Understanding the Dancing Links algorithm:

The dancing links algorithm is an algorithm designed by Donald Knuth to solve a set of problems known as "exact cover" problems. In an exact cover problem, you are given a matrix that is full of 1's and 0s (a binary matrix). The goal is to chose the right set of rows in the matrix so that all of the columns in the combined set have a 1 in them.

An example:

A	0	1	1	0	0
B	1	0	0	0	0
C	0	0	0	0	1
D	0	1	0	0	0
E	0	0	0	1	0

Given the table above, the solution is the set that contains rows A, B, C, and E. If you examine those four rows, the combination of the 1's in all of the rows gives you a 1 in every column.

The concept of a Sudoku game board can be applied to such a problem but this will not be explained here. There are plenty of websites that describe how to map a sudoku game into an exact cover matrix.

It is important to note that the dancing links algorithm that Donald Knuth made famous is not designed for sudoku, but instead for the *exact cover* problem. The dancing links algorithm solves the exact cover problem (and sudoku puzzles) with a *brute force* calculation attempt that involves backtracking. This seems to be the only way to properly solve exact cover type problems, of which Sudoku is no different. The difference though is that the dancing links algorithm is a very CPU efficient manner of solving these problems.

Links:

There are many helpful links online that will aid in understanding how to map the Sudoku game to an exact cover matrix and that will give an overview in understanding dancing links and exact cover. Here are some links:

http://www.osix.net/modules/article/?id=792
http://en.wikipedia.org/wiki/Algorithmics_of_sudoku
http://en.wikipedia.org/wiki/Exact_cover#Sudoku
http://en.wikipedia.org/wiki/Dancing_Links
http://www.ocf.berkeley.edu/~jchu/publicportal/sudoku/sudoku.paper.html

Basic Rules:

There are some important basic rules that should be understand about exact cover and dancing links:

1. As the algorithm of dancing links proceeds, columns in the matrix are eliminated. As such, as soon as there are no columns left, you have solved the problem.
2. All columns that are available must have at least one 1
3. If a column exists that is empty (has no 1's), then an invalid state has been reached in the matrix

and you must backtrack.

Algorithm Info:

The best approach to understand the algorithm is to read Knuth's paper on it. This is a bit difficult though since it contains dense pseudo code written with mathematical symbology. This document will not explain the entire algorithm but instead will provide pointers to make it easier to digest the information in Knuth's paper.

The general process to the algorithm is:

1. Choose a column of the matrix. This can be done sequentially (i.e., go from left to right and pick all of the columns in order as the algorithm Progresses). This can be done randomly also. The most efficient way (in terms of fewer backtracking sessions required) is to choose the column that has the fewest 1's in it.
2. Choose 1 of the 'filled' rows in the column that you have selected. By "filled" we mean an element in the column that contains a 1.
3. Add the chosen row to your solution. (Your solution is merely a list of rows you are keeping that will allow you to solve the problem).
3. Cover the column that you selected.
4. Cover the other columns in the matrix that are filled with the same row that you have chosen in step 2.

The act of covering is defined as:

1. Remove the column from the linked group of column headers in the matrix. This effectively removes the column from the matrix and prevents it from being selected again for covering in the future. You can only cover a column once so that is why this must be done.

    2. Delete every filled row in the column that you are covering (this includes the row that you have chosen to be part of your solution).

It is important to note that the act of "deleting" rows does not disturb the horizontal links (left-right) links between elements in a row. It simply removes the data nodes in the row from the columns that contained them before. It is also important to note that if you look closely at Knuth's examples, when you "delete" a row during the process of covering, you do not "delete" the element contained in the column you are covering. So, if in a column of your choice there is 1 filled row (in other words the column has one 1), you do not delete the filled row in the column you are covering. This can best be explained via an example down below.

Backtracking note:

The reasons for backtracking should be explained. If you choose a column to cover that has only one 1, you will not have to backtrack. When you choose a column that has more than one 1, there is a possibility you will have to backtrack though. This is because when you choose a column to cover you also have to pick a row to add to your solution. The row you are adding to your solution effects which other columns will be covered in the matrix after that which can lead to an invalid state of the matrix (a column with no data). Also, by picking the wrong row your list of solution rows will be incorrect as well.

