 3 -5 -1 3 -1

3. Edges: ..... ..... 1 3 1 3 -1 -3 -1 -3 -1 3  >> 13 23
4. Edges: ..... ..... .. 1 1 3 5 3 1 5 -1 -3 -5 -3 1  >> 14 15, (16 25), (35 36)
5. Flip 16 25 35 36:  -1 3 -1 3 3 1 3 5 -1 5 3 1 1 -3 -1 -5 3 -5 -3 1

3. Edges: ..... ..... 1 3 5 3 1 -5 -1 -3 -5 -3  >> 13 45
4. Edges: ..... ..... .. 1 -5 -1 -3 -5 3 3 5 1 3 5 1  >> 14 15, (25 36)
5. Flip 25 36:  3 5 -1 -3 5 5 3 5 1 5 3 -5 -3 -5 -3 5

3. Edges: ..... ..... -3 -1 5 3 5 1 5 -3 -1 -5  >> 23 36, (14 15), (16 35)
4. Flip 14 15 16 35:  1 1 -5 -3 -1 3 3 1 5 -3 5 1 3 1 1 -3 -5 -1

3. Edges: ..... ..... 1 3 5 -1 -3 -1 -5 -1 3 5  >> 16 23, 25 36, (13 14), (15 35)
4. Flip 13 14 15 35:  1 -5 -3 -1 3 3 1 5 -3 5 1 3 1 1 -3 -5

3. Edges: ..... ..... .. -1 -3 -1 -3 1 1 3 1 3 1 3 3  >> (15 36)
4. Flip 14 15 23 36:  3 3 1 -3 -5 1 1 5 1 3 -5 3 1 5 5 -1 -5 3
```

The 162 sequences stored in the program and used in the flip step were discovered with the Digicube feature known as back sequences (see chapter 10, "Back sequences"). The sequences have 16, 18, or 20 moves. From the many sequences discovered for each combination of edge pieces, those stored in the program were selected because they are shorter, and also, where possible, because they have more 1's and fewer 5's (as explained in the anchor step).

Additional details pertaining to the standard solution can be found in the description of the operations M and K.

Operations

S: Solve the current position. S uses the standard solution method, and replaces the current position with the solved position. The standard solution was discussed earlier in this section.

SA: Solve the current position for Anchor. SA is like S, but stops after the first step, when only the anchor is solved. This is useful in experiments, or if you wish to combine solution methods, or before SD and SG (which require a solved anchor).

SC: Solve the current position for Corners. SC is like S, but stops after the second step, when the anchor and the corner pieces are solved. This is useful in experiments, or if you wish to combine solution methods.

SE: Solve the current position for Edges. SE is like S, but stops after the edge steps, when the cube is solved except for the flip step if there are flipped edge pieces. This is useful in experiments.

SD: Solve the current position Directly. SD attempts to reach the solved position in one step. This is practical only if the solution requires a relatively small number of moves (see operation K). The current position must have the anchor solved (otherwise, because of the restriction to three move types, no solution would be found); if necessary, use SA before SD. The step title is "Solution".

SG: Solve the current position for Goal. SG attempts to reach in one step a position that matches the current goal position; that is, any color digit is accepted if the corresponding digit in the goal position is 9. This can usually be achieved in a relatively small number of moves if more than about 6 pieces in the goal position have 9's (see operation K). The current position must have the anchor solved (otherwise, because of the restriction to three move types, no solution would be found); if necessary, use SA before SG. The step title is "Goal solution".

The current goal position too must have the anchor solved (otherwise the current position would never match it, and no solution would be found). You cannot use SA to solve the anchor (because goal positions normally have 9's). If you don't want to specify the whole position, start with one that has a solved anchor and modify it as needed. To help you, the current goal position, as well as memory 98, have the anchor goal position at the beginning of a run.

When the SG move sequence takes too long to discover, there is an alternative to reaching that goal position: the method that lets you reach any position from any other position by using the solved position as intermediary (see run option R). First you solve the current position; then you use YT to generate any actual position from the current goal position, enable the R run option, and solve that position; finally, you apply the reverse moves listed by this solution to the solved position, resulting in the position generated by YT.

SG can be useful if you are seeking partial solutions that involve only three faces. You start with SA, and then design a few successive goal positions that are attainable with SG.

M: Choose order of trial Moves in solutions. Digicube discovers solutions by trying all possible combinations of move types 1, 3, and 5, in sequences of increasing length. For example, when trying sequences of ten moves, it tries the three move types in each one of the ten positions. (Actually, sequences containing moves that do nothing or have the same effect as others are ignored, as explained earlier.) This is done for the single sequence of the SD and SG solutions, and for each step in the S, SA, SC, and SE solutions.

M has six variants: M135, M153, M315, M351, M513, M531. The numbers indicate, through the order of their digits, the order in which the three types of moves are tried. Thus, for M135, the order is 1 –1 3 –3 5 –5; for M153 it is 1 –1 5 –5 3 –3; and so on. At the beginning of a run (or when changing the current model to M1), the default order, 135, is in effect. The order can be set at any time, and the new choice will be in effect until changed.

The order chosen affects the individual move sequences discovered in each solution step because, if one move is tried before another, and if a sequence of a given length that solves that step exists for both, the sequence with the first move will be discovered first. The order may also affect the total time and the total number of moves required to solve a particular position, because, if a different sequence is discovered for an early step, the following steps will likely be different. However, there is no way to predict which order is better for a given initial position (that is, which order results in shorter solution time or fewer moves); so on average, for a large number of initial positions, there is no difference between the six M variants.

The only significant difference is in the relative number of moves of type 1, 3, and 5. With M135, because 1 is tried before 3, and 3 before 5, there are about 10% more moves 1/–1 than 3/–3, and 10% more 3/–3 than 5/–5. With the other M variants this changes accordingly. The default order is M135

because face 1 is somewhat easier to rotate than face 3 with a real cube, and face 3 is easier than face 5. In the last step, *Flip*, the move sequences are predetermined, so the current M variant is irrelevant; these sequences observe the order 135.

K1–K20: Stop after trying sequences of 1–20 moves in solutions. K limits the length of move sequences tried at each step, in order to stop long solutions. You can also stop a solution yourself, at any time, by pressing X. When a solution is stopped by the K limit, the message "** Stopped" is displayed. At the beginning of a run (or when changing the current model to M1), the default K15 is in effect. The limit can be set at any time, and the new choice will be in effect until changed. Specifying K without a number restores the default limit; thus, "K" is the same as "K15".

For SD and SG, which have only one step, when the limit is exceeded the solution process is terminated, and the current position remains unchanged. If you find the default limit too high, reduce it with K14 or K13. Conversely, if you think that a particular position could be solved with a longer move sequence and are willing to accept the longer time, increase the limit with K16. Each extra move increases the time needed to try all move combinations by a factor of about 4.45. Thus, the time it takes to exceed the limit is, approximately, 20 seconds with K13, 90 seconds with K14, and 400 seconds with K15. A solution, however, if one exists for that sequence length, will likely be found before trying all move combinations.

For S, SA, SC, and SE, which use the standard solution method, the limit may be exceeded in any step. The initial position is then modified by the program with a few random moves, and the solution process is restarted. If the K limit is low, this may have to be repeated (the message "** Stopped" shows the number of repetitions). The sequence of random moves is shown with the title "Modify position" and becomes the first step in the successful solution, before the regular anchor step (any prior, unsuccessful sequences are irrelevant). The total moves displayed at the end includes the sequence of random moves (since it is, in effect, part of the solution). The time and the number of sequences, however, are shown separately for each trial.

With the default K15, the limit is never exceeded, so the solution process is never restarted (this was checked with over 200,000 solutions of random positions); the average solution time is 3.3 seconds; about one in 500 solutions takes more than 20 seconds, and the longest seen is less than 50 seconds. In each step, each extra move increases the time needed to try all move combinations by a factor of about 4.45; in the anchor step, the factor is about 8.9 (because twice as many move types are tried).

K14 tends to reduce the longest times by restarting the solution, but the average time is the same as with K15. So normally there is no reason to reduce the limit so as to stop and restart long solutions. With K14 the solution is restarted for about one in 100 random positions (always in the anchor step, see below), and with K13 and lower, much more often. If you set K so low that the solution process is restarted more than 9 times, it is abandoned and the current position remains unchanged.

The anchor step uses all six move types; consequently, sequences are shorter, but a sequence of a given length takes longer to try than in the other steps. To compensate, when the limit for the other steps is more than 7, for the anchor step it is reduced as follows: 7 for K8–K11, 8 for K12–K14, 9 for K15–K20. With these values, it takes about the same time to reach a certain limit in all steps. Thus, the default limit K15 is needed only to allow 9 moves in the anchor step (about one in 100 random positions); the corners and edge steps never exceed 14 moves.

To illustrate the restarting process, here is a solution using K14 that was stopped twice (when the anchor step exceeded 8 moves – with K15 it would have ended successfully in 9 moves):

```
Initial:  3 6 3 6 1 4 6 6 4   4 6 6 5 2 5 1 3 1   3 3 5 2 3 1 4 1 2
          5 2 6 5 4 1 2 3 4   5 4 3 1 5 3 6 5 5   2 4 1 4 6 2 1 2 2
1. Anchor: ..... ...
** Stopped 1    Time: 6.9 sec    Sequences tried: 184,235,089

