Sage Reference Manual: Games

Release 6.6

The Sage Development Team

April 18, 2015

CONTENTS

Sage includes a sophisticated Sudoku solver. It also has a Rubik's cube solver (see Rubik's Cube Group).

CHAPTER

ONE

SUDOKU PUZZLES

This module provides algorithms to solve Sudoku puzzles, plus tools for inputting, converting and displaying various ways of writing a puzzle or its solution(s). Primarily this is accomplished with the `sage.games.sudoku.Sudoku` class, though the legacy top-level `sage.games.sudoku.sudoku()` function is also available.

AUTHORS:

- Tom Boothby (2008/05/02): Exact Cover, Dancing Links algorithm
- Robert Beezer (2009/05/29): Backtracking algorithm, Sudoku class

class `sage.games.sudoku.`**`Sudoku`**(*puzzle*, *verify_input=True*)

Bases: `sage.structure.sage_object.SageObject`

An object representing a Sudoku puzzle. Primarily the purpose is to solve the puzzle, but conversions between formats are also provided.

INPUT:

- **puzzle - the first argument can take one of three forms**
  - list - a Python list with elements of the puzzle in row-major order, where a blank entry is a zero
  - matrix - a square Sage matrix over **Z**
  - string - a string where each character is an entry of the puzzle. For two-digit entries, a = 10, b = 11, etc.
- verify_input - default = `True`, use `False` if you know the input is valid

EXAMPLE:

```
sage: a = Sudoku('5...8..49...5...3..673....115..........2.8.........187....415..3...2...49..5.
sage: print a
+-----+-----+-----+
|5    |  8  |  4 9|
|     |5    |  3  |
|  6 7|3    |    1|
+-----+-----+-----+
|1 5  |     |     |
|     |2   8|     |
|     |     |  1 8|
+-----+-----+-----+
|7    |    4|1 5  |
|  3  |    2|     |
|4 9  |  5  |    3|
+-----+-----+-----+
sage: print next(a.solve())
+-----+-----+-----+
|5 1 3|6 8 7|2 4 9|
```

```
|8 4 9|5 2 1|6 3 7|
|2 6 7|3 4 9|5 8 1|
+-----+-----+-----+
|1 5 8|4 6 3|9 7 2|
|9 7 4|2 1 8|3 6 5|
|3 2 6|7 9 5|4 1 8|
+-----+-----+-----+
|7 8 2|9 3 4|1 5 6|
|6 3 5|1 7 2|8 9 4|
|4 9 1|8 5 6|7 2 3|
+-----+-----+-----+
```

backtrack()

Returns a generator which iterates through all solutions of a Sudoku puzzle.

This function is intended to be called from the `solve()` method when the `algorithm='backtrack'` option is specified. However it may be called directly as a method of an instance of a Sudoku puzzle.

At this point, this method calls `backtrack_all()` which constructs *all* of the solutions as a list. Then the present method just returns the items of the list one at a time. Once Cython supports closures and a yield statement is supported, then the contents of `backtrack_all()` may be subsumed into this method and the `sage.games.sudoku_backtrack` module can be removed.

This routine can have wildly variable performance, with a factor of 4000 observed between the fastest and slowest 9×9 examples tested. Examples designed to perform poorly for naive backtracking, will do poorly (such as `d` below). However, examples meant to be difficult for humans often do very well, with a factor of 5 improvement over the DLX algorithm.

Without dynamically allocating arrays in the Cython version, we have limited this function to 16×16 puzzles. Algorithmic details are in the `sage.games.sudoku_backtrack` module.

EXAMPLES:

This example was reported to be very difficult for human solvers. This algorithm works very fast on it, at about half the time of the DLX solver. [sudoku:escargot]

```
sage: g = Sudoku('1....7.9..3..2...8..96..5....53..9...1..8...26....4...3......1..4......7..
sage: print g
+-----+-----+-----+
|1    |    7|  9  |
|  3  |  2  |    8|
|    9|6    |5    |
+-----+-----+-----+
|    5|3    |9    |
|  1  |  8  |    2|
|6    |    4|     |
+-----+-----+-----+
|3    |     |  1  |
|  4  |     |    7|
|    7|     |3    |
+-----+-----+-----+
sage: print next(g.solve(algorithm='backtrack'))
+-----+-----+-----+
|1 6 2|8 5 7|4 9 3|
|5 3 4|1 2 9|6 7 8|
|7 8 9|6 4 3|5 2 1|
+-----+-----+-----+
|4 7 5|3 1 2|9 8 6|
|9 1 3|5 8 6|7 4 2|
|6 2 8|7 9 4|1 3 5|
```

```
+-----+-----+-----+
|3 5 6|4 7 8|2 1 9|
|2 4 1|9 3 5|8 6 7|
|8 9 7|2 6 1|3 5 4|
+-----+-----+-----+
```

