Sudoku problem (Backtracking)

5	3	1	2	7	6	8	9	4
6	2	4	1	9	5	2		
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Sudoku

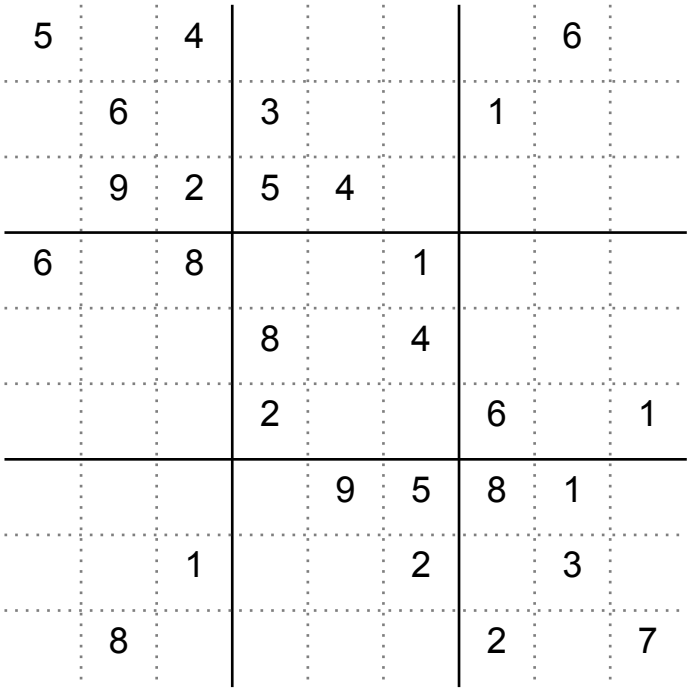
- Given an incomplete Sudoku in the form of matrix mat[][] of order 9*9, the task is to complete the Sudoku.
- A sudoku solution must satisfy all of the following rules:

- Each of the digits 1-9 must occur exactly once in each row.
- Each of the digits 1-9 must occur exactly once in each column.
- Each of the digits 1-9 must occur exactly once in each of the 9, 3x3 sub-boxes of the grid.
- Note: Zeros in the mat[][] indicate blanks, which are to be filled with some number between 1 to 9. You can not replace the element in the cell which is not blank.

Sudoku

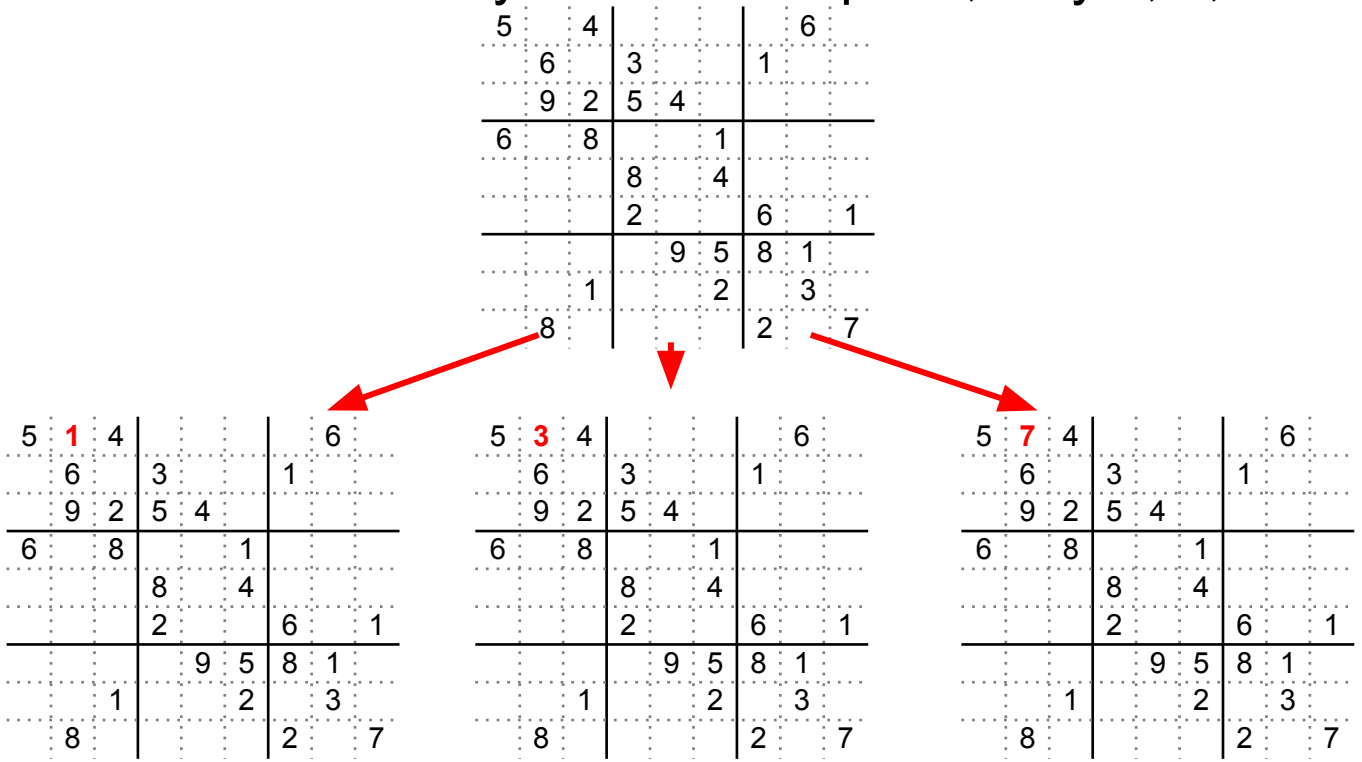
In the first case, consider the game Sudoku:

- The search space is 9^{53}



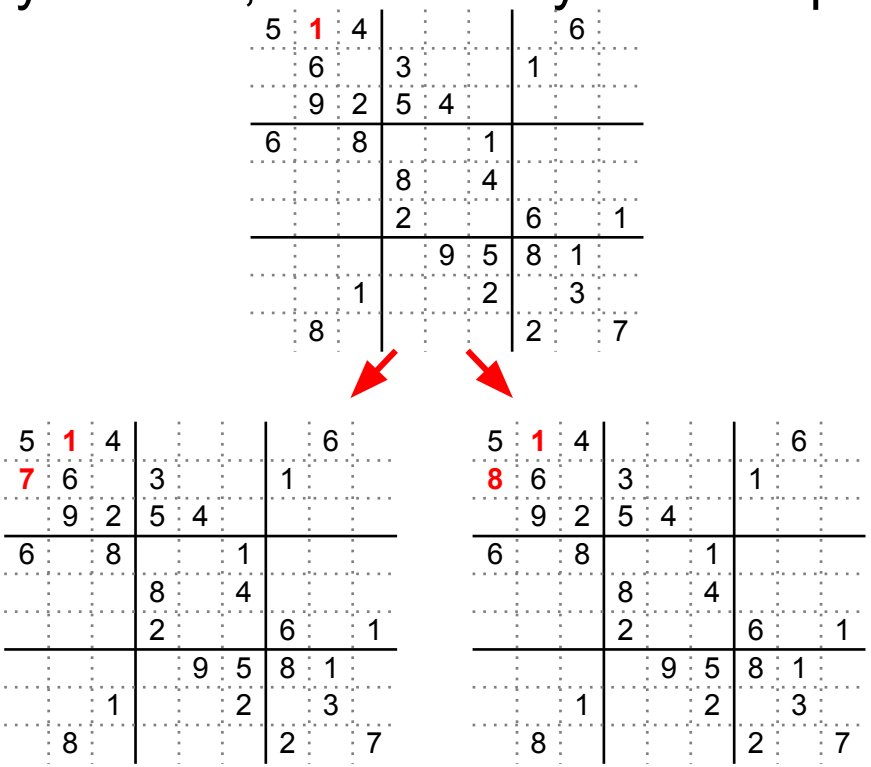
Sudoku

At least for the first entry in the first square, only 1, 3, 7 fit



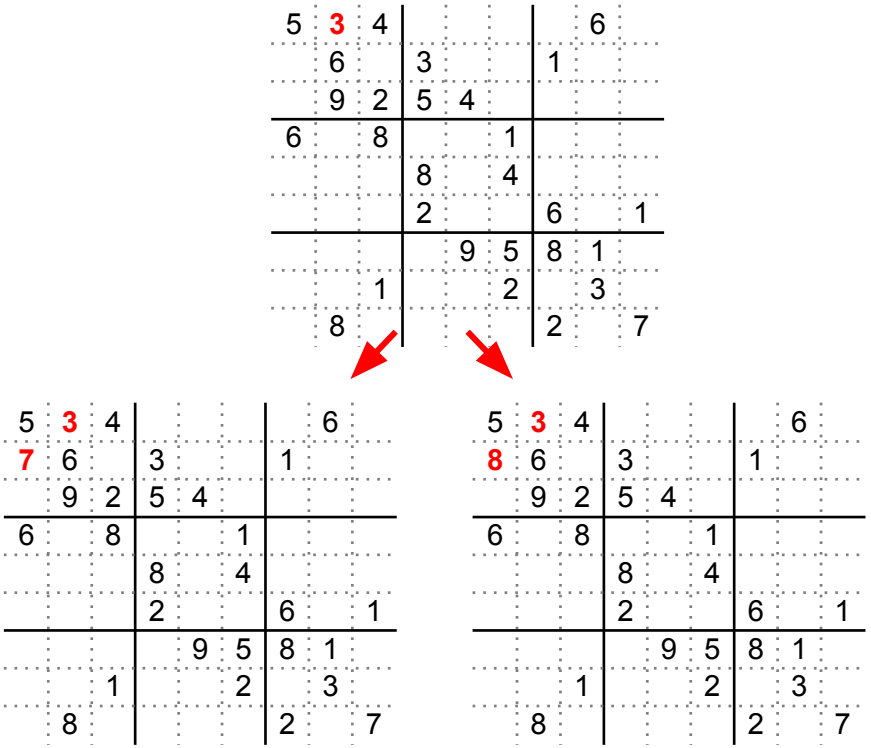
