In general, the advanced solver algorithm for the killer sudoku is based on Dancing Links, proposed by Knuth(2000).

1. Representation of Killer Sudoku Grid

As a killer sudoku, the input has no values in any grids at the beginning, so it is not useful to generate a board, and therefore, a HashMap<Integer,Cell>, meaning position and cell, is used instead of an int array. Each cell has a unique position, an integer, calculated by multiplying the size of the board with row and then plus column. In order to save cage information, an ArrayList of Cage class is created by the input, including an ArrayList of Cell class and the sum of these cells should be.

When constructing Killer Sudoku Grid, each cage will recursively find all possible values that sum to the total using findElements method and save the results in an ArrayList of HashSet<Integer>. After that, each cell will get these information from the cage they are in and generate DancingLinkCell class, which is the key for constructing cage constraints in dancing links. And each cell has an ArrayList of DancingLinkCell.

2. Consider Killer Sudoku as an exact cover problem - DancingLinkCell class

Take a killer sudoku with values from 1 to 9 as an example, no matter how cages change, the sum of all cages remains the same, which is the sum of all the grids - (1+2+...+9)*9 = 405 in this case. So if there are 405 columns for this sum, all the columns should be covered to solve the Killer Sudoku. For each cage, it will take their total columns, and therefore, cover all cages will cover 405 columns.

For example, a cage with a total of 17 will take 17 columns of cage constraints columns. And if this cage has two cells, there will be only one possible solution as found when constructing Killer Sudoku Grid, which is 8 and 9. The matrix for this cage would be like matrix 1(See Appendix). Although the first row looks exactly the same as the third row and the second row looks exactly the same as the fourth row, they are different regarding their previous columns, especially cell columns. When selecting the first row of this matrix, the first cell will have a value of 8, and the other cell cannot have a value of 8(third row) because these columns have already been covered. And in order to cover all these 17 columns, the fourth row must be selected after the first row has been selected.

A more complicated example would be a cage with four cells and a sum of 18. The matrix for one cell(As all cells in the same cage have the same matrix) would be like matrix 2(See Appendix). One possible cover solution is marked in red, which is 3+4+5+6 =18. This matrix can be considered as values with startPoints: [Value: 1, StartPoint: 0, Value: 2, StartPoint: 1, Value: 6, StartPoint: 3, Value: 9, StartPoint: 9, Value: 7, StartPoint: 3, Value: 8, StartPoint: 10, Value: 3, StartPoint: 1, Value: 5, StartPoint: 4, Value: 6, StartPoint: 4, Value: 4, StartPoint: 1, Value: 5, StartPoint: 5, Value: 6, StartPoint: 5, Value: 7, StartPoint: 11, Value: 2, StartPoint: 0, Value: 3, StartPoint: 2, Value: 4, StartPoint: 5, Value: 4, StartPoint: 2, Value: 5, StartPoint: 6, Value: 3, StartPoint: 0, Value: 4, StartPoint: 3, Value: 5, StartPoint: 7, Value: 6, StartPoint: 12], which is the DancingLinkCell class - value with a start Point. And the same value with different startPoints is not the same because they are not in the same sum solution. Start from the first cage, add these matrices to the right of the matrix for regular Sudoku (each cage matrix starts after the previous one) to create the dancing links for the Killer Sudoku.

3. Solve the dancing links deterministic

Find the column in the header list with minimum rows of 1, if there is any column with all 0, retrieve to find another solution. Cover the column and all the related rows. Call the solve method again, if false, uncover the column and all the related rows.

4. Comparison of the backtracking and advanced algorithms

The backtracking algorithm is improved by selecting values from the possible values for the cell instead of all the values. However, it will still take more time since there are no strong connections between cages, especially cages in the same column. The worst time complexity for the backtracking algorithm before improvement is $O(n^m)$ where n is the size of input values and m is the total numbers of the grids(the square of the size of the board). After the improvement, the worst time complexity for the backtracking algorithm is hard to compute but less than $O(n^m)$, depending on the possible values of cages. For each cages, the worst time complexity is $O(k * m!)$ where m is the total number of grids the cage has and k is the number of possible solutions($k <= C_n^m$). And therefore the worst time complexity would depend on the cage with most possible solutions with many grids inside, the more these cages occur, the more time complexity is close to the previous one.

However, the time complexity for the advanced algorithm is the same as the dancing link algorithm. To search for the column with the minimum rows, the time complexity is $O(N)$, where N is the columns that are left in the header list, and it decreases recursively till 0. The other time needed is the same mentioned by Knuth(2000), ‘the running time of algorithm DLX is essentially proportional to the number of times it applies operation (1) to remove an object from a list; this is also the number of times it applies operation (2) to unremove an object.’ And as I start from the column with minimum rows each time, the time complexity would be optimal.

REFERENCE

Knuth, D. (2000). Dancing links. ArXiv.org, ArXiv.org, Nov 15, 2000.

Appendix

Matrix 1:

1	1	1	1	1	1	1	1									
								1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1									
								1	1	1	1	1	1	1	1	1

Matrix 2:

1																	
1	1																
1	1	1															
	1	1															
	1	1	1														
	1	1	1	1													
		1	1	1													
		1	1	1	1												
			1	1	1	1											
			1	1	1	1	1										
			1	1	1	1	1	1									
			1	1	1	1	1	1	1								
				1	1	1	1	1									
				1	1	1	1	1	1								
					1	1	1	1									
					1	1	1	1	1								
					1	1	1	1	1	1							
							1	1	1	1	1						
									1	1	1	1	1	1	1	1	1
										1	1	1	1	1	1	1	1

											1	1	1	1	1	1	1
												1	1	1	1	1	1