This example has no entries in the top row and a half, and the top row of the solution is `987654321` and therefore a backtracking approach is slow, taking about 750 times as long as the DLX solver. [sudoku:wikipedia]

```
sage: c = Sudoku('..............3.85..1.2.......5.7.....4...1...9.......5......73..2.1......
sage: print c
+-----+-----+-----+
|     |     |     |
|     |    3|  8 5|
|    1|  2  |     |
+-----+-----+-----+
|     |5   7|     |
|    4|     |1    |
|  9  |     |     |
+-----+-----+-----+
|5    |     |  7 3|
|    2|  1  |     |
|     |  4  |    9|
+-----+-----+-----+
sage: print next(c.solve(algorithm='backtrack'))
+-----+-----+-----+
|9 8 7|6 5 4|3 2 1|
|2 4 6|1 7 3|9 8 5|
|3 5 1|9 2 8|7 4 6|
+-----+-----+-----+
|1 2 8|5 3 7|6 9 4|
|6 3 4|8 9 2|1 5 7|
|7 9 5|4 6 1|8 3 2|
+-----+-----+-----+
|5 1 9|2 8 6|4 7 3|
|4 7 2|3 1 9|5 6 8|
|8 6 3|7 4 5|2 1 9|
+-----+-----+-----+
```

Citations

dlx(*count_only=False*)

Returns a generator that iterates through all solutions of a Sudoku puzzle.

INPUT:

- count_only - boolean, default = False. If set to `True` the generator returned as output will simply generate `None` for each solution, so the calling routine can count these.

OUTPUT:

Returns a generator that that iterates over all the solutions.

This function is intended to be called from the `solve()` method with the `algorithm='dlx'` option. However it may be called directly as a method of an instance of a Sudoku puzzle if speed is important and you do not need automatic conversions on the output (or even just want to count solutions without looking at them). In this case, inputting a puzzle as a list, with `verify_input=False` is the fastest way to create a puzzle.

Or if only one solution is needed it can be obtained with one call to `next()`, while the existence of a solution can be tested by catching the `StopIteration` exception with a `try`. Calling this particular method returns solutions as lists, in row-major order. It is up to you to work with this list for your own purposes. If you want fancier formatting tools, use the `solve()` method, which returns a generator that creates `sage.games.sudoku.Sudoku` objects.

EXAMPLES:

A 9×9 known to have one solution. We get the one solution and then check to see if there are more or not.

```
sage: e = Sudoku('4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.
sage: print next(e.dlx())
[4, 1, 7, 3, 6, 9, 8, 2, 5, 6, 3, 2, 1, 5, 8, 9, 4, 7, 9, 5, 8, 7, 2, 4, 3, 1, 6, 8, 2, 5, 4
sage: len(list(e.dlx()))
1
```

A 9×9 puzzle with multiple solutions. Once with actual solutions, once just to count.

```
sage: h = Sudoku('8..6..9.5.............2.31...7318.6.24.....73...........279.1..5...8..36..
sage: len(list(h.dlx()))
5
sage: len(list(h.dlx(count_only=True)))
5
```

A larger puzzle, with multiple solutions, but we just get one.

```
sage: j = Sudoku('....a..69.3....1d.2...8....e.4....b....5..c.......7.......g...f....1.e..2.
sage: print next(j.dlx())
[5, 15, 16, 14, 10, 13, 7, 6, 9, 2, 3, 4, 11, 8, 12, 1, 13, 3, 2, 12, 11, 16, 8, 15, 1, 6, 7
```

The puzzle `h` from above, but purposely made unsolvable with addition in second entry.

```
sage: hbad = Sudoku('82.6..9.5.............2.31...7318.6.24.....73...........279.1..5...8..3
sage: len(list(hbad.dlx()))
0
sage: next(hbad.dlx())
Traceback (most recent call last):
...
StopIteration
```

A stupidly small puzzle to test the lower limits of arbitrary sized input.

```
sage: s = Sudoku('.')
sage: print next(s.solve(algorithm='dlx'))
+-+
|1|
+-+
```

ALGORITHM:

The `DLXCPP` solver finds solutions to the exact-cover problem with a "Dancing Links" backtracking algorithm. Given a $0-1$ matrix, the solver finds a subset of the rows that sums to the all 1's vector. The columns correspond to conditions, or constraints, that must be met by a solution, while the rows correspond to some collection of choices, or decisions. A 1 in a row and column indicates that the choice corresponding to the row meets the condition corresponding to the column.