Sudoku

If the first entry has a 1, the 2nd entry in that square could be 7 or 8



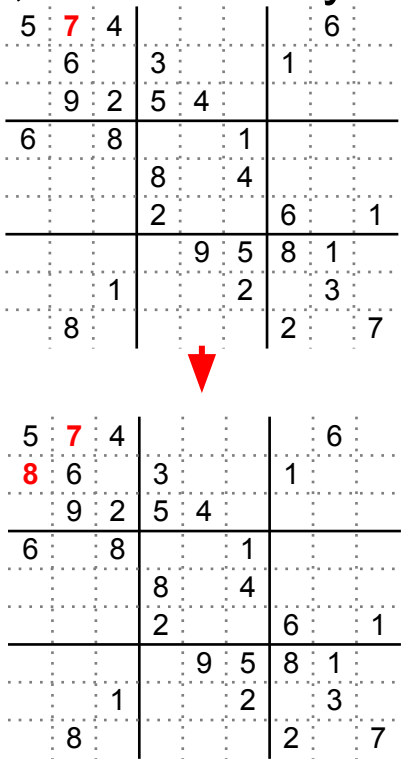
Sudoku

If the first entry has a 3, the 2nd entry in that square could be 7 or 8

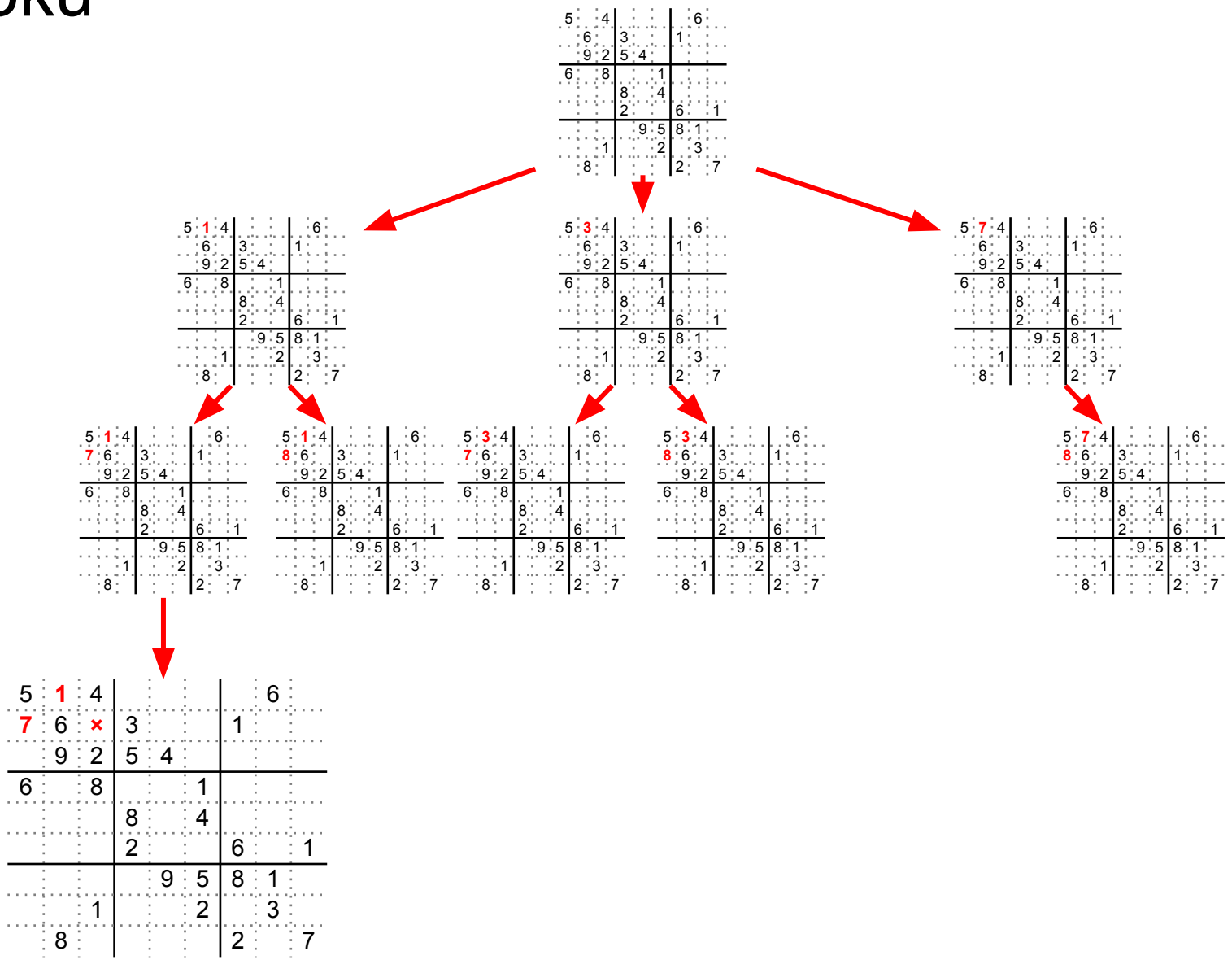


Sudoku

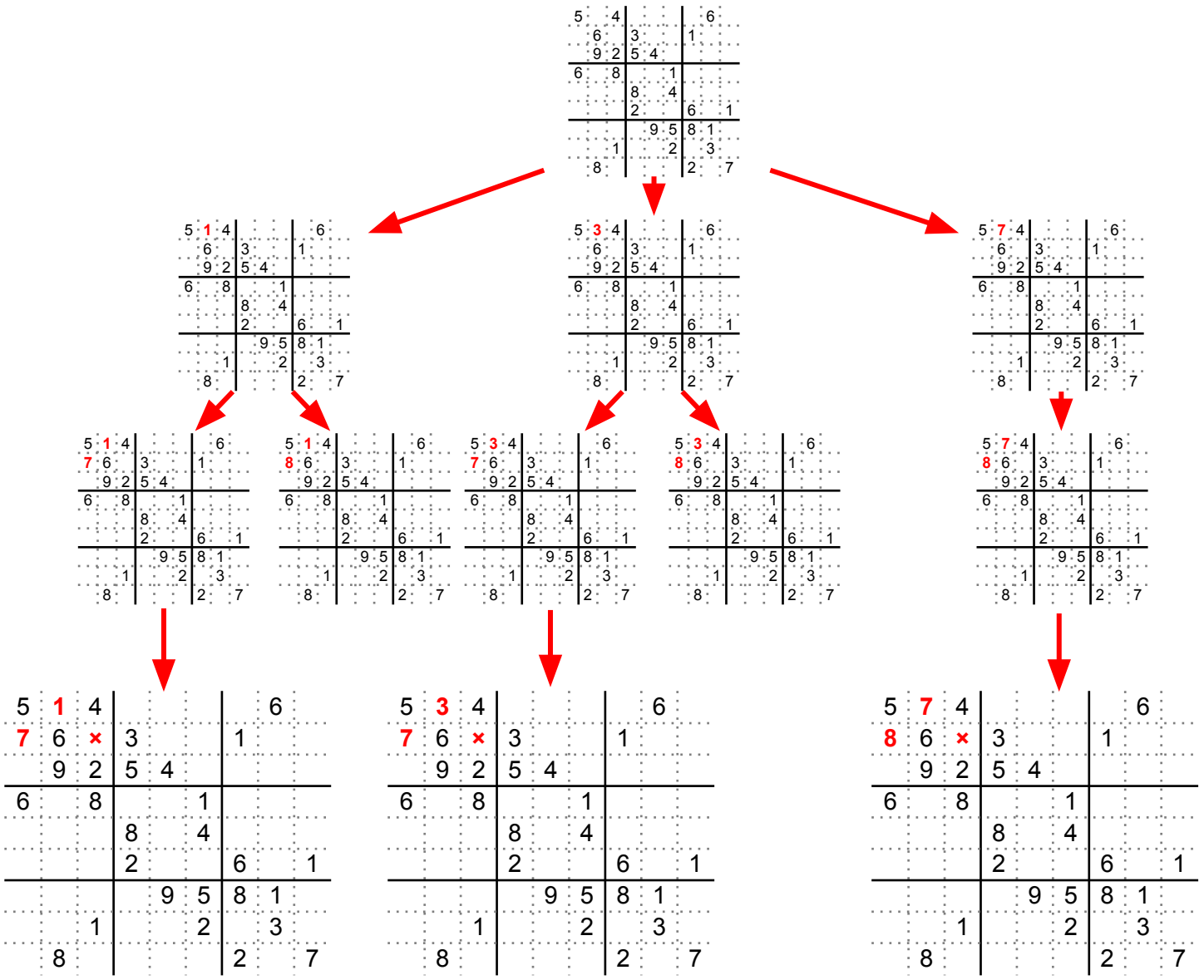
If the first entry has a 7, the 2nd entry in that square could be 8