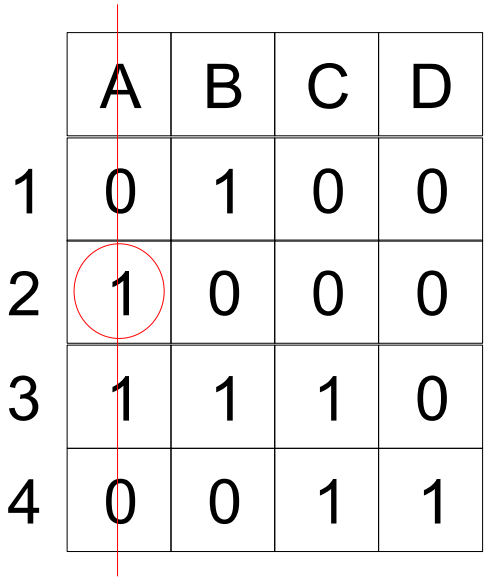
So, when you backtrack you remove a row from your solution and uncover the columns that you have covered.

An example:

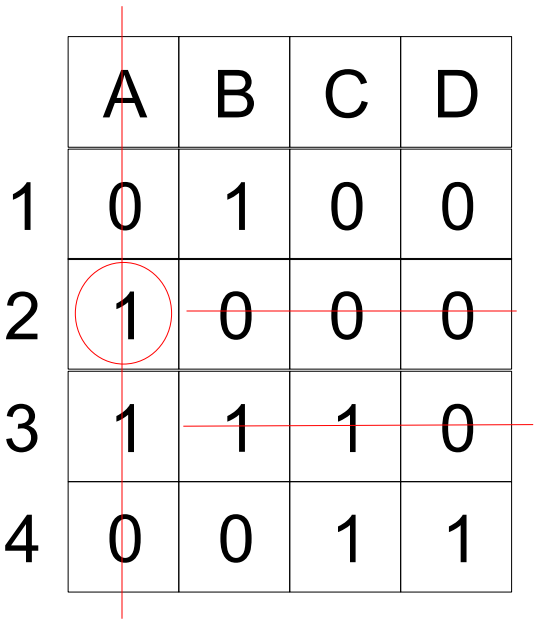
For a quick example we will look at the following matrix:

	A	B	C	D
1	0	1	0	0
2	1	0	0	0
3	1	1	1	0
4	0	0	1	1

Looking at that matrix, we will choose column A to cover. When choosing column A, we must also choose a row that is filled in column A. Rows 2 and 3 are filled. Row 2 is the correct row to choose, and 3 is incorrect. We will show a couple examples to illustrate both points.



Now we have selected row 2 of column A (this is the correct row to choose). We also add this row to the list of rows that form the solution. The next step of the algorithm is to cover the column A. The first step of covering is to delete the column (shown by a red line).



For the next step in covering column A we will delete the rows in the matrix that are filled (or have 1's) in column A. We indicate this with the red line. There is two points that need to be mentioned. Technically, in the dancing links algorithm rows are not deleted, just the "ones" that are in the matrix in that specific row are disconnected in the matrix. The dancing links algorithm does not contain zeroes in the matrix so you can't cross out or delete 0s in the matrix like in this example. The next important thing is that the red lines we are using to indicate crossing out the rows do not cross out the values in the column we are covering. We are not drawing horizontal lines through the column A, row 2 and 3 squares. This is the way that the algorithm works in Knuth's paper and no more explanation will be given why this is so, but it should be noted anyway.

The final step of the algorithm is to examine row 2 (our selected row) and find all of the 1's in it. We then cover the columns that contain 1s in that row. In this case, row 2 has all 0's left so we do not do any thing for this step.

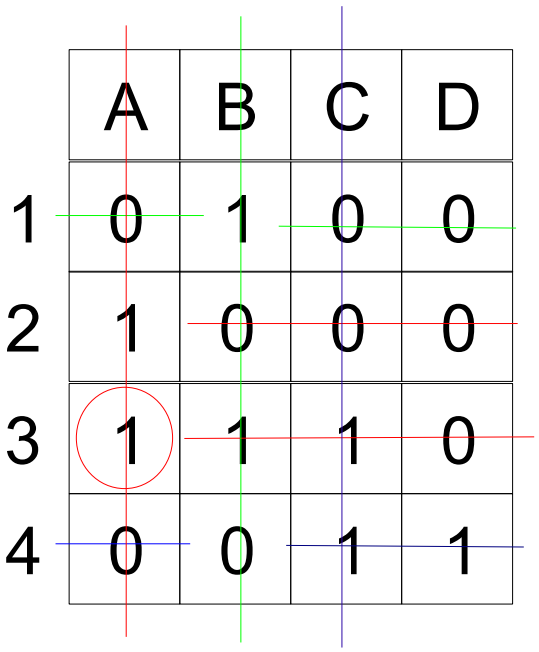
B	C	D
1	0	0
0	1	1

The above image is the matrix we are left with. This matrix is not invalid because each column that is left (B, C, and D) has at least one 1 in it.

