project

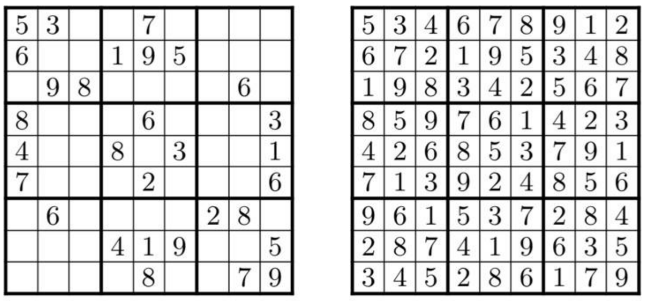
sudoku

Name : Anoud Alsubaie

Name: Meznah Alrasheed

Introduction

Sudoku, also known as Number Place in the United States, is a placement puzzle. Given a grid – most frequently a 9 × 9 grid made up of 3 × 3 subgrids called “regions” – with various digits given in some cells (the “givens”), the aim is to enter a digit from 1 through 5 in each cell of the grid so that each row, column and region contains only one instance of each digit. Fig. 1 shows a Sudoku on the left, along with its unique solution on the right [7]. Note that other symbols (e.g. letters, icons) could be used instead of digits, as their arithmetic properties are irrelevant in the context of Sudoku. This is currently a rather popular puzzle that is featured in a number of newspapers and puzzle magazines [1, 2, 5].
Several Sudoku solvers are available already [4, 6]. Since there are more than 6 · 1021 possible Sudoku grids [3].



5	3			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	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

Fig. 1. *Sudoku* example and solution

The goal of the puzzle is to find numbers for the remaining cells with three rules:

i. Each horizontal row should contain the numbers 1 - 9, without repeating any.
ii. Each vertical column should contain the numbers 1 - 9, without repeating any.
iii. Each 3 ×3 block should contain the numbers 1 - 9, without repeating any. “[8]

Uninformed search algorithm

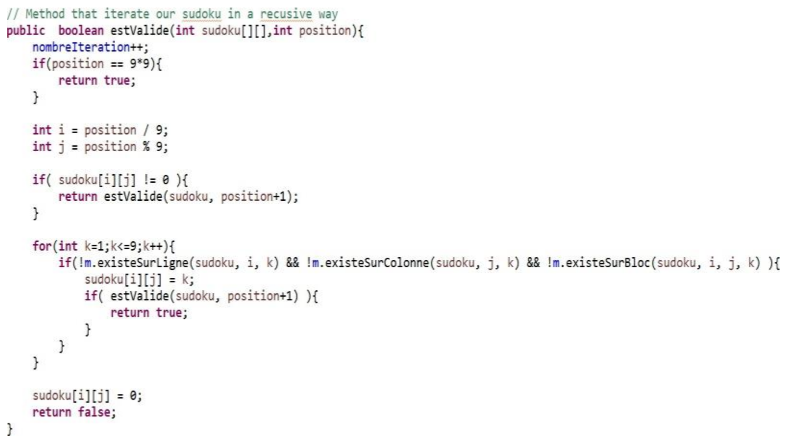
Backtracking algorithm tries to solve the puzzle by testing each cell for a valid solution.
If there’s no violation of constraints, the algorithm moves to the next cell, fills in all potential solutions and repeats all checks.
If there’s a violation, then it increments the cell value. Once, the value of the cell reaches 9, and there is still violation then the algorithm moves back to the previous cell and increases the value of that cell.
It tries all possible solutions.

Sudoku solvers therefore combine backtracking with – sometimes complicated – methods for constraint propagation. In this paper we propose a SAT-based approach: A Sudoku is translated into a propositional formula that is satisfiable if and only if the Sudoku has a solution. The propositional formula is then presented to a standard SAT solver, and if the SAT solver finds a satisfying assignment, this assignment can readily be transformed into a solution for the original Sudoku [9].

