DLX Sudoku Solver

1.0. Description

1.1. Exact Cover

Solving a sudoku is an exact cover problem. Let X be a set, let Y be a set of subsets of X. The exact cover problem is to find a set S such that the following are true:

1. $\forall A \in S(A \in Y)$
2. $\forall x \in X \exists A \in S(x \in A)$
3. $\forall A_1, A_2 \in S((A_1 \neq A_2) \to (A_1 \cap A_2 = \emptyset))$

In the case for sudokus, the set X represents the constraints that must be satisified (i.e. each cell has a single value and each value must be unique in its row, column and box, there are $81 \times 4 = 324$ different constraints) and the set Y represents the ways to satisfy each constraint (i.e. every possible combination of row, column and value on the sudoku grid, there are $9^3 = 729$ different combinations).

1.2. Dancing Links

The exact cover problem can be represented programmatically by a 2D 324 x 729 matrix. The column headers are the constraints and the ways to satisfy each constraint are rows. A cell in the matrix contains a 1 if that row satisfies the constraint and a 0 if it doesn't. However, this matrix is called a sparse matrix because it contains a lot more 0's than 1's. Therefore, an unnecessary amount of time is spent searching for 1's (see Optimisations 1.) and memory is wasted when storing 0's.

To solve these issues, the representation of the matrix can be optimised by using Donald Knuth's Dancing Links approach. The matrix is represented as a "toroidal" linked list (i.e. every node has a node connected to its `.left`, `.right`, `.up` and `.down`) where each node represents a 1 in the matrix. I decided to use OOP to create the `Node` and `ConstraintNode` classes to allow for reusability. The `ConstraintNode` inherits `Node` but contains an extra `num_of_nodes` property to count the number of `Nodes` that currently satisfy this constraint. The `ConstraintNodes` represent the column headers. Each `Node` contains a `constraint` property, which references a `ConstraintNode`, to allow me to easily determine the constraint that the node satisifies in $O(1)$ time. The same is true for `Node`'s `value` property, which specifies (using a tuple) the row, column and value that the `Node` represents.

1.3. Algorithm X

To solve exact cover problems, we have to choose a set of rows (i.e. combinations of row, column and value) such that a node appears exactly once in each column. I decided to use Donald Knuth's Algorithm X for this:

1. If there are no more constraints to satisfy, stop, the solution is found. Otherwise, continue.
2. Choose a constraint to fulfill (I choose the `ConstraintNode` with the lowest `num_of_nodes`)
3. Choose a row that satisfies that constraint (I choose `ConstraintNode.down`) and add it to the potential solution
4. For each node in the row, cover the constraint containing that node and cover any other row with a node that satisfies this constraint
5. Go to step 1 and determine if this row leads to a solution. If it doesn't, use a different row under this constraint.

2.0. Optimisations

1. In the initial sparse matrix, each row will only have four 1's because each combination of row, column and value will satisfy 4 constraints. Therefore, a maximum of $81 - 4 = 77$ elements would have to be searched to find the 1's. However, with Dancing Links, we can simply find the next 1 in the row using the `left` and `right` of `Node`.
2. Instead of creating a sparse 2D matrix and then converting this to the Dancing Links linked list, I skip the sparse 2D matrix and instantly create the linked list by inserting the nodes at their correct constraint using the fact that the distance between constraints are initially constant (i.e. the first 81 are for cells, next 81 are for columns, etc.).
3. For step 4 of Algorithm X, the Dancing Links technique allows me to efficiently temporarily remove and re-add columns / rows by changing the left and right / up and down pointers respectively. If I was using a sparse matrix, I would've had to create an intermediate list to temporarily store the whole row / column, which uses more memory and also time because the list has to be allocated in memory.

3.0. Reflections

I can't think of any more methods to improve the performance other than switching to a lower level programming language (such as C++) because they are less abstract so its more obvious where you are using memory and time.