We have now completed a step of the algorithm. So, to recap, the algorithm began by choosing a column (column A) and a row in column A to cover. We deleted the column, and then deleted the rows in the matrix that had 1's in that column. Finally, we examined our selected row to see what other 1's it

contained. That is the majority of the algorithm. After that you just repeat and select another column to cover to began the process over again.

Next, we will look at what happens if we chose the incorrect row in column A to cover off of.



Here, we select row 3 instead of row 2 and include row 3 in our list of rows that form our solution. The first step, covering column A, is shown with red lines and is identical to our previous example. The difference is that we now use row 3 to perform the basis for covering the other columns instead of row 2. We look at row 3 and find all of the 1's in row 3. There are 1's appearing in columns B and C. So, instead of original example were we stopped after covering column A, we now cover columns B and C. The process of covering B and C is identical to covering column A. We delete columns B and C by crossing them out with vertical lines. We also delete the rows in the matrix that have 1's in columns B and C.

The important part to illustrate with this example is what happens to the resulting matrix. If you look at the matrix, we have crossed out columns A, B, and C, leaving only column D left. The problem with this is that column D still exists (meaning there is no vertical line crossing it out), but column D has no elements left in its column. This is what triggers the process of backtracking. An exact cover matrix will ALWAYS have 1s in every column.

The reason the algorithm will back track at this point is for a couple of reasons. The first reason has to do with how columns are selected during the algorithm. Knuth recommends looking at every column in the matrix and to find the one that has the least number of 1s in it. When the matrix is in an invalid state, than the column with zero 1's will be choosen as the next column to start covering off of. The next step of Knuth's algorithm after choosing a column is a for loop that iterates through all of the column's members. This is Line 4 of Knuth's "search" algorithm in his paper. This for loop can not be iterated through if a column has no members. Since the invalid column has no members, the search function will exit and not recurse (since all of the algorithm's main code, including recursion instructions, are contained in the for loop). This means that the algorithm will continue back in the previous recursion state, which contains instructions on undoing what has previously been done (Lines 9, 10, 11, 12).

Implementation Notes:

The dancing links algorithm does not have any special pointers pointing to either the column headers or the rows. The only thing that points to the rows is the list of solutions. The list of solutions is best implemented as an array of pointers, each pointer pointing to a row that has been selected to be part of the solution. The array of solution row pointers should contain 81 elements, which is one for every square in a standard Sudoku board.

Since there are no special pointers pointing to the data in the matrix from outside of the matrix, besides the pointer pointing to the "head" column element, setting up the matrix can be a tricky thing.

From a perspective of keeping data tidy and eliminating memory management, it is easiest though to keep the list of column headers as an array of nodes and keep all of the column data elements as an array of nodes also. This makes it tidy for memory management. In other words, don't "malloc" every single row element and every single column element. Initialize all of the column elements in their own array and all of the row elements as their own matrix. This can be done since the number of columns and rows is already known for a Sudoku board. The fact that all of the matrix elements are kept in a big array do not defeat the structure and arrangement of the array as compared to the Knuth paper since each array element would actually be implemented as a structure. Each structure would have its own pointers (left, right, up, down) that are independent of the arrangement of the actual array in the C language.

Now, we should examine how to load up the array. First, the columns should be initialized. Create all of the columns in a big array and then for each column element, link the left and right pointers to each of the column elements in turn. Make sure to initialize the up/down pointers of the column elements to point to themselves, and make sure to initialze the "name" and "size" variables of the column elements.

Next, initialize the entire array of row elements in one go. There will be 4 x 81 of these in a Sudoku setup. Each Dancing Links matrix row will have 4 row elements. Select 4 of the elements from the C array and initialize their links so that they point to each other. Then, figure out the columns that the row must go into and link up the columns to point to the row elements you just made. It is easy to do this since the "up" pointer of each column element points to the last element in the column (bottom element).. By adding the row elements row by row, you are always adding the new row to the bottom of the stack of column elements.