1. Modify position:  -4 -2
2. Anchor: ..... ...
** Stopped 2    Time: 6.9 sec    Sequences tried: 184,235,089

1. Modify position:  5 4 -5 -5 -3 -5 -4 3
2. Anchor: ..... .. -1 -6 -3 -4 5 -2 6
3. Corners: ..... ..... . 3 -5 3 5 3 1 1 5 5 -1 3
4. Edges: ..... ..... 1 5 3 -5 -1 -3 -1 -5 1 3  >> 23 45
5. Edges: ..... ..... .. 1 1 -3 -1 -5 -1 3 3 1 3 5 3  >> 15 16
6. Edges: ..... ..... 5 -1 -5 -1 -5 -1 5 1 5 1  >> (25 36)
7. Edges: ..... ..... .. -1 3 1 3 1 3 3 -1 -3 -1 -3 -1  >> (13 14)
8. Flip 13 14 25 36:  3 3 5 1 -3 1 5 3 3 -5 -3 -1 5 -1 -3 5 5 -3
Moves: 88    Time: 2.4 sec    Sequences tried: 47,968,279
```

When a solution is restarted, if it is important that exactly the same solution be discovered for a certain position on different occasions in the same run, use the NS and NR operations to save and restore the program's random state. Otherwise, since the moves that modify the position are random, the solution will be different on each occasion.

5.9 Random moves and turns

Introduction

If you specify the digit 0 for a move or a turn, Digicube generates and executes a random value. Random moves and turns are useful in experiments and for scrambling the current position. The operation ON lets you display the values actually generated, when this is important.

A series of consecutive 0's can be specified as a sequence or as a loop: "0 0 0 0 0" has the same effect as "[5 0]". With both methods, Digicube keeps track of the generated values and avoids combinations of moves or turns that would be useless or would produce the same result as a shorter combination; in other words, combinations that a careful user would not apply to a real cube. Thus, there will never be consecutive opposites like "1 -1" or "-1 1" (which do nothing); or 3 or more consecutive identical values like "1 1 1" (which is the same as "-1"), "1 1 1 1" (which does nothing), etc. There is no risk, therefore, that a random sequence of a certain length will effectively be shorter than specified.

Note, however, that this feature may not work if mixing random values with specific ones, as in "0 0 1 0 3 1 0 0", because Digicube only keeps track of the values it generates; so a sequence like "1 3 0" could actually result in "1 3 -3", and hence just "1".

You can design your own sequence of random moves and turns as a substitute for the operation Y (which generates a scrambled position). Y is equivalent to "N6 [200 (0) 0] R" (the repeat count,

however, is a random number in the range 190–210); it has both turns and moves, uses all 6 faces for the moves, and resets the final position. So if you need a different form of scrambling, use random moves directly. For example, if all you want is a different number of repetitions, simply use the sequence shown above with a different repeat count; or, if you want to scramble the current position without disturbing the anchor, use "R N3 [200 0]".

Operations

N3, **N6**: Use 3 or 6 faces in random moves. Normally only 3 faces (the faces at sides 1, 3, 5) are used for random moves, in order to match the moves used in solutions, and also to preserve the anchor (when the cube is in the standard orientation). But you can instruct Digicube to generate random moves that use all 6 faces, if you need this in experiments. N3 restricts moves to 3 faces, and N6 allows 6 faces. At the beginning of a run (or when changing the current model), N3 is in effect. You can execute N3 or N6 at any time, and the new choice will be in effect until changed. If you need N6 in only one place, remember to execute N3 immediately after that, to prevent the use of 6 faces in other places.

Random turns are restricted to 3 faces even when N6 is in effect. Turns rotate the whole cube, so a face only designates the turning axis. Thus, with 3 axes, a turn for face 2, 4, or 6 is identical to the reverse of a turn for face 1, 3, or 5, respectively (2 is the same as -1, -4 is the same as 3, etc.). Those turns, therefore, are being generated anyway, even with N3.

The choice of N3 or N6 does not affect the generation of scrambled positions by Y. Y uses 6 faces for its random moves even if N3 is in effect.

NS, **NR**: Save/Restore the program's random state. These operations allow you to repeat a series of previously generated random moves and turns. This is useful in experiments, when you want to perform different operations with the same series of moves or turns. NS saves the current state of the random number generator, and NR restores it. So you execute NS before the series is first generated, and NR later, whenever you want the series repeated. The stored state remains unchanged until NS is executed again, and can be restored any number of times. If executing NR without first executing NS in that run, the state restored is the one at the beginning of the run.

The following script illustrates these operations. A series of random moves and turns is generated twice and displayed, and we can confirm visually that they are identical. The script also shows how to do this for many moves and turns, and without displaying them: we use them to modify the same position (saved and restored with T1/F1), and we use T2/C2 to confirm that the final positions are identical.

```
on y t1 ns l"#1" (0) [8 0] (0) t2  f1 nr l"#2" (0) [8 0] (0) c2 of x

Disp on
Scramble
#1
 (3) -1 5 3 1 5 -3 -3 -5 (5)
#2
 (3) -1 5 3 1 5 -3 -3 -5 (5)
Match mem 2: 16 moves, 4 turns, 16 cycles
Disp off
```

If you use N3 and N6 to select 3 or 6 faces for the random moves, make sure the same choice is in effect after NS and after NR.

NS and NR do not affect the generation of random positions by Y, YP, and YT; these positions cannot be replicated by restoring a previous random state. If you need a random position identical to the one generated on a previous occasion, use a sequence like "N6 [200 (0) 0] R", which is almost as random as Y and can be controlled. The following script illustrates this, using T1/C1 to confirm that the generated positions are identical (memory 10 can hold any position):

```
n6 ns f10 [200 (0) 0] r t1  nr f10 [200 (0) 0] r c1 n3 x

Reset position
Reset position
Match mem 1: 400 moves, 400 turns, 400 cycles
```

5.10 Individual pieces

Introduction

The operations "^", "/", and "\" modify one piece in the current position. The piece is identified by its location (see section 3.3, "Pieces"); the location is a fixed number that does not depend on the cube's orientation or whether it is solved or scrambled. Locations are one-digit numbers for the 6 center pieces, two-digit numbers for the 12 edge pieces, and three-digit numbers for the 8 corner pieces. The digits represent the sides, 1 to 6. The colors of a piece are also one-, two-, and three-digit numbers, which depend on the piece: their digits represent the colors of the piece and, through their arrangement, its orientation.

These two numbers determine, therefore, the location, colors, and orientation of every piece, for every possible position (including invalid positions). The expression for each piece is **^loc=col** (location and colors). This, we saw, is how U displays a position, as in this example of a scrambled cube:

```
^135=316 ^136=451 ^145=523 ^146=135
^235=254 ^236=236 ^245=642 ^246=164
^13=63 ^14=64 ^15=51 ^16=23
^23=13 ^24=45 ^25=25 ^26=14
^35=26 ^36=16 ^45=35 ^46=42
^1=1 ^2=2 ^3=3 ^4=4 ^5=5 ^6=6
```

With "/" and "\" you specify the location, and with "^" both the location and the colors. Digicube verifies these numbers and, if invalid, displays an error message describing the problem (see chapter 12, "Error messages"). But it only checks the validity of the one piece being modified, not the entire position. A position that was valid before the operation may become invalid even with a valid piece (for example, if the piece is a duplicate or is incorrectly oriented); conversely, an invalid position can be corrected with an appropriate specification for one piece. A complete check of the current position is performed by V and S (and its variants).

Operations

"^": Specify the colors of one piece in the current position. "^" (caret) modifies the colors of the piece currently residing at the location specified. The operation uses the standard expression **^loc=col** (location and colors), with no spaces between the numbers and the symbols "^" and "=".

The color digits correspond to the location digits, thus determining both the colors and the orientation of the piece (see section 3.3, "Pieces"). For example, ^16=23 means that, for the piece in location 16, color 2 will be at side 1, and color 3 at side 6; ^245=642 means that, for the piece in location 245, color 6 will be at side 2, color 4 at side 4, and color 2 at side 5. You can also set the color of a center piece (for example, ^3=4). Note that for a corner piece, only 3 of the 6 possible combinations of colors are valid in a particular location.

A series of "^" can be used to specify part of a position, when only some of the pieces must change. Even an entire position can be specified with "^". For example, since the display operation U uses the same format, **^loc=col**, you can re-create a position displayed by U in a previous run by copying those lines from the output file into a script in the input file.

"^" can also be used to swap pieces, by specifying two operations. For example, to swap the edge pieces in locations 15 and 16 in the solved position, while properly orienting them, use "^15=61 ^16=51"; for the corner pieces in locations 135 and 235, use "^135=325 ^235=315". Swapping pieces in this manner is the equivalent of physically removing and replacing pieces in a real cube, which is likely to invalidate the cube. Thus, with "^", you must perform an even number of swaps, and also ensure proper orientation of the pieces in their new location, if you want the new position to be valid. You can use V to check its validity.

"^" can be used only for the current position, but you can specify 9's as color digits. This is useful if you are using the current position to modify a goal position. If a piece has 9's, all its faces must be 9.

When the color number is the same as the location number (the piece is as it is in the solved position), you can abbreviate the operation **^loc=col** as **^loc**. For example, ^2 is the same as ^2=2, ^35 is the same as ^35=35, ^246 is the same as ^246=246.

"/", "\": Flip clockwise (cw) or counterclockwise (ccw) one piece in the current position. "/" (slash) and "\" (back slash) modify the orientation of the piece currently residing at the location specified. The location number follows the symbol, with no intervening spaces (/23, /135, \46, \246, etc.).

An edge piece can have two orientations, so from its current orientation it can be flipped in only one way: "/" and "\" have the same effect, and two of them restore the original orientation. A corner piece in a particular location can have three orientations, so from its current orientation it can be flipped in two ways: cw with "/", ccw with "\". The direction, cw or ccw, is judged by looking straight at that corner. Two consecutive "/" is the same as one "\", and vice versa; three of the same type return the piece to its original orientation. "/" and "\" are invalid with center pieces.

5.11 Miscellaneous operations

"+", "–": Set/reset the current run options. (See chapter 4, "Run options".) The option letter follows the symbol, with no intervening spaces (+D, –R, etc.). "+" and "–" allow you to specify run options during a run. This is useful when it is more convenient than before the run, or when you need to set and reset an option several times during a run.

Before a run, specifying an option enables it, or assigns to it a certain value; not specifying it leaves it disabled or with a default value. During the run, "+" is equivalent to specifying the option before the run, and "–" restores the state or value it has when not specified before the run. "+" for an option currently enabled, or "–" for an option currently disabled, or D1 when D1 is in effect, etc., is ignored. (M is different, as explained later.) The following options can be set and reset: M, D, J, U, A, R, W, K. The options S, I, O, and N, which determine values needed before the beginning of a run, cannot be set and reset. Thus, only the following formats are valid for "+" and "–":

+M1 +M2 +M3 +D +D1 +D2 +J +U +A +R +W +K
–M –D –J –U –A –R –W –K

Note that –M must have no number, since it implies restoring the default value, M1; thus, –M is the same as +M1. Similarly, –D needs no number, since D, D1, and D2 are mutually exclusive; thus, –D disables any current D. The others (J, U, A, R, W, K) are enabled by "+" and disabled by "–".

For M, with either "+" or "–", and whether changing the current model or not, the work environment is initialized, similarly to starting a new run, as listed below. (The values listed are for model M1; for the other models, see chapters 8, "Model M2" and 9, "Model M3".) The message "Initialize M" is displayed, showing the new model.

For the operations ON/OF, N3/N6, M, and K, the default choices take effect: OF, N3, M135, and K15. The CC and GG lists, and the move, turn, and cycle counts are cleared. The current position is set as the solved position, and the current goal position as the anchor goal position. Memories 1 to 96 are set as the solved position, memory 97 as the corners goal position, and memory 98 as the anchor goal position. Memory 99 is set with correct center pieces and 9's for all edge and corner pieces.

Thus, if you want to initialize the work environment without starting a new run (in a script, for example), simply execute +M specifying the current model.

The state of the run options (D, J, U, A, R, W, K) is not affected by this initialization. The program's random state is not affected either: if you saved it with NS, you can restore it with NR, as usual, after changing the current model.

In addition to allowing you to specify an option during, rather than before, a run, "+" and "–" allow you to use run options selectively, similarly to operations, by setting and resetting them as needed. For example, you can set D1 or D2 for a particular solution, D for another, and no display at all for the rest; or you can set K for an operation that will generate many lines if you don't want them added to the output file, and then reset it.

I: Begin or end Interactive mode. (See chapter 6, "Interactive mode".) When encountered in a script, I causes Digicube to interrupt the script and begin interactive mode. When executed interactively, I causes it to end interactive mode, return to the script, and continue from the point following the I there. The message "Begin interactive" is displayed when entering interactive mode, and "End interactive" when exiting it.

Digicube can begin interactive mode also from the start, if you specify the run option I. The run is then an interactive run, and when you execute I to end interactive mode, the run too ends.

Z: End script in custom operation. (See chapter 11, "Custom operations".) Some custom operations execute a script in the input file, and then continue their special function. For example, through a script you can specify an initial position for such an operation. Z is needed in the script to tell Digicube when to stop executing the script and return to the custom operation. You cannot use X to stop executing the script, as this would end the run. Z is ignored if that script is executed directly rather than through a custom operation.

X: End current run. When encountered in a script, X causes Digicube to stop executing the script and end the run. You must always end a script with X, else Digicube will read the input file past the script and attempt to execute whatever elements happen to be there. X is optional if that script is the last thing in the file, but it is good practice to have an X anyway, in case you add some lines there later or move the script to another place in the file.

If executed interactively, X ends interactive mode and then ends the run, without returning to the script from where interactive mode was initiated. In an interactive run (started with the run option I), both I and X end the run.

CHAPTER 6

Interactive mode

The simplest way to use Digicube is interactively. This is also a good way to familiarize yourself with the program, since you can execute operations one at a time. No script and hence no input file is needed, but your keyboard entries and the resulting display are written to the output file, where you can review them later if necessary.

Interactive mode is useful even for experienced users who write scripts. For example, you can quickly test the effects of an operation, if you are unsure, before adding it to the script. It is also an excellent debugging tool: you can pause the script at any point by inserting there, temporarily, the operation I; this will take you to interactive mode, where you can examine, compare, and modify positions, display counts, or try various moves and operations; then execute an I, and the script will continue from where it stopped. Several such test points can be added to a script. A test point in a loop will stop the repetition in each cycle.

Scripts and interactive sessions can be profitably used together also outside the domain of tests and debugging. Positions are easier to define in a script, and are often needed more than once; thus, an application that is mainly interactive could start with a script where you set up the current position, and perhaps also some memory positions, rather than entering them interactively each time. Conversely, an application that is mainly in a script could start interactively if you wanted to perform a few introductory operations differently each time (the first operation in the script would then be an I).

Working in interactive mode

To start an interactive run, which requires no script and hence no input file, include the option I in the list of run options. To start it from a script, use the operation I.

When the prompt ">" is displayed, you can enter an operation, or moves and turns. You can use the arrow keys and the editing keys to recall and edit previous entries (this is especially useful for sequences of moves or turns). X ends the run. I returns to the script that initiated interactive mode or, in an interactive run, ends the run.

An operation may display a message or a position; S and its variants display the solution steps; some operations display nothing. An error message is displayed if the operation, or a move or turn, or a related position, is invalid (see chapter 12, "Error messages"). Then the prompt is displayed again and the program waits for your next entry.

Moves and turns can be entered one at a time or several in a sequence (see section 3.4, "Moves and turns"). They modify the current position. If you discover an error after a sequence was executed, you can correct it simply by reversing the moves or turns, just as you would with a real cube. For example, if instead of the sequence "1 3 5" you entered "1 3 3 5", enter next "–5 –3 5" to correct it. And you can always display the current position to confirm that it is what you expected.

Up to 30 moves and turns can be entered at one time, in any combination, separated by one or more spaces. There is no difference between entering a series of moves or turns one at a time, or together as one long sequence, or as several consecutive short sequences. The digit 0 is replaced with a random move or turn, as usual (see section 5.9, "Random moves and turns").

Like all keyboard entries, moves and turns are added to the output file, so the operation ON/OF (which switches on and off the display of moves and turns, and hence their inclusion in the output file) is less important than in scripts. ON simply repeats the entries, but is still valuable for random moves and turns, to show the actual values.

When more than one move or turn is specified, an error message like "Invalid move" can refer to any place in the sequence. To help you identify the place, Digicube displays the character that prompted the error and a few characters preceding it. The whole sequence is validated before being executed, so when an error is found, the current position is unchanged. Thus, you can recall and correct the sequence (using the arrow and editing keys), and then re-enter it.

A sequence can be specified as a *loop* by enclosing it in square brackets. The repeat count (the number of times the sequence is to be repeated, 1 to 999999999) is specified after the opening bracket. This is identical to the loops used in scripts, except that only moves and turns can be included. The sequence can have up to 30 moves and turns, and only one loop can be specified on a line. Also, no single moves or turns can be included on the line, before or after the loop (so the line must start and end with the brackets). As an example, the following sequence of turns and moves will be executed 10 times:

```
[10 (3) (-1) (3) 3 -1 3]
```

You can use X to stop a lengthy process and return to the prompt. This can be useful for loops with a large repeat count, and for some solutions. When stopping a solution, the current position remains unchanged.

The moves and turns executed, as well as the cycles of a loop, are added to their respective counts just as they are in a script (see section 5.6, "Counts"). Changing from script to interactive mode or vice versa does not alter the counts. Thus, you must use Q0 if you want to clear counts that were incremented in the script prior to beginning interactive mode.

Memories in the CC and GG lists (see section 5.7, "Memories") are checked for a match with the current position after each move or turn. Thus, the message "Match mem" or "Match goal mem" may be displayed for any one of the moves or turns in the sequence. Several consecutive messages are displayed if a match is found in several places in the sequence.

Examples

These examples illustrate the interactive mode. The keyboard entries are shown following the prompt ">".

In example 1 we generate a random position, store it in memory for constant comparison, solve the position for corners showing reverse moves, and then enter these moves. The end position must be the same as the initial one, and this is confirmed by the message.

```
> * Example 1
> q0
Clear count
> y
Scramble
> t1
> cc1
> +r
> sc
1. Anchor: ..... ... 1 5 3 -5 4 -6 4 -6
2. Corners: ..... ..... 3 -5 1 3 1 3 5 -1 5 -3
Moves: 18    Time: 1.1 sec    Sequences tried: 25,021,000

Reverse moves
1.  3 -5 1 -5 -3 -1 -3 -1 5 -3
2.  6 -4 6 -4 5 -3 -5 -1

> 3 -5 1 -5 -3 -1 -3 -1 5 -3
> 6 -4 6 -4 5 -3 -5 -1
Match mem 1: 18 moves
>
```

In example 2 we start twice with the solved position and flip 2 or 4 non-anchor edge pieces. The solution is then simply one of the flip sequences stored in the program.

```
> * Example 2
> /13
> /14
> s
1. Anchor: ---
2. Corners: ---
3. Flip 13 14:  3 1 -3 -5 1 3 1 -3 -1 -3 -1 3 1 5 1 3 1 -3
Moves: 18    Time: 0.0 sec    Sequences tried: 6

> /15
> /25
> /35
> /45
> s
1. Anchor: ---
2. Corners: ---
3. Flip 15 25 35 45:  1 5 3 1 5 -3 5 1 3 1 1 -3 -5 1 -5 -3 -1 3 -5 -1
Moves: 20    Time: 0.0 sec    Sequences tried: 6

>
```

In example 3 we use R with a valid and an invalid cube.

```
> * Example 3
> f1
> d

  1 1 1 1 1 1 1 1 1   2 2 2 2 2 2 2 2 2   3 3 3 3 3 3 3 3 3
  4 4 4 4 4 4 4 4 4   5 5 5 5 5 5 5 5 5   6 6 6 6 6 6 6 6 6

> * reset valid cube
> (5)
> v
Cube incorrectly oriented
> r
Reset position
> *
> * reset invalid cube
> ^1=2
> ^2=1
> r
Invalid cube, cannot reset position
> v
Invalid cube, some centers wrong
>
```

In example 4 we generate a random position, solve the anchor, apply to it 5 random moves and store it in the current goal position, then apply to the same position 5 other random moves. The current position is then 10 random moves away from the goal, so SG should be able to solve it in 10 moves or less.

```
> * Example 4
> y
Scramble
> sa
Anchor: ..... . 3 6 1 6 -1 4
Moves: 6    Time: 0.1 sec    Sequences tried: 522,342

> on
Disp on
> t1
> [5 0]
 -3 5 -3 -3 1
> e
> f1
> [5 0]
 -5 -5 3 -5 -3
> sg
Goal solution: ..... ..... 3 5 -3 5 5 -3 5 3 3 1
Moves: 10    Time: 0.2 sec    Sequences tried: 3,164,868

>
```

In example 5 we change the current model, enable the display option J, enter a position, and solve it.

```
> * Example 5
> +m2
Initialize M2
> +j
> p1 3 3 1 5
> p2 2 5 1 1
> p3 2 6 6 3
> p4 6 4 2 5
> p5 4 5 4 3
> p6 6 2 4 1
> s
Initial:
      6 2
      4 1

  2 5  2 6  3 3  6 4
  1 1  6 3  1 5  2 5

      4 5
      4 3

Solution: ..... ..... .. 1 3 -5 3 1 -5 3 1 3 -1 5 3
Moves: 12    Time: 1.0 sec    Sequences tried: 29,653,314

>
```

In example 6 we confirm that moves and turns have the same effect on the corner pieces of M1 and M2 cubes. We start with the solved position in M1, save the random state, perform 100 random moves and turns, and display the end position. Then we change the model to M2, restore the random state, perform the same moves and turns, and display the end position. We can confirm visually that the 8 corner pieces are identical.

```
> * Example 6
> +m1
Initialize M1
> ns
> [100 0 (0)]
> u

  ^135=415 ^136=416 ^145=153 ^146=623
  ^235=361 ^236=532 ^245=524 ^246=462
  ^13=36 ^14=41 ^15=46 ^16=25
  ^23=35 ^24=42 ^25=45 ^26=61
  ^35=62 ^36=51 ^45=23 ^46=31
  ^1=3 ^2=4 ^3=6 ^4=5 ^5=2 ^6=1

> +m2
Initialize M2
> nr
> [100 0 (0)]
> u

  ^135=415 ^136=416 ^145=153 ^146=623
  ^235=361 ^236=532 ^245=524 ^246=462

>
```

In example 7 we show that the move sequence 2 3 5 3 2 causes an M1 cube to return to its starting position every 126 cycles, and an M2 cube every 18 cycles. We monitor the current position through constant comparison.

```
> * Example 7
> y
Scramble
> t1
> cc1
> q0
Clear count
> [400 2 3 5 3 2]
Match mem 1: 630 moves, 126 cycles
Match mem 1: 1,260 moves, 252 cycles
Match mem 1: 1,890 moves, 378 cycles
> *
> +m2
Initialize M2
> y
Scramble
> t1
> cc1
> q0
Clear count
> [100 2 3 5 3 2]
Match mem 1: 90 moves, 18 cycles
Match mem 1: 180 moves, 36 cycles
Match mem 1: 270 moves, 54 cycles
Match mem 1: 360 moves, 72 cycles
Match mem 1: 450 moves, 90 cycles
>
```

In example 8 we demonstrate the principle of parity, starting with any position.