So here, we code the notion of a Sudoku puzzle, and the hints already present, into such a $0-1$ matrix. Then the `sage.combinat.matrices.dlxcpp.DLXCPP` solver makes the choices for the blank entries.

solve (*algorithm='dlx'*)

Returns a generator object for the solutions of a Sudoku puzzle.

INPUT:

- algorithm - default = `'dlx'`, specify choice of solution algorithm. The two possible algorithms are `'dlx'` and `'backtrack'`.

OUTPUT:

A generator that provides all solutions, as objects of the `Sudoku` class.

Calling `next()` on the returned generator just once will find a solution, presuming it exists, otherwise it will return a `StopIteration` exception. The generator may be used for iteration or wrapping the generator with `list()` will return all of the solutions as a list. Solutions are returned as new objects of the `Sudoku` class, so may be printed or converted using other methods in this class.

Generally, the DLX algorithm is very fast and very consistent. The backtrack algorithm is very variable in its performance, on some occasions markedly faster than DLX but usually slower by a similar factor, with the potential to be orders of magnitude slower. See the docstrings for the `dlx()` and `backtrack_all()` methods for further discussions and examples of performance. Note that the backtrack algorithm is limited to puzzles of size 16×16 or smaller.

EXAMPLES:

This puzzle has 5 solutions, but the first one returned by each algorithm are identical.

```
sage: h = Sudoku('8..6..9.5.............2.31...7318.6.24.....73...........279.1..5...8..36..
sage: h
+-----+-----+-----+
|8    |6    |9   5|
|     |     |     |
|     |  2  |3 1  |
+-----+-----+-----+
|    7|3 1 8|  6  |
|2 4  |     |  7 3|
|     |     |     |
+-----+-----+-----+
|    2|7 9  |1    |
|5    |  8  |  3 6|
|    3|     |     |
+-----+-----+-----+
sage: next(h.solve(algorithm='backtrack'))
+-----+-----+-----+
|8 1 4|6 3 7|9 2 5|
|3 2 5|1 4 9|6 8 7|
|7 9 6|8 2 5|3 1 4|
+-----+-----+-----+
|9 5 7|3 1 8|4 6 2|
|2 4 1|9 5 6|8 7 3|
|6 3 8|2 7 4|5 9 1|
+-----+-----+-----+
|4 6 2|7 9 3|1 5 8|
|5 7 9|4 8 1|2 3 6|
|1 8 3|5 6 2|7 4 9|
+-----+-----+-----+
sage: next(h.solve(algorithm='dlx'))
+-----+-----+-----+
|8 1 4|6 3 7|9 2 5|
|3 2 5|1 4 9|6 8 7|
|7 9 6|8 2 5|3 1 4|
```

```
+-----+-----+-----+
|9 5 7|3 1 8|4 6 2|
|2 4 1|9 5 6|8 7 3|
|6 3 8|2 7 4|5 9 1|
+-----+-----+-----+
|4 6 2|7 9 3|1 5 8|
|5 7 9|4 8 1|2 3 6|
|1 8 3|5 6 2|7 4 9|
+-----+-----+-----+
```

Gordon Royle maintains a list of 48072 Sudoku puzzles that each has a unique solution and exactly 17 “hints” (initially filled boxes). At this writing (May 2009) there is no known 16-hint puzzle with exactly one solution. [sudoku:royle] This puzzle is number 3000 in his database. We solve it twice.

```
sage: b = Sudoku('8..6..9.5.............2.31...7318.6.24.....73...........279.1..5...8..36..
sage: next(b.solve(algorithm='dlx')) == next(b.solve(algorithm='backtrack'))
True
```

These are the first 10 puzzles in a list of “Top 95” puzzles, [sudoku:top95] which we use to show that the two available algorithms obtain the same solution for each.

```
sage: top =['4.....8.5.3..........7......2.....6.....8.4......1.......6.3.7.5..2.....1.4...
            '52...6.........7.13...........4..8..6......5...........418.........3..2...87..
            '6.....8.3.4.7.................5.4.7.3..2.....1.6.......2.....5.....8.6......1.
            '48.3............71.2.......7.5....6....2..8.............1.76...3.....4......5.
            '....14....3....2...7..........9...3.6.1.............8.2.....1.4....5.6.....7.8
            '......52..8.4......3...9...5.1...6..2..7........3.....6...1..........7.4......
            '6.2.5.........3.4..........43...8....1....2........7..5..27...........81...6..
            '.524.........7.1..............8.2...3.....6...9.5.....1.6.3...........897.....
            '6.2.5.........4.3..........43...8....1....2........7..5..27...........81...6..
            '.923.........8.1...........1.7.4...........658.........6.5.2...4.....7.....9..
sage: p = [Sudoku(top[i]) for i in range(10)]
sage: verify = [next(p[i].solve(algorithm='dlx')) == next(p[i].solve(algorithm='backtrack'))
sage: verify == [True]*10
True
```