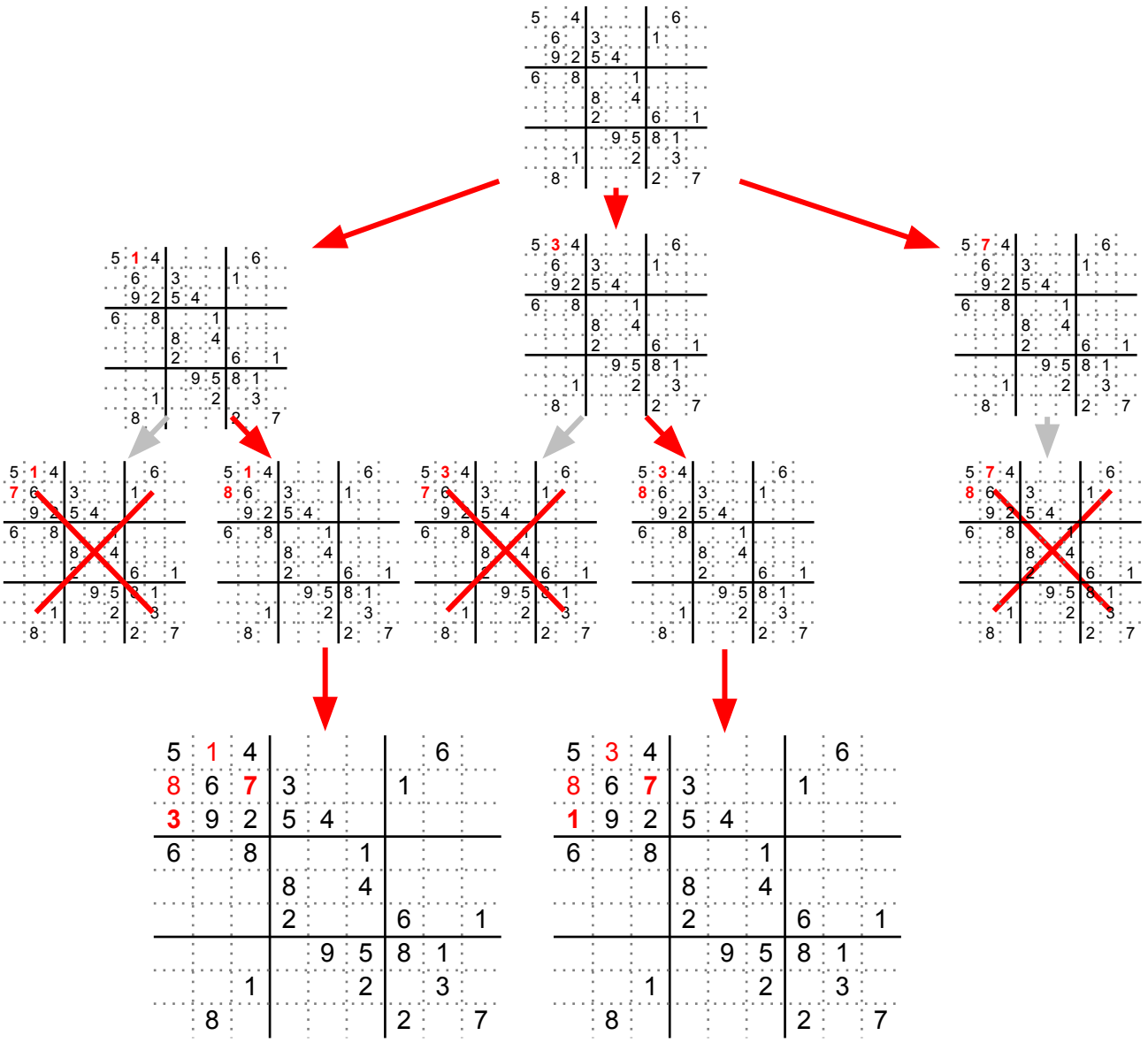


Sudoku



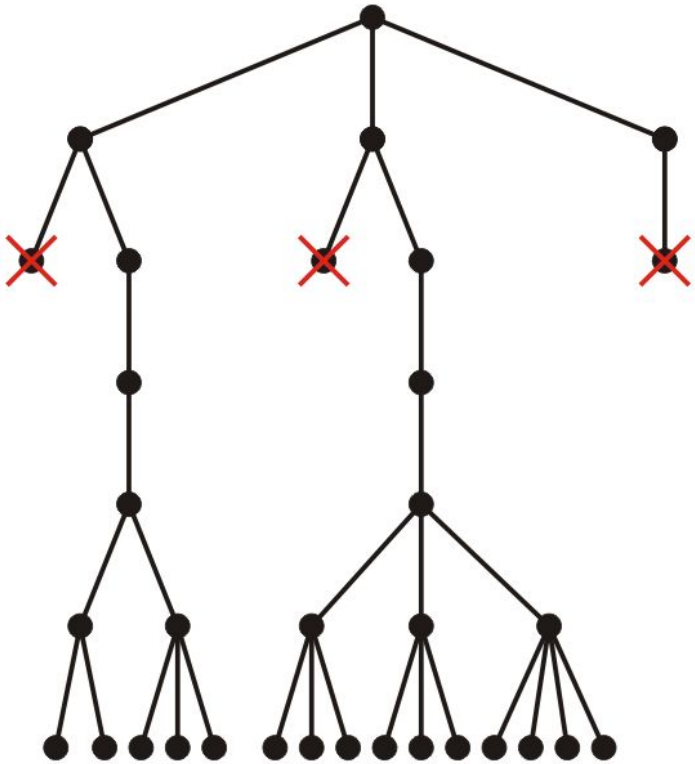
Sudoku





Sudoku

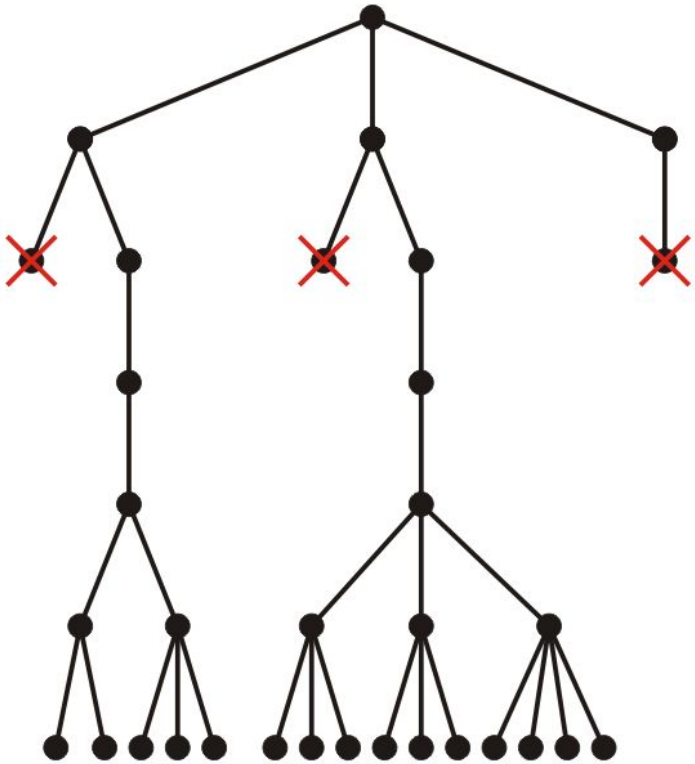
It may seem that this is a reasonably straight-forward method; however, the decision tree continues to branch quick once we start filling the second square



Sudoku

A binary tree of this height would have around $2^{54} - 1$ nodes

- Fortunately, as we get deeper into the tree, more get cut



Implementation

Our straight-forward implementation takes a 9 × 9 matrix

- Default entries are values from 1 to 9, empty cells are `0`
- Two helper functions:

```
bool next_location( int[9][9], int &i, int &j )
```