```
> * Example 8
> u
Cube OK
> * demonstrate edge parity (must flip 2 edges)
> /15
> u
Invalid cube, some edges flipped wrongly
> /35
> u
Cube OK
> * demonstrate corner parity (must flip 2 corners, cw & ccw)
> /135
> u
Invalid cube, some corners flipped wrongly
> /235
> u
Invalid cube, some corners flipped wrongly
> \235
> \235
> u
Cube OK
>
```

CHAPTER 7

Scripts

If you need to execute more than once a series of operations, or a sequence of moves and turns, you can include them in a script and store them in the input file. The input file is an ordinary text file; you create it and access it with any text editor (see chapter 2, "Installing and running Digicube"). For proper alignment of the text data, use a monospaced font (like Courier) and no word wrap.

The input file

Any number of scripts can be stored in the input file. Active scripts are identified by a label, #1–#999, which starts in the first position on a line; the script can then start on the same line (after one or more spaces) or on the following line. To execute a certain script, you specify the run option S with that script's label number, S1–S999 (see chapter 4, "Run options"). If you omit the S, the first script in the file is executed, and it should have no label; if it does, you must invoke it with S, otherwise the label would be seen as an invalid operation.

Labels have no numeric significance, and need not be numerically consecutive in the file. If the same label is used more than once, the first script with that label is executed. Here is an example of an input file with a few scripts:

```
* The position of a cube is entered and solved with the run option d.
p 2 2 6 6 1 2 1 6 5   3 5 5 4 2 1 1 1 4   4 6 4 3 3 3 2 5 6
  1 4 2 5 4 1 2 5 3   5 3 3 6 5 2 5 4 3   6 2 4 1 6 3 1 4 6
+d s x


* The current model is changed to m3, then the position of a pyramid
* is entered and solved with the run option a.
* (This script is inaccessible, because it has no label.)
+m3 p  3 1 2 4 2 4  4 2 4 2 4 3  3 3 2 1 1 2  1 3 1 1 3 4  +a s x


This position is solved 6 times with different m values,
resulting in different solutions.
#4
p 5 4 1 3 1 3 6 3 1   2 6 1 1 2 1 1 4 3   5 1 2 4 3 1 2 5 4
  6 6 5 2 4 5 6 2 3   6 4 2 6 5 5 5 5 4   4 3 3 2 6 2 4 6 3
t1 m135 s  f1 m153 s  f1 m315 s  f1 m351 s  f1 m513 s  f1 m531 s x


#21
* This repeated sequence causes the cube to return to its starting
* position only after 1260 cycles. To prove this, the current position
* is compared constantly with the original, random position.
y t1 cc1 q0 [1260 1 3 3 -4 6 -4] x
```

```
The cube is scrambled with 10 random moves, then the position is solved directly.
#15 [10 0] sd x


In this position, the 4 edges in each face have the color of the
opposite face, forming a nice pattern.
#101
p 1 2 1 2 1 2 1 2 1  2 1 2 1 2 1 2 1 2  3 4 3 4 3 4 3 4 3
  4 3 4 3 4 3 4 3 4  5 6 5 6 5 6 5 6 5  6 5 6 5 6 5 6 5 6
* to create it with a real cube, solve the position with the r run option
* in order to display the reverse moves, then apply the moves to a solved cube
+r s x


This script specifies a position, solves the anchor, then solves the 9 remaining
edge pieces using sg in two steps. The original goal position is the anchor
goal position; it is modified by specifying 5 solved edge pieces for the first
step, and 4 more for the second step. The final position is displayed with u.
#30
p 3 3 1 1 1 6 2 3 4   5 2 4 2 2 4 5 2 5   6 4 2 5 3 6 3 1 5
  6 3 1 4 4 5 6 3 2   1 5 3 6 5 1 4 6 2   4 5 3 4 6 2 1 1 6
sa e ^15 ^25 ^35 ^45 ^16 ^36 e sg
e ^13 ^14 ^23 e sg u x
```

This example illustrates also the use of notes and comments in the file. Lines with "*" in the first position are ignored when running a script, so they can be mixed freely with script lines. But you need the "*" only for lines that lie in the path of execution of the script, between its label and the X that ends it. Elsewhere in the file you can have any text, since Digicube reads only scripts: given a certain label, it reads the file from the beginning searching for that label and ignoring any other text; and when the script ends, it doesn't read past it.

The first script in the example has no label and is executed without using the option S. Digicube treats the beginning of the file as script, and the "*" is needed because that line lies in the path of execution.

The second script (with the position of a pyramid) is inaccessible, since it has no label. To execute it, add a label or move it to the beginning of the file.

The third script has label #4 and is executed by specifying S4. Its note needs no "*", because those lines are not in the path of execution.

The fourth script has label #21 and is executed by specifying S21. Its note needs the "*", because those lines follow the label and are, therefore, in the path of execution.

The fifth script is on the same line as its label, #15, and is executed by specifying S15.

The sixth script has label #101 and is executed by specifying S101. Its first note needs no "*", but the second one does, because it is embedded in the script and is, therefore, in the path of execution.

The seventh script has label #30 and is executed by specifying S30. Its note needs no "*".

The input file, then, can hold more than just scripts. You can use it to store any related text: old scripts, unfinished scripts, various positions, results of experiments copied from the output file, and miscellaneous documentation. All that matters is that the script you want to execute has a label and ends with X. (Alternatively, move or copy the script to the beginning of the file, with no label.)

Working with scripts

The elements that make up a script are operations, moves, and turns. They are separated by one or more spaces, and are executed in the order in which they are read. Sequences of moves and turns can be mixed with operations, and can have any length (see section 3.4, "Moves and turns"). A group of elements that must be executed repeatedly can be specified as a loop (this is discussed later).

There can be any number of elements on a line, and a script can span any number of lines. The end of a line has no significance, so you can break up sequences of operations, or sequences of moves and turns, or the color digits that define a position, in any way you like. There are two restrictions: a turn and its enclosing parentheses must be on the same line; and the optional character string attached to the operations D, J, U, A, L, and Q must be entirely on one line. Blank lines can be inserted both within a script and between scripts, and have no significance.

An operation may display a message or a position; S and its variants display the solution steps; some operations display nothing. Moves and turns are displayed when the operation ON is in effect. All displayed lines are also added to the output file.

Errors discovered while executing a script are due to invalid script elements or invalid positions. Digicube displays a message and usually ends the run (see chapter 12, "Error messages"). You must then correct the script and re-execute it. V and VG, which only verify the position, continue after an error message. With some messages, you are given the choice to end the run or skip the element that caused the error and continue with the next one. To help you identify the place in the script where the error occurred, Digicube displays, along with the message, the element that prompted the error and a few characters preceding it.

The moves and turns executed in the script, as well as the cycles of loops, are counted by the program (see section 5.6, "Counts"). The counts are 0 when the script starts, and can be cleared anywhere in the script with Q0. Q displays the current counts. QC, when used in a loop, displays the current cycle.

The operation I, when encountered in a script, causes Digicube to interrupt the script and begin interactive mode. Then, an I in interactive mode will cause Digicube to return to the script and continue from where it left. An I encountered in a loop will do this in each cycle.

You can stop a running script by pressing X. The script will stop after the element (operation, move, or turn) being executed. This may be needed in a loop with a large repeat count. If pressing X during a solution, the solution is stopped and the current position remains unchanged. Once stopped, a message is displayed with the choice to end the run or continue with the next element.

Loops

A sequence of elements (any combination of operations, moves, and turns) can be specified as a *loop* by enclosing it in square brackets. The repeat count (the number of times the sequence is to be repeated, 1 to 999999999) is specified after the opening bracket, before the sequence of repeated elements. Spaces around the brackets are optional. In the following example, 100 random positions are generated, displayed, and solved:

```
[100 y qc d s] x
```

Any number of loops can be used in a script, but one at a time: you cannot enclose one loop within another. Single elements can be used freely between loops. An empty loop (a repeat count but no elements) is valid.

A loop can start and end anywhere on a line, and can span any number of lines. Multiple lines are needed when there are many elements in the loop, or if you want to separate elements for readability.

A loop cannot exceed 2000 bytes. Each move or turn needs one byte; most operations need one or two bytes; M, "+", "–", "/", and "\" need three bytes; "^" needs five bytes; D, J, U, A, L, and Q, if using a character string, need one extra byte plus one byte for each character in the string (up to the number of characters actually displayed). If the limit is exceeded, the message "Loop limit exceeded" is displayed, showing the element where this occurred. If the limit is a problem, consider custom operation C5, which lets you execute repeatedly scripts of any size.

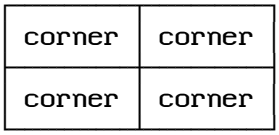
The following operations cannot be included in a loop: X, Z, P and its variants P1–P6, PG and its variants PG1–PG6. X and Z are not useful in a loop; and if you need to specify a position in a loop, use P or PG before the loop, copy the position to a memory with T or TG, and restore it inside the loop with F or FG.

CHAPTER 8

Model M2 (Pocket cube)

M2 is the 2× cube (known as Pocket cube). You select M2 as a run option, or during a run with the operation +M2. It is assumed that you are familiar with the earlier chapters, so only the differences from M1 are discussed here.

M2 is similar to M1, but much simpler. Its 8 corner pieces are the same as in M1, but it has no edge pieces or center pieces. This is the arrangement of pieces in each face:



If you ignore the references to edge pieces and to center pieces, the entire discussion in chapter 3, "Definitions and terminology", applies also to M2. Thus, the face schematics and the conversion tables, digits-to-sides and digits-to-colors, are the same; and, since most 2× cubes currently available have the same color scheme as the 3× cubes, the colors shown for M1 can also be used for M2. The moves and turns also are the same, and the sequences shown as examples for M1 have the same effect with M2. The run options and operations discussed in chapters 4 and 5 are the same (if you ignore the references to edge and center pieces), apart from the differences mentioned here.

Positions

The locations of the corner pieces are three-digit numbers: 135, 136, 145, 146, 235, 236, 245, 246. These are the same as the M1 corner locations (see section 3.3, "Pieces").

There are only 4 digits per face, and they are entered and read for each face in the usual way: top to bottom, and left to right in each row. Since M2 positions depict 6 faces of 4 digits, the list of colors displayed by the operation D and the run option D has 24 digits; the operations P and PG must be followed by 24 digits; and their variants, P1–P6 and PG1–PG6, must be followed by 4 digits. Here is an example of P and the equivalent P1–P6:

```
p  6 3 5 3   2 2 2 6   5 4 1 4   6 4 1 4   3 1 5 5   6 2 3 1

p1  6 3 5 3
p2  2 2 2 6
p3  5 4 1 4
p4  6 4 1 4
p5  3 1 5 5
p6  6 2 3 1
```

The solved position, in the four display formats (list of colors, rows of colors, list of pieces, schematic), looks like this:

```
1 1 1 1   2 2 2 2   3 3 3 3   4 4 4 4   5 5 5 5   6 6 6 6

     6 6
     6 6

2 2  3 3  1 1  4 4
2 2  3 3  1 1  4 4

     5 5
     5 5

^135=135 ^136=136 ^145=145 ^146=146
^235=235 ^236=236 ^245=245 ^246=246

        +---+---+
        ¦ 6 ¦ 6 ¦
        +---+---+
        ¦ 6 ¦ 6 ¦
+---+---+---+---+---+---¦---+---+
¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---+---+---+
¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 1 ¦ 1 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---¦---+---+
        ¦ 5 ¦ 5 ¦
        +---+---+
        ¦ 5 ¦ 5 ¦
        +---+---+
```

And here is a scrambled position in the standard orientation (note how the faces and digits correspond in the four formats):

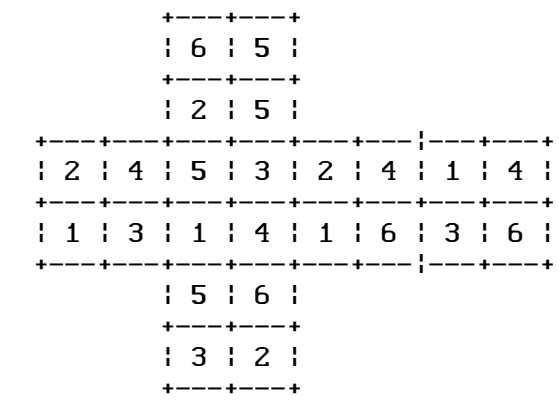
```
2 4 1 6   2 4 1 3   5 3 1 4   1 4 3 6   5 6 3 2   6 5 2 5