TESTS:

A 25×25 puzzle that the backtrack algorithm is not equipped to handle. Since `solve` returns a generator this test will not go boom until we ask for a solution with `next`.

```
sage: too_big = Sudoku([0]*625)
sage: next(too_big.solve(algorithm='backtrack'))
Traceback (most recent call last):
...
ValueError: The Sudoku backtrack algorithm is limited to puzzles of size 16 or smaller.
```

An attempt to use a non-existent algorithm.

```
sage: next(Sudoku([0]).solve(algorithm='bogus'))
Traceback (most recent call last):
...
NotImplementedError: bogus is not an algorithm for Sudoku puzzles
```

Citations

to_ascii()

Constructs an ASCII-art version of a Sudoku puzzle. This is a modified version of the ASCII version of a subdivided matrix.

EXAMPLE:

```
sage: s = Sudoku('.4..32....14..3.')
sage: print s.to_ascii()
+---+---+
|  4|   |
|3 2|   |
+---+---+
|   |1 4|
|   |3  |
+---+---+
sage: s.to_ascii()
'+---+---+\n|  4|   |\n|3 2|   |\n+---+---+\n|   |1 4|\n|   |3  |\n+---+---+'
```

to_latex()

Creates a string of LaTeX code representing a Sudoku puzzle or solution.

EXAMPLE:

```
sage: s = Sudoku('.4..32....14..3.')
sage: print s.to_latex()
\begin{array}{|*{2}{*{2}{r}|}}\hline
&4& & \\
3&2& & \\\hline
& &1&4\\
& &3& \\\hline
\end{array}
```

TEST:

```
sage: s = Sudoku('.4..32....14..3.')
sage: s.to_latex()
'\\begin{array}{|*{2}{*{2}{r}|}}\\hline\n &4& & \\\\\n3&2& & \\\\\\hline\n & &1&4\\\\\n & &3
```

to_list()

Constructs a list representing a Sudoku puzzle, in row-major order.

EXAMPLE:

```
sage: s = Sudoku('1.......2.9.4...5...6...7...5.9.3.......7.......85..4.7.....6...3...9.8...
sage: print s.to_list()
[1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 9, 0, 4, 0, 0, 0, 5, 0, 0, 0, 6, 0, 0, 0, 7, 0, 0, 0, 5, 0, 9
```

TEST:

This tests the input and output of Sudoku puzzles as lists.

```
sage: alist = [0, 4, 0, 0, 3, 2, 0, 0, 0, 0, 1, 4, 0, 0, 3, 0]
sage: alist == Sudoku(alist).to_list()
True
```

to_matrix()

Constructs a Sage matrix over $\mathbf{Z}$ representing a Sudoku puzzle.

EXAMPLES:

```
sage: s = Sudoku('.4..32....14..3.')
sage: s.to_matrix()
[0 4 0 0]
[3 2 0 0]
```

```
[0 0 1 4]
[0 0 3 0]
```

TEST:

This tests the input and output of Sudoku puzzles as matrices over **Z**.

```
sage: g = matrix(ZZ, 9, 9, [ [1,0,0,0,0,7,0,9,0], [0,3,0,0,2,0,0,0,8], [0,0,9,6,0,0,5,0,0],
sage: g == Sudoku(g).to_matrix()
True
```

to_string()

Constructs a string representing a Sudoku puzzle.

Blank entries are represented as periods, single digits are not converted and two digit entries are converted to lower-case letters where `10 = a`, `11 = b`, etc. This scheme limits puzzles to at most 36 symbols.

EXAMPLE:

```
sage: b = matrix(ZZ, 9, 9, [ [0,0,0,0,1,0,9,0,0], [8,0,0,4,0,0,0,0,0], [2,0,0,0,0,0,0,0,0],
sage: Sudoku(b).to_string()
'....1.9..8..4.....2.........7..3..........2.4.......58.6....13.7..2........8.....'
```

TESTS:

This tests the conversion of alphabetic characters as well as the input and output of Sudoku puzzles as strings.

```
sage: j = Sudoku([0, 0, 0, 0, 10, 0, 0, 6, 9, 0, 3, 0, 0, 0, 0, 1, 13, 0, 2, 0, 0, 0, 8, 0,
sage: st = j.to_string()
sage: st
'....a..69.3....1d.2...8....e.4....b....5..c.......7.......g...f....1.e..2.b.8..3.......4.d.
sage: st == Sudoku(st).to_string()
True
```

A 49×49 puzzle with all entries equal to 40, which doesn't convert to a letter.

```
sage: empty = [40]*2401
sage: Sudoku(empty).to_string()
Traceback (most recent call last):
...
ValueError: Sudoku string representation is only valid for puzzles of size 36 or smaller
```

sage.games.sudoku.**sudoku**(*m*)

Solves Sudoku puzzles described by matrices.