In the beginning, the program using java code , will start by displaying a page GUI and containing five buttons. The first game starts with Sudoku. The second game is solved by the Backtracking algorithm and the third is via Forward Checking algorithm and the fourth is via Simulated Annealing algorithm , with respect to the fifth button gives the user the right to choose the level of the game. The time it took to solve the game and the number of iterations.

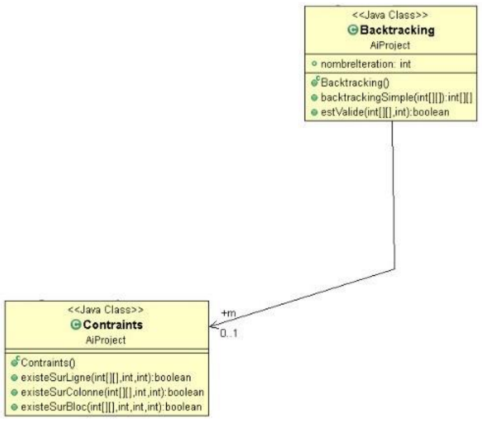
Implementation :

Let’s create a **estValide**() method that takes the board as the input parameter and iterates through rows, columns and values testing each cell for a valid solution:



```java
// Method that iterate our sudoku in a recusive way
public  boolean estValide(int sudoku[][],int position){
    nombreIteration++;
    if(position == 9*9){
        return true;
    }

    int i = position / 9;
    int j = position % 9;

    if( sudoku[i][j] != 0 ){
        return estValide(sudoku, position+1);
    }

    for(int k=1;k<=9;k++){
        if(!m.existeSurLigne(sudoku, i, k) && !m.existeSurColonne(sudoku, j, k) && !m.existeSurBloc(sudoku, i, j, k) ){
            sudoku[i][j] = k;
            if( estValide(sudoku, position+1) ){
                return true;
            }
        }
    }

    sudoku[i][j] = 0;
    return false;
}
```

Fig.2. EstValide()



<<Java Class>>
Backtracking
AiProject
nombreIteration: int
Backtracking()
backtrackingSimple(int[][]):int[][]
estValide(int[][],int):boolean
<<Java Class>>
Contraints
AiProject
Contraints()
existeSurLigne(int[][],int,int):boolean
existeSurColonne(int[][],int,int):boolean
existeSurBloc(int[][],int,int,int):boolean
+m
0..1

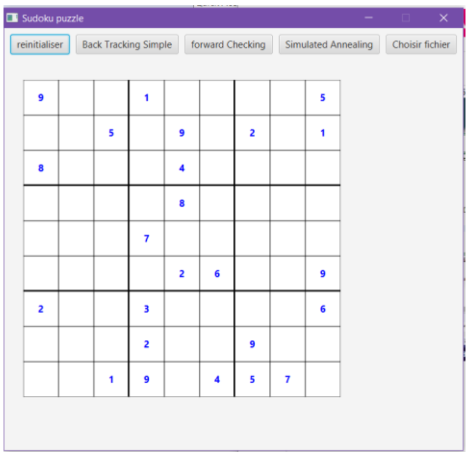
Fig.3. class Backtracking & constraints

In Fig. 3, the Backtracking class is displayed and its relationship with class constraints, and contains the class three method : - Backtracking() - BacktrackingSimple(int[][]) - estValide(int[][],int) and class four method - contraints() - existeSurLigne(int[][],int,int) -existeSurColonne(int[][],int,int) - existeSurBloc(int[][],int,int,int)

Results:

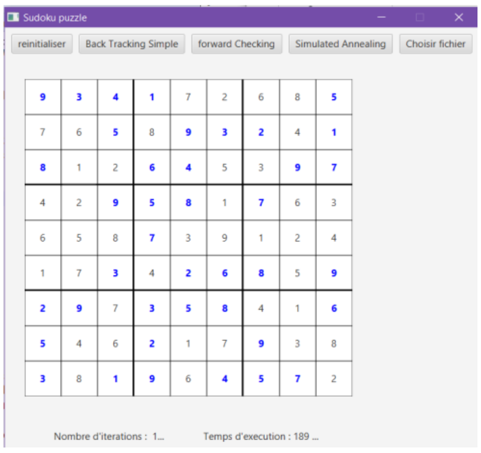
- Test Board

- Solved Board



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

Fig.4. Test Board



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

Nombre d'iterations : 1...   Temps d'execution : 189 ...