     6 5
     2 5

2 4  5 3  2 4  1 4
1 3  1 4  1 6  3 6

     5 6
     3 2

^135=146 ^136=235 ^145=632 ^146=415
^235=315 ^236=452 ^245=163 ^246=246
```



At the beginning of a run (or when changing the current model to M2), the current position is identical to the solved position, and the current goal position is identical to the anchor goal position (piece 246 correct and 9's for the other 7 pieces). Memories 1 to 97 have the solved position, memory 98 has the anchor goal position, and memory 99 has 9's for all pieces.

Orientation

For M1, the center pieces determine its orientation; thus, lacking center pieces, a different method must be used for M2. The cube is in the standard orientation when piece 246 is in its solved location and orientation; that is, colors 2, 4, 6 are *left*, *back*, *up*, respectively (in piece notation, ^246=246). In the standard orientation, therefore, piece 246 is in the same location and orientation as piece 246 in M1.

If piece 246 is incorrect, the current position cannot be reset, and Digicube displays the message "Invalid cube, cannot reset position". This can happen if you perform R before V or S and its variants (which would discover the problem through their validity checks). Similarly, to reset the current goal position with RG, piece 246 must be correct and cannot have 9's; the error message is "Invalid goal, cannot reset position".

To place a real cube in the standard orientation, you must turn it until you can confirm visually that piece 246 is in that location and correctly oriented. With a solved cube, all colors will then match the sides (color 1 at side 1, color 2 at side 2, and so on). With a scrambled cube, the colors can form any pattern.

In addition to its role in determining orientation, piece 246 serves as the M2 anchor. It is thus the counterpart of the M1 anchor (which includes 3 edge pieces in addition to the corner 246), and its role is similar: when it is in the correct location and orientation, we only need moves involving 3 faces (1, 3, 5, which are at sides 1, 3, 5 – *right*, *front*, *down*), since these moves can access all remaining pieces. So with M2, placing the cube in the standard orientation also solves the anchor and prepares the cube for the solution moves.

Solutions

There are three solution types, and all need only one step: S is the standard solution, SD solves directly, and SG solves for the current goal position. As explained for M1 solutions, the search method guarantees that these one-step solutions discover the shortest sequence of moves (allowing for the restriction to three faces).

The other variants, SA, SC, and SE, will display "Invalid for current model" if attempted with M2. The operations M (to set the order of trial moves) and K (to limit move sequences) are the same as for M1. The sequences that are ignored in searches (because they contain moves that do nothing or have the same effect as others) are the same as for M1, so the average number of moves tried is reduced from 6 (1, -1, 3, -3, 5, -5) to about 4.45.

The reason there is no need for more than one step is that the longest sequence has only 14 moves, so a direct solution is always possible. (Only about one in 15,000 random positions requires 14 moves.) The average sequence is 10.7 moves, and the average time is 0.5 of a second. Here is an example of the display:

```
Initial:  2 4 3 1   2 5 1 1   2 6 3 2   1 4 5 4   6 5 6 3   6 5 4 3
Solution: ..... ..... 5 5 -1 3 -1 3 5 3 -5 -3
Moves: 10    Time: 0.2 sec    Sequences tried: 4,625,336
```

Since S too is a direct solution, the only difference between it and SD is how they treat solutions stopped by the K limit (see operation K). They function the same as they do for M1: with SD, when the limit is exceeded the solution process is terminated (and the current position remains unchanged); with S, the initial position is modified with a few random moves and the solution process is restarted. The default limit is K13, and with this limit the difference between S and SD is only for the very rare 14-move solutions (see above); use S (or SD with K14) if you want to be sure that a solution is always discovered.

SG too functions the same as it does for M1. However, since it finds the solution in a short time, SG has an additional function: it can be used to reach directly any position from any other position, by specifying a complete goal position (no 9's) as the target position.

When you enter the position of a real cube with P or its variants P1–P6, the cube is normally in the standard orientation. Thus, rather than using the center pieces to identify the cube's faces, as with M1, use piece 246: when it is in the corner *left-back-up*, face 1 is *right*, face 2 is *left*, face 3 is *front*, and so on. Then you must hold the cube in the standard orientation for each move, throughout the solution process. Again, rather than watching the center pieces, as with M1, watch piece 246: it must always be in the corner *left-back-up*.

CHAPTER 9

Model M3 (Pyraminx)

M3 is the 3× pyramid (known as Pyraminx). You select M3 as a run option, or during a run with the operation +M3. It is assumed that you are familiar with the earlier chapters, so only the differences from M1 are discussed here.

M3 is much simpler than M1, but, because of its shape, very different. It has 4 corner pieces and 6 edge pieces, but no center pieces. The corner pieces are called axials, and, although they are very similar to the M1 corner pieces, this is not obvious. Each axial has a small tip attached, which looks like the actual corner but is in fact a trivial piece that is not part of the puzzle: it simply rotates on the axial piece and can always be aligned with it so as to match their colors. The 6 edge pieces are placed between the axial pieces and are like the edge pieces of M1.

M3 has only 4 sides, faces, colors, and types of moves and turns, and therefore only the digits 1 to 4 are needed. Otherwise, these concepts are the same as for M1, so the discussion in chapter 3, "Definitions and terminology", applies also to M3. The run options and operations discussed in chapters 4 and 5 are the same (if you ignore the references to center pieces), apart from the differences mentioned here.

The digit 5 or 6 for a color, move, or turn causes the message "Invalid for current model". (A digit like 7, which is invalid for all models, causes the usual message: "Invalid data", "Invalid move", etc.) This message is also caused by the operations P5 and P6, PG5 and PG6, N3/N6 (all 4 faces are used for random moves), M, SA, SC, and SE.

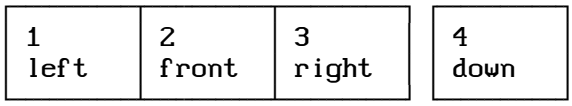
In section 5.9, "Random moves and turns", we saw that Digicube avoids combinations of random moves or turns that would be useless or would produce the same result as a shorter combination. In M3 the only difference is that combinations of just 2 or more consecutive identical values are avoided (for example, "1 1" is the same as "–1", "1 1 1" does nothing, etc.).

Positions

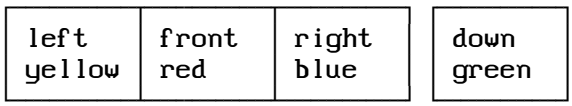
Each face has 3 axial pieces and 3 edge pieces, which it shares with the adjacent faces. The faces are triangular, but if we exclude the 3 irrelevant tips, they are in effect rectangles with 2 rows of 3 pieces, arranged like this:

edge	axial	edge
axial	edge	axial

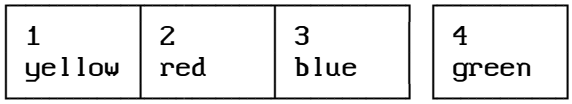
The normal view of the pyramid, when entering its position or when solving it, is with one of its faces as *front*; other faces are then *left* and *right*, and the fourth one is *down*. These 4 sides form the fixed frame of reference within which the pyramid rotates, and the digits are assigned as follows:



There are 4×3=12 possible orientations (each one of the 4 faces can be *down*, and in each case 3 faces can be *front*), and we choose the following one as the standard orientation:



This color scheme matches most pyramids currently available. Still, as was the case with M1, you can choose any other orientation as the standard one. By combining the previous two schematics, we derive the following correspondence between digits and colors:



Note that, with this choice for the standard orientation, colors 2, 3, and 4 are the same as for M1, which is convenient if you use both models.

The locations of the edge pieces are two-digit numbers: 12, 13, 14, 23, 24, 34. The locations of the axial pieces are three-digit numbers: 123, 124, 134, 234. These locations are different from the M1 edge and corner locations (see section 3.3, “Pieces”).

Since M3 positions depict 4 faces of 6 digits, the list of colors displayed by the operation D and the run option D has 24 digits; the operations P and PG must be followed by 24 digits; their variants, P1–P4 and PG1–PG4, must be followed by 6 digits. Here is an example of P and the equivalent P1–P4:

```
p  2 1 3 1 3 4   1 2 3 2 4 3   2 3 4 1 2 4   4 3 1 2 1 4

p1  2 1 3 1 3 4
p2  1 2 3 2 4 3
p3  2 3 4 1 2 4
p4  4 3 1 2 1 4
```

Similarly to M1, positions are displayed or entered by listing or specifying the 4 faces in the order of the 4 sides, 1 to 4: first the face at side 1, then the face at side 2, and so on. Within each face, the 6 pieces are treated as 2 rows of 3 digits, read top to bottom, and left to right in each row. If faces are seen as triangles, the bottom of faces 1, 2, and 3 is the base of their triangle. For face 4, you expose it by tilting the pyramid around the edge shared with face 3, and the bottom is then the base of its triangle (face 3 is then *down*).

The solved position, in the four display formats (list of colors, rows of colors, list of pieces, schematic), looks like this:

```
1 1 1 1 1 1   2 2 2 2 2 2   3 3 3 3 3 3   4 4 4 4 4 4

1 1 1  2 2 2  3 3 3   4 4 4
1 1 1  2 2 2  3 3 3   4 4 4

^123=123 ^124=124 ^134=134 ^234=234
^12=12 ^13=13 ^14=14 ^23=23 ^24=24 ^34=34

+---+---+---¦---+---+---¦---+---+---+  +---+---+---+
¦ 1 ¦ 1 ¦ 1 ¦ 2 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 3 ¦  ¦ 4 ¦ 4 ¦ 4 ¦
+---+---+---+---+---+---+---+---+---+  +---+---+---+
¦ 1 ¦ 1 ¦ 1 ¦ 2 ¦ 2 ¦ 2 ¦ 3 ¦ 3 ¦ 3 ¦  ¦ 4 ¦ 4 ¦ 4 ¦
+---+---+---¦---+---+---¦---+---+---+  +---+---+---+
```

And here is a scrambled position in the standard orientation (note how the faces and digits correspond in the four formats):