INPUT:

- `m` - a square Sage matrix over **Z**, where zeros are blank entries

OUTPUT:

A Sage matrix over **Z** containing the first solution found, otherwise `None`.

This function matches the behavior of the prior Sudoku solver and is included only to replicate that behavior. It could be safely deprecated, since all of its functionality is included in the `Sudoku` class.

EXAMPLE:

An example that was used in previous doctests.

```
sage: A = matrix(ZZ,9,[5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0, 0,3,0, 0,6,7, 3,0,0, 0,0,1, 1,5,0, 0,0
sage: A
[5 0 0 0 8 0 0 4 9]
```

```
[0 0 0 5 0 0 0 3 0]
[0 6 7 3 0 0 0 0 1]
[1 5 0 0 0 0 0 0 0]
[0 0 0 2 0 8 0 0 0]
[0 0 0 0 0 0 0 1 8]
[7 0 0 0 0 4 1 5 0]
[0 3 0 0 0 2 0 0 0]
[4 9 0 0 5 0 0 0 3]
sage: sudoku(A)
[5 1 3 6 8 7 2 4 9]
[8 4 9 5 2 1 6 3 7]
[2 6 7 3 4 9 5 8 1]
[1 5 8 4 6 3 9 7 2]
[9 7 4 2 1 8 3 6 5]
[3 2 6 7 9 5 4 1 8]
[7 8 2 9 3 4 1 5 6]
[6 3 5 1 7 2 8 9 4]
[4 9 1 8 5 6 7 2 3]
```

Using inputs that are possible with the `Sudoku` class, other than a matrix, will cause an error.

```
sage: sudoku('.4..32....14..3.')
Traceback (most recent call last):
...
ValueError: sudoku function expects puzzle to be a matrix, perhaps use the Sudoku class
```

CHAPTER

TWO

FAMILY GAMES AMERICA'S QUANTUMINO SOLVER

This module allows to solve the Quantumino puzzle made by Family Games America (see also this video on Youtube). This puzzle was left at the dinner room of the Laboratoire de Combinatoire Informatique Mathematique in Montreal by Franco Saliola during winter 2011.

The solution uses the dancing links code which is in Sage and is based on the more general code available in the module `sage.combinat.tiling`. Dancing links were originally introduced by Donald Knuth in 2000 (arXiv:cs/0011047). In particular, Knuth used dancing links to solve tilings of a region by 2D pentaminos. Here we extend the method for 3D pentaminos.

This module defines two classes :

- `sage.games.quantumino.QuantuminoState` class, to represent a state of the Quantumino game, i.e. a solution or a partial solution.
- `sage.games.quantumino.QuantuminoSolver` class, to find, enumerate and count the number of solutions of the Quantumino game where one of the piece is put aside.

AUTHOR:

- Sebastien Labbe, April 28th, 2011

DESCRIPTION (from [1]):

> " Pentamino games have been taken to a whole different level; a 3-D level, with this colorful creation! Using the original pentamino arrangements of 5 connected squares which date from 1907, players are encouraged to "think inside the box" as they try to fit 16 of the 17 3-D pentamino pieces inside the playing perimeters. Remove a different piece each time you play for an entirely new challenge! Thousands of solutions to be found! Quantumino hands-on educational tool where players learn how shapes can be transformed or arranged into predefined shapes and spaces. Includes: 1 wooden frame, 17 wooden blocks, instruction booklet. Age: 8+ "

EXAMPLES:

Here are the 17 wooden blocks of the Quantumino puzzle numbered from 0 to 16 in the following 3d picture. They will show up in 3D in your default (=Jmol) viewer:

```
sage: from sage.games.quantumino import show_pentaminos
sage: show_pentaminos()
Graphics3d Object
```

To solve the puzzle where the pentamino numbered 12 is put aside:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: s = next(QuantuminoSolver(12).solve())          # long time (10 s)
sage: s                                               # long time (<1s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (2, 1, 1)], Color: blue
```

```
sage: s.show3d()                                    # long time (<1s)
Graphics3d Object
```

To remove the frame:

```
sage: s.show3d().show(frame=False)                  # long time (<1s)
```

To solve the puzzle where the pentamino numbered 7 is put aside:

```
sage: s = next(QuantuminoSolver(7).solve())         # long time (10 s)
sage: s                                             # long time (<1s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
sage: s.show3d()                                    # long time (<1s)
Graphics3d Object
```

The solution is iterable. This may be used to explicitly list the positions of each pentamino:

```
sage: for p in s: p                                 # long time (<1s)
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
Polyomino: [(0, 0, 1), (0, 1, 0), (0, 1, 1), (0, 2, 1), (1, 2, 1)], Color: deeppink
Polyomino: [(0, 2, 0), (0, 3, 0), (0, 4, 0), (1, 4, 0), (1, 4, 1)], Color: green
Polyomino: [(0, 3, 1), (1, 3, 1), (2, 2, 0), (2, 2, 1), (2, 3, 1)], Color: green
Polyomino: [(1, 3, 0), (2, 3, 0), (2, 4, 0), (2, 4, 1), (3, 4, 0)], Color: red
Polyomino: [(1, 0, 1), (2, 0, 1), (2, 1, 0), (2, 1, 1), (3, 1, 1)], Color: red
Polyomino: [(2, 0, 0), (3, 0, 0), (3, 0, 1), (3, 1, 0), (4, 0, 0)], Color: gray
Polyomino: [(3, 2, 0), (4, 0, 1), (4, 1, 0), (4, 1, 1), (4, 2, 0)], Color: purple
Polyomino: [(3, 2, 1), (3, 3, 0), (3, 3, 1), (4, 2, 1), (4, 3, 1)], Color: yellow
Polyomino: [(3, 4, 1), (3, 5, 1), (4, 3, 0), (4, 4, 0), (4, 4, 1)], Color: blue
Polyomino: [(0, 4, 1), (0, 5, 0), (0, 5, 1), (0, 6, 1), (1, 5, 0)], Color: midnightblue
Polyomino: [(0, 6, 0), (0, 7, 0), (0, 7, 1), (1, 7, 0), (2, 7, 0)], Color: darkblue
Polyomino: [(1, 7, 1), (2, 6, 0), (2, 6, 1), (2, 7, 1), (3, 6, 0)], Color: blue
Polyomino: [(1, 5, 1), (1, 6, 0), (1, 6, 1), (2, 5, 0), (2, 5, 1)], Color: yellow
Polyomino: [(3, 6, 1), (3, 7, 0), (3, 7, 1), (4, 5, 1), (4, 6, 1)], Color: purple
Polyomino: [(3, 5, 0), (4, 5, 0), (4, 6, 0), (4, 7, 0), (4, 7, 1)], Color: orange
```

To get all the solutions, use the iterator returned by the `solve` method. Note that finding the first solution is the most time consuming because it needs to create the complete data to describe the problem:

```
sage: it = QuantuminoSolver(7).solve()
sage: next(it)                                   # not tested (10s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
sage: next(it)                                   # not tested (0.001s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
sage: next(it)                                   # not tested (0.001s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
```

To get the solution inside other boxes:

```
sage: s = next(QuantuminoSolver(7, box=(4,4,5)).solve())         # not tested (2s)
sage: s.show3d()                                                 # not tested (<1s)
```

```
sage: s = next(QuantuminoSolver(7, box=(2,2,20)).solve())        # not tested (1s)
sage: s.show3d()                                                 # not tested (<1s)
```

If there are no solution, a StopIteration error is raised:

```
sage: next(QuantuminoSolver(7, box=(3,3,3)).solve())
Traceback (most recent call last):
...
StopIteration
```

The implementation allows a lot of introspection. From the `TilingSolver` object, it is possible to retrieve the rows that are passed to the DLX solver and count them. It is also possible to get an instance of the DLX solver to play with it:

```
sage: q = QuantuminoSolver(0)
sage: T = q.tiling_solver()
sage: T
Tiling solver of 16 pieces into the box (5, 8, 2)
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
sage: rows = T.rows()                           # not tested (10 s)
sage: len(rows)                                 # not tested (but fast)
5484
sage: x = T.dlx_solver()                        # long time (10 s)
sage: x                                         # long time (fast)
<sage.combinat.matrices.dancing_links.dancing_linksWrapper object at ...>
```

REFERENCES:

- [1] Family Games America's Quantumino
- [2] Quantumino - How to Play on Youtube
- [3] Knuth, Donald (2000). "Dancing links". arXiv:cs/0011047.

class `sage.games.quantumino.`**`QuantuminoSolver`**(*aside*, *box=(5, 8, 2)*)

Bases: `sage.structure.sage_object.SageObject`

Return the Quantumino solver for the given box where one of the pentamino is put aside.

INPUT:

- `aside` - integer, from 0 to 16, the aside pentamino
- `box` - tuple of size three (optional, default: `(5,8,2)`), size of the box

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: QuantuminoSolver(9)
Quantumino solver for the box (5, 8, 2)
Aside pentamino number: 9
sage: QuantuminoSolver(12, box=(5,4,4))
Quantumino solver for the box (5, 4, 4)
Aside pentamino number: 12
```

`number_of_solutions`()

Return the number of solutions.