  - Finds the next location empty location returning false if none is found

```
bool is_valid( int[9][9], int i, int j, int value )
```

  - Checks if there are any conflicts created if `matrix[i][j]` is assigned `value`
- The `backtracing` function:
  - Finds the next unoccupied cell
  - For each value from 1 to 9, it checks if it is valid to insert that it there
    - If so, backtracking is called recursively on the matrix with that entry set

The main function creates the initial matrix and calls backtrack

Implementation

```
// Find the next empty location in 'matrix'
// If one is found, assign 'i' and 'j' the indexes of that entry
// Otherwise, return false
//  - In this case, the values of 'i' and 'j' are undefined
bool next_location( int matrix[9][9], int &i, int &j ) {
    for ( int i1 = 0; i1 < 3; ++i1 ) {
        for ( int j1 = 0; j1 < 3; ++j1 ) {
            for ( int i2 = 0; i2 < 3; ++i2 ) {
                for ( int j2 = 0; j2 < 3; ++j2 ) {
                    i = 3*i1 + i2;
                    j = 3*j1 + j2;

                    // return 'true' if we find an
                    // unoccupied entry
                    if ( matrix[i][j] == 0 )
                        return true;
                }
            }
        }
    }

    return false;  // all the entries are occupied
}
```

```
// If 'value' already appears in
//  - the row 'm'
//  - the column 'n'
//  - the 3x3 square of entries it (m, n) appears in
// return true, otherwise return false
bool is_valid( int matrix[9][9], int m, int n, int value ) {
    // Check if 'value' already appears in either a row or column
    for ( int i = 0; i < 9; ++i ) {
        if ( (matrix[m][i] == value) || ( matrix[i][n] == value ) )
            return false;
    }

    // Check if 'value' already appears in either a row or column
    int ioff = 3*(m/3);
    int joff = 3*(n/3);

    for ( int i = 0; i < 3; ++i ) {
        for ( int j = 0; j < 3; ++j ) {
            if ( matrix[ioff + i][joff + j] == value )
                return false;  // 'value' already in the 3x3 square
        }
    }

    return true;  // 'value' could be added
}
```

Implementation

```
bool backtrack( int matrix[9][9] ) {
   int i, j;

   // If the matrix is full, we are done
   if ( !next_location( matrix, i, j ) ) {
      return true;
   }
   for ( int value = 1; value <= 9; ++value ) {
      if ( is_valid( matrix, i, j, value ) ) {
         // Assume this entry is part of the
         // solution--recursively call backtrack
         matrix[i][j] = value;
         // If we found a solution, return
         // otherwise, reset the entry to 0
         if ( backtrack( matrix ) ) {
            return true;
         } else {
            matrix[i][j] = 0;
         }
      }
   }
   // No solution found--reset the entry to 0
   return false;
}
```

Implementation

```
int main() {
    int matrix[9][9] = {
        {5, 0, 4, 0, 0, 0, 0, 6, 0},
        {0, 6, 0, 3, 0, 0, 1, 0, 0},
        {0, 9, 2, 5, 4, 0, 0, 0, 0},
        {6, 0, 8, 0, 0, 1, 0, 0, 0},
        {0, 0, 0, 8, 0, 4, 0, 0, 0},
        {0, 0, 0, 2, 0, 0, 6, 0, 1},
        {0, 0, 0, 0, 9, 5, 8, 1, 0},
        {0, 0, 1, 0, 0, 2, 0, 3, 0},
        {0, 8, 0, 0, 0, 0, 2, 0, 7}
    };
    // If found, print out the resulting matrix
    if ( backtrack( matrix ) ) {
        for ( int i = 0; i < 9; ++i ) {
            for ( int j = 0; j < 9; ++j ) {
                std::cout << matrix[i][j] << " ";
            }

            std::cout << std::endl;
        }
    }
    return 0;
}
```