```
4 1 3 4 2 2   2 2 4 4 4 3   3 3 1 4 3 1   1 1 2 3 1 2

4 1 3  2 2 4  3 3 1   1 1 2
4 2 2  4 4 3  4 3 1   3 1 2

^123=123 ^124=241 ^134=413 ^234=342
^12=32 ^13=41 ^14=21 ^23=43 ^24=42 ^34=31

+---+---+---¦---+---+---¦---+---+---+  +---+---+---+
¦ 4 ¦ 1 ¦ 3 ¦ 2 ¦ 2 ¦ 4 ¦ 3 ¦ 3 ¦ 1 ¦  ¦ 1 ¦ 1 ¦ 2 ¦
+---+---+---+---+---+---+---+---+---+  +---+---+---+
¦ 4 ¦ 2 ¦ 2 ¦ 4 ¦ 4 ¦ 3 ¦ 4 ¦ 3 ¦ 1 ¦  ¦ 3 ¦ 1 ¦ 2 ¦
+---+---+---¦---+---+---¦---+---+---+  +---+---+---+
```

In the 2-dimensional schematic, face 4 is separated from the others because of its orientation. To be strict, it ought to be drawn upside down, with its base joined to the base of face 3 and its rows and columns reversed. But it is more natural to view it like the other faces, with its base at the bottom, and to read the rows left to right. This depiction also matches the other display formats and the way the position is specified when entering it.

At the beginning of a run (or when changing the current model to M3), the current position is identical to the solved position, and the current goal position has correct axial pieces and 9's for the edge pieces (an arbitrary choice). Memories 1 to 97 have the solved position, memory 98 is the same as the current goal position, and memory 99 has 9's for all pieces.

Orientation

For M1, the center pieces determine its orientation; thus, lacking center pieces, a different method must be used for M3. The pyramid is in the standard orientation when axial piece 123 is in its solved location and orientation; that is, the axial with its tip piece are at the top of the pyramid, and colors 1, 2, 3 are *left*, *front*, *right*, respectively (in piece notation, ^123=123).

You can easily determine from a displayed position whether it is in the standard orientation. The list of pieces, obviously, must have ^123=123. In the list of colors, piece 123 is represented by the second

digit in the first three faces, and in the rows of colors and the schematic by the middle digit in the top row of the first three faces; so these digits must be 1, 2, 3, in this order. (See the examples shown earlier.)

If piece 123 is incorrect, the current position cannot be reset, and Digicube displays the message "Invalid pyramid, cannot reset position". This can happen if you perform R before V or S and its variants (which would discover the problem through their validity checks). Similarly, to reset the current goal position with RG, piece 123 must be correct and cannot have 9's; the error message is "Invalid goal, cannot reset position".

To place a real pyramid in the standard orientation, you must turn it until you can confirm visually that piece 123 is in that location and correctly oriented. With a solved pyramid, all colors will then match the sides (color 1 at side 1, color 2 at side 2, and so on). With a scrambled pyramid, the colors can form any pattern.

Solutions

Unlike M1 and M2, M3 solutions cannot be restricted to a subset of move types. Even with a portion solved (one face, or one axial piece and the adjacent edge pieces), all move types, 1 to 4, are still needed to solve the rest of the pyramid. Thus, there is no anchor, and no anchor goal position. Even though four move types must be tried (as opposed to three for M1 and M2), M3 is relatively simple and direct solutions are fast.

The sequences that are ignored in searches (because they contain moves that do nothing or have the same effect as others) are somewhat different from M1 and M2: the combinations of moves are consecutive opposites like "1 –1" and "–1 1" (which do nothing), and pairs like "1 1" and "–1 –1" (which are the same as "–1" and "1", respectively). This reduces the average number of moves tried from 8 (1, –1, 2, –2, 3, –3, 4, –4) to 6.

Because there is no anchor, the current position can be in any orientation when executing S or its variants. The standard orientation described earlier is needed only to help users hold the pyramid in a specific way when entering its position and when applying the solution moves.

There are three solution types, and all need only one step: S is the standard solution, SD solves directly, and SG solves for the current goal position. As explained for M1 solutions, the search method guarantees that these one-step solutions discover the shortest sequence of moves.

The other variants, SA, SC, and SE, will display "Invalid for current model" if attempted with M3. The operation M, which sets the order of trial moves, will also display this message, because the order cannot be modified; it is always 1 2 3 4 –1 –2 –3 –4. The operation K (to limit move sequences) is the same as for M1.

The reason there is no need for more than one step is that the longest sequence has only 11 moves, so a direct solution is always possible. (Only about one in 30,000 random positions requires 11 moves.) The average sequence is 7.8 moves, and the average time is 0.1 of a second. Here is an example of the display (the "reset" line is discussed later):

```
Initial:  1 1 1 3 4 4   2 2 3 1 3 3   4 3 4 4 1 4   2 2 2 1 3 2
Solution: ..... ... 3 -1 2 -3 -1 4 2 3
Reset position
Moves: 8    Time: 0.1 sec    Sequences tried: 1,153,649
```

Since S too is a direct solution, the only difference between it and SD is how they treat solutions stopped by the K limit (see operation K). They function the same as they do for M1: with SD, when the limit is exceeded the solution process is terminated (and the current position remains unchanged); with S, the initial position is modified with a few random moves and the solution process is restarted. The default limit is K11, and with this limit a solution is always discovered, so there is no difference between S and SD.

SG too functions the same as it does for M1, but the goal position can be in any orientation. Since it always finds the solution, SG has an additional function: it can be used to reach directly any position from any other position, by specifying a complete goal position (no 9's) as the target position.

When you enter the position of a real pyramid with P or its variants P1–P4, it can be in any orientation. Digicube solves it starting from that orientation, and expects you to do the same with the real pyramid. It is important, therefore, that you hold the pyramid, when applying the moves, in the same orientation as when you entered its position (otherwise the moves will not lead to the solved position). This is why it is a good idea to always use the standard orientation, easily recognizable by the piece 123 in location 123 (the top of the pyramid), rather than a different orientation each time.

The solved position reached by Digicube can be in any orientation. But, to be consistent, when this is not the standard orientation, S and SD reset the position and display "Reset position" (as shown in the example above). Since there are 12 possible orientations, this happens on average in 11 out of 12 solutions. So the final position is always in the standard orientation for S and SD. For SG too, a match is deemed successful in any orientation; but the final position is not reset.

You must bear this in mind if you use the run option R to re-create the initial scrambled position with a real pyramid. R displays the reverse solution moves, but these moves do not take into account the final reset operation (which involves only turns). Thus, you must also use the run option D1 or D2, so as to display the position before it is reset. You first orient the solved pyramid according to this position, and then apply the reverse moves. Here is an example (using D1):

```
Initial:  3 2 4 4 2 4   1 3 1 2 4 1   2 1 1 4 3 2   4 3 3 1 2 3
Solution: ..... ... -1 -2 -1 3 2 -1 -4 2
End:      2 2 2 2 2 2   1 1 1 1 1 1   4 4 4 4 4 4   3 3 3 3 3 3
Reset position
End:      1 1 1 1 1 1   2 2 2 2 2 2   3 3 3 3 3 3   4 4 4 4 4 4
Moves: 8    Time: 0.1 sec    Sequences tried: 1,732,589

Reverse moves
  -2 4 1 -2 -3 1 2 1
```

Changeable reference

When solving a real pyramid, you must hold it in the same orientation for each move, throughout the solution process. But, without an anchor that can act as reference, it is easy to get confused. Even if you start in the standard orientation, you cannot count on piece 123, because it will not remain in the same location.

The problem is made worse by the large number of pieces that rotate for each move. In M1, only 8 out of 20 pieces rotate, and we have the center pieces as reference; in M2, 4 out of 8 pieces rotate, and

we have the anchor piece as reference; in M3, 6 out of 10 pieces rotate, and we have no reference. There are more rotating pieces than stationary ones, so it is easy to forget that it is in fact the latter that determine the orientation.

The solution is to use a changeable reference; that is, a reference that depends on the rotating face. This is the axial piece opposite that face, plus the 3 adjacent edge pieces; in other words, the 4 stationary pieces. We could call these pieces a changeable anchor. So, for each move, to ensure that the pyramid remains in the same orientation, ensure that these pieces remain in the same position and orientation while rotating the face.

You must also be careful with turns. It helps if you remember that, just as with M1 and M2, the face in the side specified by the turn (1 for *left*, 2 for *front*, etc.) turns but stays at the same side (if *left* it stays *left*, if *front* it stays *front*, etc.); it is the other 3 faces that change sides.

CHAPTER 10

Back sequences

In a position, pieces are said to be wrong if they are in the wrong location, or, when in the right location, if they are flipped. Solutions are designed to search for move sequences leading from a position with wrong pieces to the solved position. But the reverse search, from the solved position to a position with wrong pieces, is also useful. Digicube lets you perform such reverse searches, and the move sequences discovered are known as back sequences.

The ultimate purpose of back sequences is to provide ready-made one-step solutions for various positions. If we discovered in the past a back sequence leading to a position with a particular combination of wrong pieces, we can now solve that position, each time we encounter it, simply by applying those moves in reverse. Typically, back sequences are used for positions with just a few wrong pieces.

The benefits of back sequences can be illustrated with a general example. If we seek a direct solution in 18 moves for a particular position with two wrong corner pieces, and if such a solution exists, the operation SD will discover it in 3–10 hours (depending on the moves in the sequence that is found first). But if we start with the solved position, apply all possible 18-move sequences, and select from the end positions those with two wrong corner pieces, we will have a complete list of positions with two wrong corner pieces (which includes, of course, our original position). And by showing the respective sequences in reverse, we will also have the solutions for these positions. This search will take about 13 hours. It will provide the solution, however, not for one position but for every position with two wrong corner pieces. And it only needs to be done once. (If actually performing this search, Digicube discovers 1410 sequences with all 42 unique combinations of 2 flipped corner pieces, excluding the anchor; these are the only wrong corner pieces possible with 18-move sequences.)

A practical demonstration of this concept is offered by the edge steps in the standard solution, S (see section 5.8, “Solutions”). These steps may end with the solved position, but they end usually with two or four wrong edge pieces (in the correct position but flipped). A sequence of 16, 18, or 20 moves is then applied to reach the solved position. These sequences were discovered as back sequences, and are now stored in the program. The search took many hours of computer time, but was done only once. There are 36 combinations of two flipped edge pieces, and 126 combinations of four. From the thousands of back sequences leading to these combinations, the shortest and most convenient ones were selected and are now part of the solution process. (The full lists of back sequences discovered are included in the digicube.zip package.)

Options

The back sequence search is implemented as a custom operation rather than a regular operation, because of the many options that you must specify: number and type of wrong pieces (wrong location or only flipped), sequence length, and so on. (See “C4 List back sequences” in chapter 11, “Custom operations”.)

One option lets you restrict the search to sequences starting with a certain combination of moves: you can specify a short sub-sequence of one to four moves, and only sequences that start with these moves will be considered. This reduces the search time, on average, by a factor of about 4.45 for each move

in the sub-sequence (more for moves 1, 3, 5, less for -1, -3, -5). It also allows you to divide the search into several parts that can run simultaneously in different cores (remember to specify different output files) or on different computers.

Another option lets you specify any initial position, instead of starting from the solved position. This is useful when you want to discover sequences leading to a particular position, starting from positions that differ from it in just a few pieces. Wrong pieces are then wrong relative to that position, not to the solved position. Since back sequences use only the three move types, 1, 3, and 5, the anchor must be correct in the initial position; if it is not, or if the position is invalid, Digicube will display an error message.

Model M3 has no anchor (see chapter 9, "Model M3"), so the axial piece 123 is used as reference when deciding whether a piece is correct or wrong in the end position. Specifically, at the end of each back sequence tried, the pyramid is placed in the standard orientation (piece 123 in location 123 and correctly oriented) before counting the wrong pieces.

In addition to the sequences found (and, optionally, their reverse), Digicube lists the actual wrong pieces in each end position, sorted numerically by their location. The pieces are shown using the standard expression **^loc=col** (location and colors, and hence their orientation, as explained in section 3.3, "Pieces"). Here is an example of the C4 prompts and answers, and a small portion of the resulting sequences:

```
Number of wrong pieces (0 - 16): 4
Type of wrong pieces (1 - 3)
1 for edges only, 2 for corners only, 3 for both: 3
1 for flipped in place only, 2 for wrong location only, 3 for both: 1
Sequence lengths (up to 10 lengths of 1 - 20): 16
List only sequences starting with these (up to 4) moves:
R to show also the reverse moves: r
S to start with script, else starts with solved position:
Press Enter to continue, X to cancel:

Sequence length: 16
...
  1 1 5 3 -1 3 5 1 5 5 -1 -3 5 -3 -1 -5    ^14=41 ^15=51 ^35=53 ^36=63
  Reverse: 5 1 3 -5 3 1 -5 -5 -1 -5 -3 1 -3 -5 -1 -1
  1 3 1 1 -3 -5 1 -5 -3 -1 3 3 1 5 -3 5    ^13=31 ^15=51 ^36=63 ^45=54
  Reverse: -5 3 -5 -1 -3 -3 1 3 5 -1 5 3 -1 -1 -3 -1
  1 3 -1 3 -5 -3 5 3 -5 -3 5 -3 1 3 -1 -3    ^13=31 ^36=63 ^136=613 ^145=451
  Reverse: 3 1 -3 -1 3 -5 3 5 -3 -5 3 5 -3 1 -3 -1
  1 3 -1 3 -5 -3 5 -3 1 3 -1 3 -5 -3 5 -3    ^135=513 ^136=613 ^145=514 ^236=623
  Reverse: 3 -5 3 5 -3 1 -3 -1 3 -5 3 5 -3 1 -3 -1
  1 3 -1 -3 1 3 -1 3 -5 -3 5 3 -5 -3 5 -3    ^13=31 ^15=51 ^135=513 ^236=362
  Reverse: 3 -5 3 5 -3 -5 3 5 -3 1 -3 -1 3 1 -3 -1
...
```

Since the wrong pieces are displayed in a standard format (sorted by location, separated by one space), you can use the text editor's search feature to find, in a long list of sequences, not only individual pieces but also specific combinations of pieces, or pieces flipped in a specific way.

An interesting search is for back sequences leading to a position identical to the initial position. You perform such a search simply by specifying 0 wrong pieces. The starting position is then irrelevant, but, with a real cube, it is easier to note the effect when starting with the solved position. These sequences are known as *do-nothing* sequences (or *identity* sequences), and there are many of them: 24 of 12 moves (the shortest), 96 of 14 moves, 792 of 16 moves, 5556 of 18 moves, etc. Here are the first three sequences in each one of the four lists (the full lists are included in the digicube.zip package):

```
Sequence length: 12
  1 3 -1 -5 3 5 -3 -1 5 1 -5 -3
  1 3 -5 -3 5 1 -3 -1 3 5 -1 -5
  1 -3 -1 3 5 -1 -5 1 3 -5 -3 5
...

Sequence length: 14
  1 1 3 -1 -5 3 5 -3 -1 5 1 -5 -3 -1
  1 1 3 5 -1 -5 1 3 -5 -3 5 1 -3 1
  1 1 3 -5 -3 5 1 -3 -1 3 5 -1 -5 -1
...

Sequence length: 16
  1 1 3 5 -3 -1 5 1 -5 -3 1 3 -1 -5 1 1
  1 1 -3 1 3 -1 -5 3 5 -3 -1 5 1 -5 1 1
  1 1 -3 -1 5 1 -5 -3 1 3 -1 -5 3 5 1 1
...

Sequence length: 18
  1 1 3 1 3 -1 -5 3 5 -3 -1 5 1 -5 3 3 1 1
  1 1 3 1 3 -5 -3 5 1 -3 -1 3 5 -1 -5 -3 1 1
  1 1 3 1 3 -5 -3 -5 -1 5 -3 -5 1 5 3 5 -3 1
...
```

Note that the do-nothing sequences discovered are non-trivial. A trivial sequence would be one where, for example, the moves in the second half are the reverse of those in the first half. Since move combinations like "1 -1" and "-3 3", which do nothing, are specifically avoided everywhere in a sequence (see below), trivial sequences are automatically excluded. (A trivial sequence requires at least one such combination of moves, in the middle.)

There is additional information about the available options under C4 in chapter 11, "Custom operations".

The search

Digicube discovers back sequences the same way it discovers sequences in each step of a solution, except that here they have a fixed length. Also, the search doesn't end when a successful sequence is found; it lists all successful sequences. Only sequences consisting of combinations of three move types, 1, 3, and 5, are considered. The anchor is correct and undisturbed throughout the search process. So, obviously, the wrong pieces shown never include the four anchor pieces.

As in solutions, to save time, Digicube ignores sequences containing moves that do nothing or have the same effect as others (see section 5.8, "Solutions"), which reduces from 6 to about 4.45 the average number of moves tried for each move added to the sequence length. This means that the time needed to discover all sequences of a certain length is longer by a factor of about 4.45 than the time for sequences one move shorter: for 2 wrong pieces, approximately 2 minutes for 14 moves, 9 for 15 moves, 40 for 16 moves, etc. The time increases significantly when increasing the number of wrong pieces, but is not affected by the types selected.

For model M3, the ignored sequences are somewhat different (see chapter 9, "Model M3"), and the average number of moves tried is reduced from 8 to 6. The time needed to discover all sequences grows, therefore, by a factor of 6 for each extra move.

Digicube displays continuously the search progress as a percentage and, for times longer than one minute, the elapsed time and estimated remaining time in hours and minutes. So it is easy to determine how long a search would take for a given set of conditions: enter your selections, start the search, note the estimated time, and then stop it by pressing X. The longest time displayed is 9999 hours. Note that for some selections the displayed sequences are scrolled so fast that the line with the time values is obscured. You can then estimate the rate of display by noting how many digits at the beginning of the displayed sequences stay constant.

The sequences found and displayed are added at the same time to the output file. Since you usually want to examine the many sequences displayed when the operation ends, the output file is indispensable. If you want to keep these sequences, make sure you copy them to another file before you re-create the output file.

If you plan to experiment with back sequences, there are a few things to keep in mind. You can stop the search at any time by pressing X (with long sequences, it may take a few seconds for the X to take effect). For a given set of conditions selected, there is a minimum sequence length, below which no sequences are found; the larger the number of wrong pieces, the shorter the minimum length. (For models M2 and M3, the minimum sequence length for a given number of wrong pieces is less than for M1.) Also, the greater the sequence length specified, the more sequences are found. For many sets of conditions, sequences exist only for lengths that are an even number. For certain conditions, many minutes elapse between two displayed sequences, while for others hundreds of sequences are displayed per second. In general, if no sequences are found in the first 10% to 20% of a search, you can stop it, since it is highly unlikely that any will be found.

CHAPTER 11

Custom operations

Introduction

The custom operations are performed through run options C1 to C99. These are special functions that, for various reasons, lie outside the range of operations that are possible or convenient in a script or interactively. For example, a simple operation may need neither a script nor interactive data entry; or, conversely, an operation may require many values to be specified for each run. Custom operations also allow a programmer to add new functions without having to modify, or to fully understand, the existing Digicube code.

Custom operations can read scripts in the input file, display results and add them to the output file, and use run options, similarly to the regular operations. The prompts and the related keyboard entries are also added to the output file.

The custom operations currently implemented are described in this chapter. They perform useful functions and, at the same time, serve to demonstrate this concept. These operations work with all three models, but generally, a custom operation can be designed for a specific model. The run option C (with no number) lists these operations as a reminder.

C1 Solve a random position

C1 generates a random position and solves it. This is useful as a demonstration of the solution process. Each time it is executed, a different position is solved. Any run options can be used: M to choose model, D to display the initial position, K to prevent adding lines to the output file, etc. You can stop the solution by pressing X, and this will also end the run.

C2 Solve several random positions

C2 generates a random position and solves it, like C1, but repeats this a number of times. You must answer the following prompts:

Number of positions (1–25000): Enter the required number of repetitions.

Press Enter to continue, X to cancel: Enter will start the operation, X will end the run.

You can stop the repetitions at any time by pressing X, and this will also end the run. As with C1, any run options can be used: M, D, K, etc. D provides a record of the initial positions, in case you find some interesting solutions in the displayed results and wish to study them. C2 is useful as a demonstration of the solution process and, with thousands of repetitions, to provide data for statistical analysis of solutions.

C3 Solve benchmark position

C3 solves a fixed, built-in position. This is useful if you want to compare the solution time on different computers or on the same computer under different conditions (or, if you modified Digicube, to verify whether this has affected the solution time). You select the model with the run option M, and the other run options can also be used. These are the built-in positions used for the three models:

```
M1:  3 6 2 6 1 4 5 2 6   1 2 1 4 2 1 1 3 1   3 5 5 3 3 1 4 5 2
     6 5 6 2 4 1 3 4 6   5 4 4 2 5 6 3 6 2   4 1 4 5 6 3 5 3 2