OUTPUT:

integer

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: QuantuminoSolver(4, box=(3,2,2)).number_of_solutions()
0
```

This computation takes several days:

```
sage: QuantuminoSolver(0).number_of_solutions()                # not tested
??? hundreds of millions ???
```

solve (*partial=None*)

Return an iterator over the solutions where one of the pentamino is put aside.

INPUT:

- `partial` - string (optional, default: `None`), whether to include partial (incomplete) solutions. It can be one of the following:
  - `None` - include only complete solution
  - `'common'` - common part between two consecutive solutions
  - `'incremental'` - one piece change at a time

OUTPUT:

iterator of QuantuminoState

EXAMPLES:

Get one solution:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: s = next(QuantuminoSolver(8).solve())         # long time (9s)
sage: s                                             # long time (fast)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 0)], Color: yellow
sage: s.show3d()                                    # long time (< 1s)
Graphics3d Object
```

The explicit solution:

```
sage: for p in s: p                                 # long time (fast)
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
Polyomino: [(0, 0, 1), (0, 1, 0), (0, 1, 1), (0, 2, 1), (1, 2, 1)], Color: deeppink
Polyomino: [(0, 2, 0), (0, 3, 0), (0, 4, 0), (1, 4, 0), (1, 4, 1)], Color: green
Polyomino: [(0, 3, 1), (1, 3, 1), (2, 2, 0), (2, 2, 1), (2, 3, 1)], Color: green
Polyomino: [(1, 3, 0), (2, 3, 0), (2, 4, 0), (2, 4, 1), (3, 4, 0)], Color: red
Polyomino: [(1, 0, 1), (2, 0, 0), (2, 0, 1), (2, 1, 0), (3, 0, 1)], Color: midnightblue
Polyomino: [(0, 4, 1), (0, 5, 0), (0, 5, 1), (0, 6, 0), (1, 5, 0)], Color: red
Polyomino: [(2, 1, 1), (3, 0, 0), (3, 1, 0), (3, 1, 1), (4, 0, 0)], Color: blue
Polyomino: [(3, 2, 0), (4, 0, 1), (4, 1, 0), (4, 1, 1), (4, 2, 0)], Color: purple
Polyomino: [(3, 2, 1), (3, 3, 0), (4, 2, 1), (4, 3, 0), (4, 3, 1)], Color: yellow
Polyomino: [(3, 3, 1), (3, 4, 1), (4, 4, 0), (4, 4, 1), (4, 5, 0)], Color: blue
Polyomino: [(0, 6, 1), (0, 7, 0), (0, 7, 1), (1, 5, 1), (1, 6, 1)], Color: purple
Polyomino: [(1, 6, 0), (1, 7, 0), (1, 7, 1), (2, 7, 0), (3, 7, 0)], Color: darkblue
Polyomino: [(2, 5, 0), (2, 6, 0), (3, 6, 0), (4, 6, 0), (4, 6, 1)], Color: orange
Polyomino: [(2, 5, 1), (3, 5, 0), (3, 5, 1), (3, 6, 1), (4, 5, 1)], Color: gray
Polyomino: [(2, 6, 1), (2, 7, 1), (3, 7, 1), (4, 7, 0), (4, 7, 1)], Color: orange
```

Enumerate the solutions:

```
sage: it = QuantuminoSolver(0).solve()
sage: next(it)                                            # not tested
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
sage: next(it)                                            # not tested
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
```

With the partial solutions included, one can see the evolution between consecutive solutions (an animation would be better):

```
sage: it = QuantuminoSolver(0).solve(partial='common')
sage: next(it).show3d()               # not tested (2s)
sage: next(it).show3d()               # not tested (< 1s)
sage: next(it).show3d()               # not tested (< 1s)
```

Generalizations of the game inside different boxes:

```
sage: next(QuantuminoSolver(7, (4,4,5)).solve())        # long time (2s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
sage: next(QuantuminoSolver(7, (2,2,20)).solve())       # long time (1s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (0, 2, 1), (1, 0, 0)], Color: orange
sage: next(QuantuminoSolver(3, (2,2,20)).solve())       # long time (1s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (1, 0, 0), (1, 0, 1)], Color: green
```

If the volume of the box is not 80, there is no solution:

```
sage: next(QuantuminoSolver(7, box=(3,3,9)).solve())
Traceback (most recent call last):
...
StopIteration
```

If the box is too small, there is no solution:

```
sage: next(QuantuminoSolver(4, box=(40,2,1)).solve())
Traceback (most recent call last):
...
StopIteration
```

tiling_solver()

Return the Tiling solver of the Quantumino Game where one of the pentamino is put aside.