Implementation

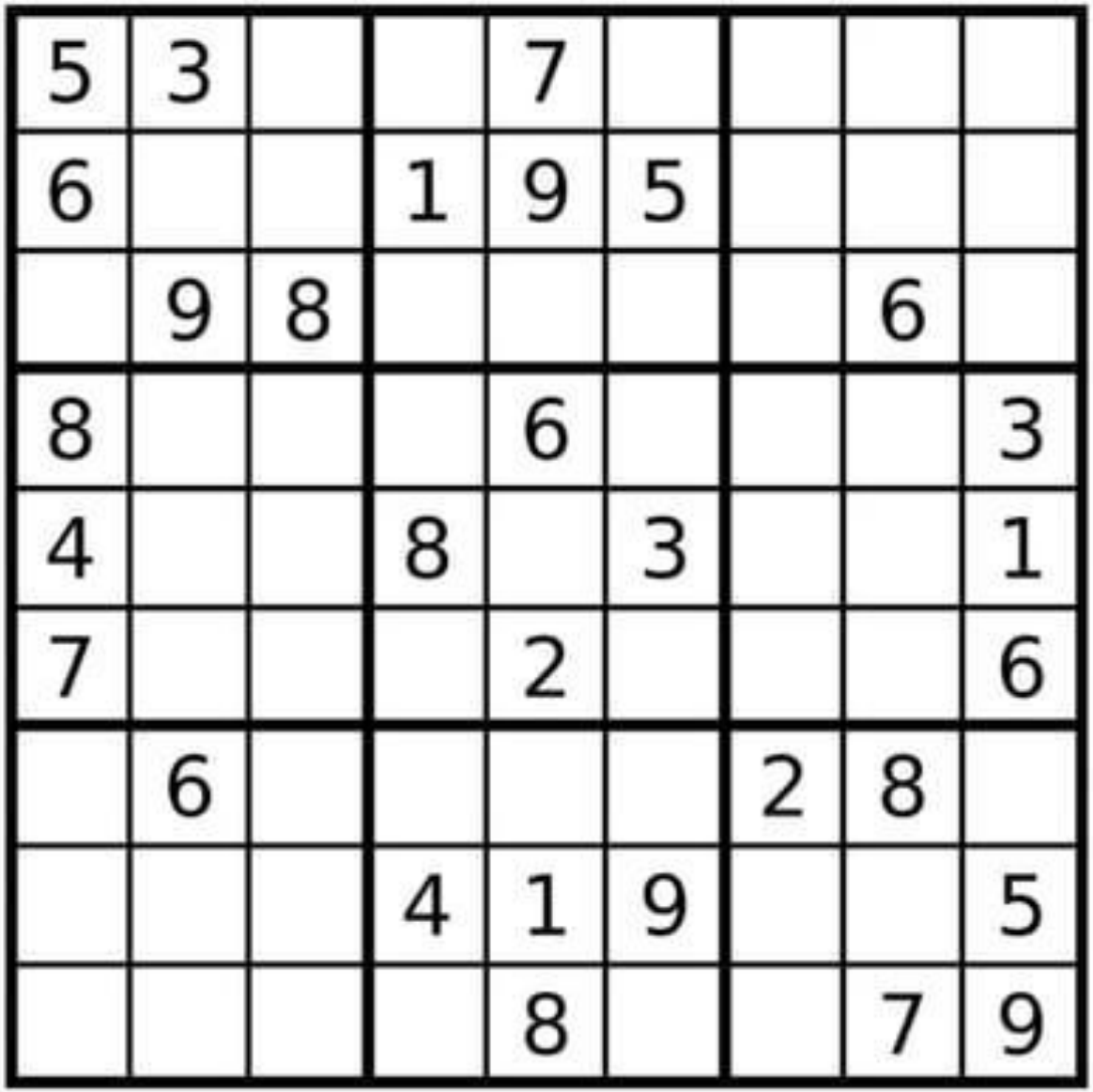
In this case, the traversal:

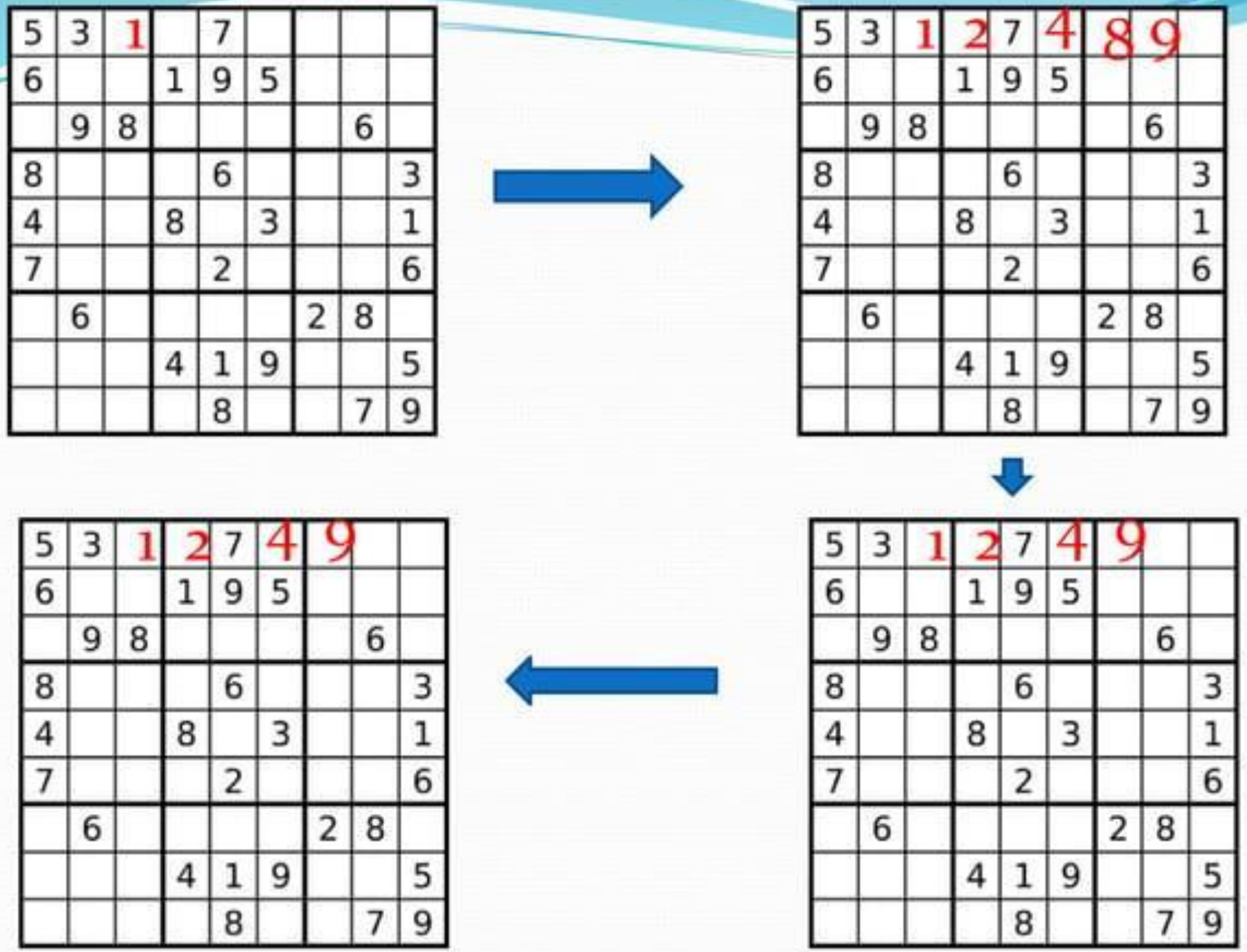
- Recursively calls backtrack 874 times
  - The last one determines that there are no unoccupied entries
- Checks if a placement is valid 7658 times