M2:  4 3 1 2   2 1 4 3   3 1 6 5   2 4 6 5   1 4 2 3   6 5 5 6

M3:  2 3 1 4 4 3   3 1 4 1 1 1   2 2 3 2 1 3   3 4 2 2 4 4
```

The solution times with a 3.4 GHz Intel i5-7500 processor running Windows 10 are, in seconds: M1 2.2, M2 1.7, M3 1.4 (as displayed at the end of the solution). The M1 time is shorter than average, but the M2 and M3 times are much longer than average. The positions were chosen so as to provide a quick way to measure time differences as small as 10%. The times are meaningful, of course, only if no other programs are running in the same core.

Note that the benchmark is accurate only for the particular function performed by C3. Thus, if C3 shows one computer to be 50% faster than another, other positions, other Digicube functions, and other programs may show a somewhat different performance ratio. This is true because other functions may generate a different mix of low-level operations than C3. Nevertheless, C3 provides a quick and simple way to find differences. If you require greater accuracy (still within the domain of Digicube solutions), use C2 with at least a few hundred positions, and measure the total elapsed time.

C4 List back sequences

C4 gives you access to the feature known as back sequences (see chapter 10, “Back sequences”). You select the various search conditions by answering a number of prompts, and Digicube lists the back sequences that fulfill those conditions. Use the run option M to select the model; you may also have to use some of the other run options. The sequences found and displayed are added at the same time to the output file, so you can examine them when the operation ends.

Here are the prompts:

Number of wrong pieces (0–16): Enter the number of wrong pieces in the positions reached by the back sequences, between 0 and 16. For model M2 the prompt is “0–7”; for M3 it is “0–9”. With 0 wrong pieces, *do-nothing* sequences will be found (sequences with the end position identical to the initial position).

The upper limit is due to the fact that the anchor pieces do not change, so they cannot be wrong. Thus, 4 pieces are excluded for M1 and one for M2. M3 has no anchor, but piece 123 is used as reference and cannot be wrong: since a piece that is wrong in one orientation of the pyramid may be correct in another, each end position is checked with the pyramid in the standard orientation (in piece notation, ^123=123). This is also how you must orient a real pyramid after applying the moves, if you want it to match the list of wrong pieces displayed for that back sequence.

Type of wrong pieces (1–3): There are two prompts under this heading. If you selected 0 wrong pieces, the heading and the prompts are omitted.

1 for edges only, 2 for corners only, 3 for both: Enter 1 or 2 to restrict the type of wrong pieces to edge or corner pieces, or 3 to allow both. For model M2, the prompt is just "corners"; for M3, it is "axials" instead of "corners".

1 for flipped in place only, 2 for wrong location only, 3 for both: Enter 1 or 2 to restrict the type of wrong pieces to correct location but flipped, or to wrong location; or enter 3 to allow both types. Note that 3 may find more sequences than the sum of those found through 1 and 2 separately, because 3 may include sets of wrong pieces that are a mix of 1 and 2, and hence not found through either 1 or 2.

Sequence lengths (up to 10 lengths of 1–20): When you only need sequences of a specific length, specify one value, between 1 and 20. But you can also specify several lengths. This is useful with short sequences, which take little time to search. Enter up to 10 values, separated by spaces, each one between 1 and 20. The lengths need not be numerically consecutive or in increasing order. The search is performed for each length in turn, so the lists of sequences appear the same as when the lengths are specified one at a time. Note that the values continuously displayed as elapsed and remaining time are for each length separately.

Display only sequences starting with these (up to 4) moves: Enter a short sequence of one, two, three, or four moves if you want to restrict the results to back sequences that start with these moves (see chapter 10, "Back sequences"). Otherwise enter nothing.

The moves must be a combination of the following values, separated by one or more spaces: 1, –1, 3, –3, 5, –5. Examples: "3", "–1 5", "1 3 5 –3". Note that if the short sequence contains combinations of moves that would cause a back sequence to be ignored (because they do nothing or have the same effect as others), you will get no results at all. Examples of such sequences: "1 –1", "1 –3 3", "1 5 –5 3", "1 1 1", "1 –3 –3". Also, fewer sequences are found starting with –1, –3, or –5 than with 1, 3, or 5 (because the pairs "–1 –1", "–3 –3", and "–5 –5" have the same effect as "1 1", "3 3", and "5 5", so sequences starting with these pairs are ignored). In other words, no back sequences will be listed that would not be listed in a regular, full search.

R to show also the reverse moves: Enter R (lower or upper case) if you want to list also the reverse moves of each back sequence. Entering anything else or nothing is interpreted as No. R will show on a separate line, under each back sequence, the same moves but reversed and listed backwards. This is needed only if you plan to use the back sequences for solutions, since it is their reverse that must be applied to a position.

Note that if you specified as wrong pieces only edge pieces flipped in place, or if you specified 0 wrong pieces, the reverse moves are not really needed: in these cases the back sequences themselves can be used, since they have the same effect as their reverse.

S to start with script, else starts with solved position: Enter S (lower or upper case) if you want to specify a particular initial position (see chapter 10, "Back sequences"). If you enter anything else or nothing, the solved position will be used as the initial position; this is the normal use of back sequences.

To specify a position, you must use a script (see chapter 7, "Scripts"). Typically, the script is simply the operation P with the required position, but it can also contain moves and other operations. It must end with Z (see operation Z), and the current position at that point will be used as the initial position for the C4 search. The position must be valid, and its anchor must be correct (because moves of only three types are used in back sequences), otherwise Digicube will display an error message and end the run. You must use the run option S if the script starts at a label; and the run option I if it is in an input file other than the default one.

Press Enter to continue, X to cancel: Enter will start the operation, X will end the run.

Digicube displays continuously the search progress as a percentage and, for times longer than one minute, the elapsed time and estimated remaining time in hours and minutes. The longest remaining time displayed is 9999 hours (the elapsed time resets to 0 after 5928 hours). You can stop the search at any time by pressing X (with long sequences it may take a few seconds for the X to take effect).

C5 Repeat script

C5 repeats a script a number of times. The script is in the input file and starts with a label, as usual. Use the run option S to identify the label. No label is needed if the script starts at the beginning of the file. (See chapter 7, "Scripts".) The script must end with Z (see operation Z). Any run options can be used. You must answer the following prompts:

Number of repetitions (1–25000): Enter the number of times you want the script to be repeated.

Display cycle number every n cycles (1/10/100/1000, 0 for none): Enter one of these values to choose the frequency with which a number is displayed (on a separate line) during execution to identify the current cycle. You may choose, for example, 1 or 10 if the script is repeated 100 times, and 1000 if repeated 20,000 times. Enter 0 or nothing to omit this feature.

Press Enter to continue, X to cancel: Enter will start the operation, X will end the run.

The run ends when the repetitions end, but you can press X to stop the script at any time. Once stopped, a message is displayed with the choice to end the run or continue with the next element in the script (and hence continue the repetitions).

C5 is useful as an alternative to loops, or in conjunction with loops, especially when you must bypass a restriction. For example, if the limit of 2000 bytes for a loop is a problem, replace the loop with a repeated script, which can have any size. Or, if you need two levels of repetition (a loop within a loop), use a regular loop as the inner loop and a repeated script as the outer loop.

CHAPTER 12

Error messages

The messages are listed alphabetically in each one of the following categories: file access, run options, operations, orientation check, position check, anchor check, fatal errors.

File access

Cannot create —: The operating system reported an error when trying to create the output file indicated (invalid folders path or no free space).

Cannot open —: The operating system reported an error when trying to open the file indicated (file not found or access denied).

Cannot write —: The operating system reported an error when trying to write to the output file indicated.

Error accessing —: The operating system reported an error when trying to read from or write to the file indicated.

Error reading input file: The operating system reported an error when trying to read from the input file (while looking for a label or while reading a script).

Error reading options file: The operating system reported an error when trying to read from the options file (while looking for a label or while reading the options line).

Run options

If you used the option G, the error message may refer to either the original options list or the one found in the options file.

Duplicate option: The option is specified more than once or is used together with a related one (e.g., D D1).

Invalid option: The letter indicated is correct, but something else is wrong in the specification (e.g., D3).

Label #— not found: The label indicated could not be found in the input file.

Options label #— not found: The label indicated could not be found in the options file.

Redirection invalid in options file: The option G cannot be used in the options file.

Too many options: There are more than 15 options, or the options list exceeds 125 characters.

Unidentified option: The letter indicated is not an option.

Operations

Incomplete operation: The character indicated is correct as the beginning of an operation, but the rest is missing (script or interactive).

Invalid 9's: The value used as colors in the operation "^" has partial 9's (script or interactive). If a piece has 9's, all its faces must be 9.

Invalid colors: The value used as colors in the operation "^" is invalid (script or interactive). The valid colors for the current model are the same as the locations in the list of pieces displayed by the operation U, plus the color combinations of flipped pieces. For corner and axial pieces, only 3 of the 6 combinations of colors are valid in a particular location.

Invalid data: Something is wrong with the characters indicated (script or interactive): wrong digit or other character, wrong use of dash or parentheses, etc.

Invalid for current model: The operation indicated cannot be used with the current model, or the value 5 or 6 is used for a move, turn, or color in M3 (script or interactive).

Invalid location: The value used as location in the operation "^", "/", or "\" is invalid (script or interactive). The valid locations for the current model are those in the list of pieces displayed by the operation U.

Invalid memory: The value indicated should be a memory but is not a number 1–99 (script or interactive).

Invalid move: The element indicated should be a move but is not a number 1–6 (script or interactive).

Invalid operation: The characters indicated do not constitute a valid operation (script or interactive).

Invalid repeat count: The value indicated should be a repeat count, but is not a number 1–999999999 (script or interactive).

Invalid run option: The character indicated should be one of the run options that can be set and reset with the operations "+" and "–", but is incorrect (script or interactive).

Invalid turn: The element indicated should be a turn but is not a number 1–6 (script or interactive).

Loop limit exceeded: With the element indicated, the loop exceeds the limit of 2000 bytes. The space requirement is explained in chapter 7, "Scripts". Perhaps the closing "]" is missing and the following elements are seen as part of the loop.

Misplaced "[" or "]": In a script, "[" is used within a loop, or "]" is used outside a loop.

Missing ")": In a script, there was an opening "(" for a turn, but no closing ")".

Missing "]": There was a "[" to start a loop, but no matching "]" to end it (script or interactive).

Missing quotation mark: In a script, there was an opening double quotation mark for a character string, but no closing one.

Operation invalid in loop: These operations cannot be used in a loop: X, Z, P and its variants, PG and its variants.

Too many moves/turns: A sequence in interactive mode, in a loop or not, must not have more than 30 moves and/or turns.

Unexpected end of input file: In a script, the input file ended before all the expected color digits following the operation P or PG (or their variants) were read.

Orientation check

Anchor wrong, cube incorrectly oriented: In M2, the current position is not in the standard orientation (according to the piece in location 246).

Cube incorrectly oriented: In M1, the current position is not in the standard orientation (according to the center pieces).

Goal anchor missing: In M2, the current goal position should be in the standard orientation; but its orientation cannot be determined, because the anchor piece 246 has 9's.

Goal incorrectly oriented: The current goal position is not in the standard orientation; if M1, according to the center pieces; if M2, according to the piece in location 246.

Position check

A position is deemed invalid if it cannot be reached starting from the solved position and using only moves and turns (in Digicube or with a real cube). Thus, a position in Digicube can be invalid only if specified incorrectly or if modified incorrectly with the operation "^", "/", or "\". Valid goal positions with 9's are seen as invalid if verified as regular positions. If the position of a real cube was entered and is invalid, then it was entered incorrectly or the cube itself was invalid (disassembled and then reassembled incorrectly).

The position messages have two parts; for example, "Invalid cube, some edges wrong". The first part identifies the model and the position, as follows:

Invalid cube: M1 or M2, current position.
Invalid pyramid: M3, current position.
Invalid goal: M1, M2, or M3, current goal position.

The second part describes the problem, as follows:

cannot reset position: The position is invalid and the request to reset it cannot be fulfilled. Use the operation V or VG to show what is wrong.

some axials duplicated: Two or more axial pieces have the same colors.

some axials have invalid 9's: One or more axial pieces have partial 9's. If a piece has 9's, all its faces must be 9.

some axials swapped wrongly: One or more pairs of axial pieces are swapped in such a way that the parity check fails. This means that an odd number of swaps took place since the position was valid. Swapping now any pair of axial pieces should solve the problem.

some axials wrong: One or more axial pieces have incorrect colors. (Examples: 112; 132 when in location 123.)

some centers wrong: A color appears in more than one center piece, or the color arrangement of the center pieces is not one of the 24 valid orientations.

some colors wrong: One or more of the color digits is not 1–6 (or 9 in a goal position); or a color digit is used in too many places in the position (caused by a piece having incorrect colors, such as 12 or 133 in a cube).

some corners duplicated: Two or more corner pieces have the same colors.

some corners flipped wrongly: One or more corner pieces are oriented in such a way that the parity check fails. This means that an odd number of corner pieces were flipped since the position was valid, or pairs of pieces were flipped without properly matching the direction, clockwise (cw) or counterclockwise (ccw). Flipping now any corner piece in a specific direction, cw or ccw, should solve the problem.

some corners have invalid 9's: One or more corner pieces have partial 9's. If a piece has 9's, all its faces must be 9.

some corners wrong: One or more corner pieces have incorrect colors. (Examples: 125, 335; 253 when in location 235.)

some edges duplicated: Two or more edge pieces have the same colors.

some edges flipped wrongly: One or more edge pieces are oriented in such a way that the parity check fails. This means that an odd number of edge pieces were flipped since the position was valid. Flipping now any edge piece should solve the problem.

some edges have invalid 9's: One or more edge pieces have partial 9's. If a piece has 9's, all its faces must be 9.

some edges swapped wrongly: In M3, one or more pairs of edge pieces are swapped in such a way that the parity check fails. This means that an odd number of swaps took place since the position was valid. Swapping now any pair of edge pieces should solve the problem.

some edges wrong: One or more edge pieces have incorrect colors. (Examples: 33; 12 in a cube.)

some pieces swapped wrongly: In M1, one or more pairs of edge or corner pieces are swapped in such a way that the parity check fails. This means that an odd number of swaps took place since the position was valid. Swapping now any pair of edge or corner pieces should solve the problem.

Anchor check

Anchor wrong: In M1, the anchor must be solved in the current position (because the operation involves only three move types, and many sequences would not be discovered with a wrong anchor).

Goal anchor wrong: In M1, the anchor must be solved in the current goal position (because the operation involves only three move types, and the current position would never match a goal position with a wrong anchor).

Fatal errors

You should never see these messages. They are displayed when Digicube finds itself in a situation that normally cannot occur. The current run is terminated.

Not solved, unidentified error: The end position reached with the standard solution is not the solved position.

⋆⋆ **Error — ⋆⋆**: A problem was discovered during a solution ("—" is a short word identifying the problem).