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: QuantuminoSolver(0).tiling_solver()
Tiling solver of 16 pieces into the box (5, 8, 2)
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
sage: QuantuminoSolver(14).tiling_solver()
Tiling solver of 16 pieces into the box (5, 8, 2)
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
sage: QuantuminoSolver(14, box=(5,4,4)).tiling_solver()
Tiling solver of 16 pieces into the box (5, 4, 4)
```

```
Rotation allowed: True
Reflection allowed: False
Reusing pieces allowed: False
```

class sage.games.quantumino.**QuantuminoState**(*pentos*, *aside*)

Bases: sage.structure.sage_object.SageObject

A state of the Quantumino puzzle.

Used to represent an solution or a partial solution of the Quantumino puzzle.

INPUT:

- pentos - list of 16 3d pentamino representing the (partial) solution
- aside - 3d polyomino, the unused 3D pentamino

EXAMPLES:

```
sage: from sage.games.quantumino import pentaminos, QuantuminoState
sage: p = pentaminos[0]
sage: q = pentaminos[5]
sage: r = pentaminos[11]
sage: S = QuantuminoState([p,q], r)
sage: S
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 0, 1), (1, 1, 0), (1, 2, 0)], Color: darkblue

sage: from sage.games.quantumino import QuantuminoSolver
sage: next(QuantuminoSolver(3).solve())      # not tested (1.5s)
Quantumino state where the following pentamino is put aside :
Polyomino: [(0, 0, 0), (0, 1, 0), (0, 2, 0), (1, 0, 0), (1, 0, 1)], Color: green
```

list()

Return the list of 3d polyomino making the solution.

EXAMPLES:

```
sage: from sage.games.quantumino import pentaminos, QuantuminoState
sage: p = pentaminos[0]
sage: q = pentaminos[5]
sage: r = pentaminos[11]
sage: S = QuantuminoState([p,q], r)
sage: L = S.list()
sage: L[0]
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 1, 0), (1, 1, 1), (1, 2, 0)], Color: deeppink
sage: L[1]
Polyomino: [(0, 0, 0), (1, 0, 0), (1, 0, 1), (1, 1, 0), (2, 0, 1)], Color: red
```

show3d(*size=0.85*)

Return the solution as a 3D Graphic object.

OUTPUT:

3D Graphic Object

EXAMPLES:

```
sage: from sage.games.quantumino import QuantuminoSolver
sage: s = next(QuantuminoSolver(0).solve())    # not tested (1.5s)
sage: G = s.show3d()                            # not tested (<1s)
```

```
sage: type(G)                                    # not tested
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

To remove the frame:

```
sage: G.show(frame=False) # not tested
```

To see the solution with Tachyon viewer:

```
sage: G.show(viewer='tachyon', frame=False) # not tested
```

sage.games.quantumino.**show_pentaminos**(*box=(5, 8, 2)*)

Show the 17 3-D pentaminos included in the game and the $5 \times 8 \times 2$ box where 16 of them must fit.

INPUT:

- `box` - tuple of size three (optional, default: `(5,8,2)`), size of the box

OUTPUT:

3D Graphic object

EXAMPLES:

```
sage: from sage.games.quantumino import show_pentaminos
sage: show_pentaminos()     # not tested (1s)
```

To remove the frame do:

```
sage: show_pentaminos().show(frame=False)   # not tested (1s)
```

CHAPTER

THREE

INDICES AND TABLES

- Index
- Module Index
- Search Page

[sudoku:escargot] "Al Escargot", due to Arto Inkala, http://timemaker.blogspot.com/2006/12/ai-escargot-vwv.html

[sudoku:wikipedia] "Near worst case", Wikipedia: "Algorithmics of sudoku", http://en.wikipedia.org/wiki/Algorithmics_of_sudoku

[sudoku:top95] "95 Hard Puzzles", http://magictour.free.fr/top95, or http://norvig.com/top95.txt

[sudoku:royle] Gordon Royle, "Minimum Sudoku", http://people.csse.uwa.edu.au/gordon/sudokumin.php

PYTHON MODULE INDEX

g