Fig.5. Solved Board

We've successfully implemented backtracking algorithm that solves the Sudoku puzzle!

Obviously, there's room for improvements as the algorithm naively checks each possible combination over and over again (even though we know the particular solution is invalid).

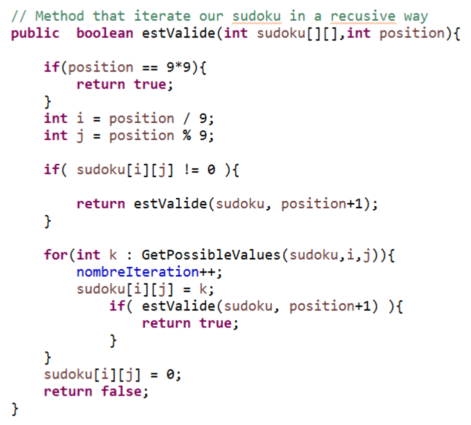
Forward checking

Forward checking is the easy way to solve sudoku. Instead of performing arc consistency to the instantiated variables, it performs restricted form of arc consistency to the not yet instantiated variables.

The advantage of this is that if the domain of a future variable becomes empty, it is known immediately that the current partial solution is inconsistent. Forward checking therefore allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. Note that whenever a new variable is considered, all its remaining values are guaranteed to be consistent with the past variables, so the checking an assignment against the past assignments is no longer necessary.

Forward checking detects the inconsistency earlier than simple backtracking and thus it allows branches of the search tree that will lead to failure to be pruned earlier than with simple backtracking. This reduces the search tree and (hopefully) the overall amount of work done. But it should be noted that forward checking does more work when each assignment is added to the current partial solution [10].

Implementation:



```
// Method that iterate our sudoku in a recusive way
public  boolean estValide(int sudoku[][],int position){

    if(position == 9*9){
        return true;
    }
    int i = position / 9;
    int j = position % 9;

    if( sudoku[i][j] != 0 ){

        return estValide(sudoku, position+1);
    }

    for(int k : GetPossibleValues(sudoku,i,j)){
        nombreIteration++;
        sudoku[i][j] = k;
            if( estValide(sudoku, position+1) ){
                return true;
            }
    }
    sudoku[i][j] = 0;
    return false;
}
```

Fig.6. EstValide()

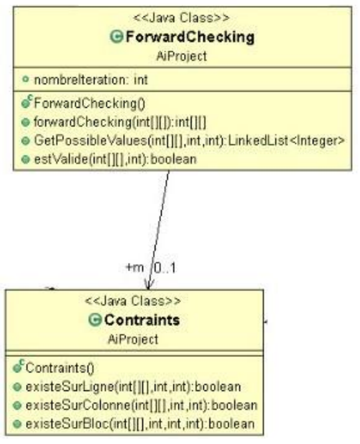
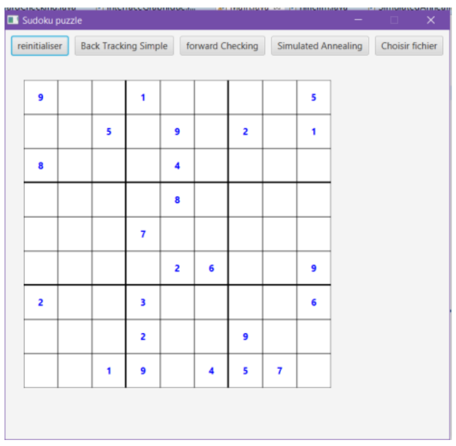

<<Java Class>>
ForwardChecking
AiProject
nombreIteration: int
ForwardChecking()
forwardChecking(int[][]):int[][]
GetPossibleValues(int[][],int,int):LinkedList<Integer>
estValide(int[][],int):boolean
+m
0..1
<<Java Class>>
Contraints
AiProject
Contraints()
existeSurLigne(int[][],int,int):boolean
existeSurColonne(int[][],int,int):boolean
existeSurBloc(int[][],int,int,int):boolean