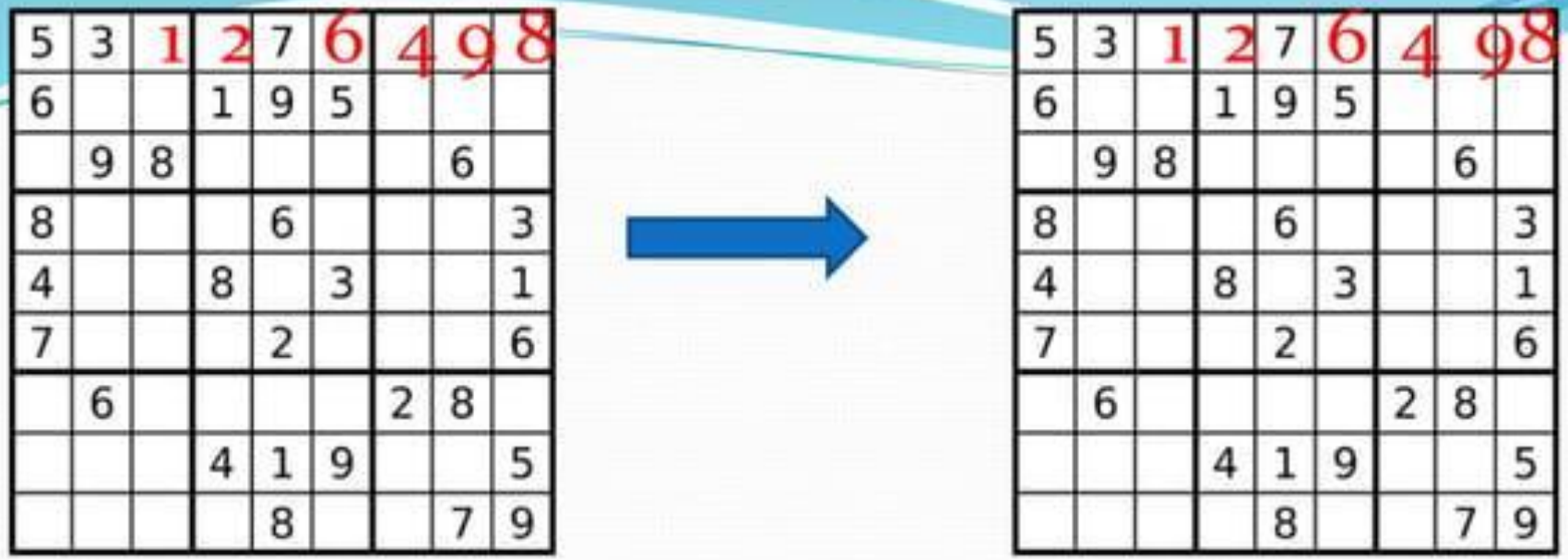
5	3	4	1	7	8	9	6	2
8	6	7	3	2	9	1	4	5
1	9	2	5	4	6	3	7	8
6	7	8	9	3	1	5	2	4
2	1	5	8	6	4	7	9	3
3	4	9	2	5	7	6	8	1
4	2	3	7	9	5	8	1	6
7	5	1	6	8	2	4	3	9
9	8	6	4	1	3	2	5	7

Example:

Solve the given Sudoku problem.




5 3 1 7
6 1 9 5
9 8 6
8 6 3
4 8 3 1
7 2 6
6 2 8
4 1 9 5
8 7 9
5 3 1 2 7 4 8 9
6 1 9 5
9 8 6
8 6 3
4 8 3 1
7 2 6
6 2 8
4 1 9 5
8 7 9
5 3 1 2 7 4 9
6 1 9 5
9 8 6
8 6 3
4 8 3 1
7 2 6
6 2 8
4 1 9 5
8 7 9
5 3 1 2 7 4 9
6 1 9 5
9 8 6
8 6 3
4 8 3 1
7 2 6
6 2 8
4 1 9 5
8 7 9



MOVING THROUGH THE NEXT LINES AND FOLLOWING THE SAME PROCEDURE WE WILL BE ABLE TO SOLVE THIS PROBLEM WITH BACKTRACKING ALGORITHM.

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Algorithm Performance

- **Time complexity:** $O(9^{(n*n)})$, For every unassigned index, there are 9 possible options and for each index, we are checking other columns, rows and boxes.
  Auxiliary Space: O(1)