Fig.7. class ForwardChecking & constraints

In Fig. 7 , the ForwardChecking class is displayed and its relationship with class constraints, and contains the class four method : - ForwardChecking() -forwardChecking(int[][])
 - GetPossibleValues(int[][],int,int) -estValideve(int[][],int)
and class four method - contraints() - existeSurLigne(int[][],int,int) -existeSurColonne(int[][],int,int)
 - existeSurBloc(int[][],int,int,int)

Results:

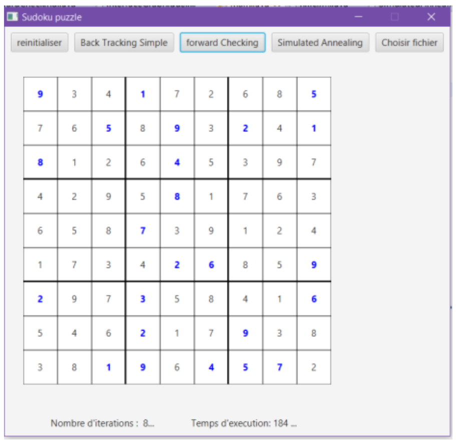
- **Test Board**



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

Fig.8. Test Board

- **Solved Board**



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

Nombre d'iterations : 8... Temps d'execution: 184 ...

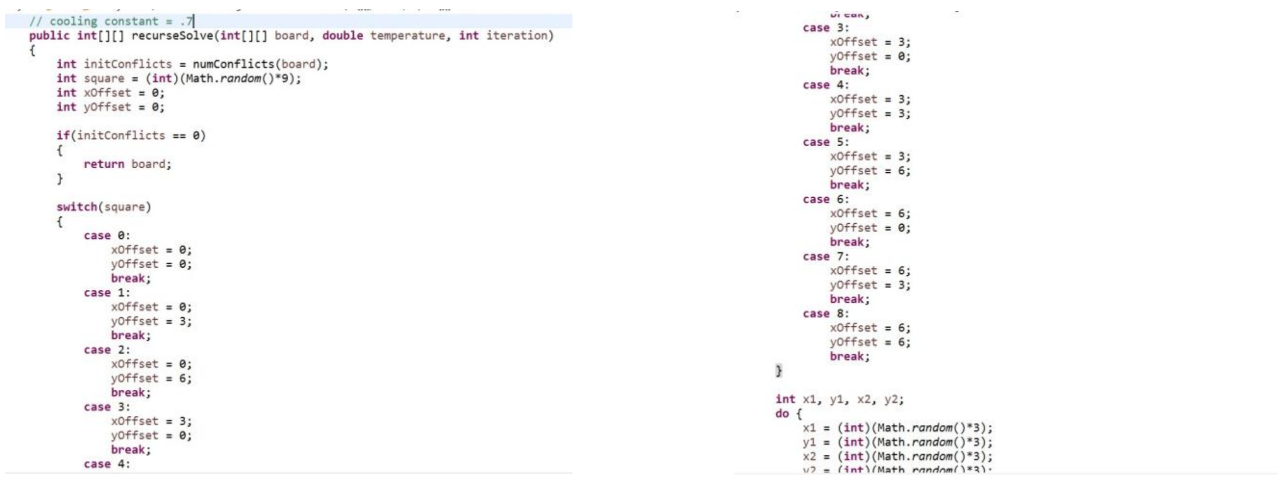
Fig.9. Solved Board

Local search algorithm

Simulated Annealing (SA) is an effective and general form of optimization. It is useful in finding global optima in the presence of large numbers of local optima. Simulated annealing uses the objective function of an optimization problem instead of the energy of a material.

write a detailed description of how the project was designed. The description should be readable by less technical people, and at the same time helpful for more technical people. Use tables, uml and diagrams if needed.

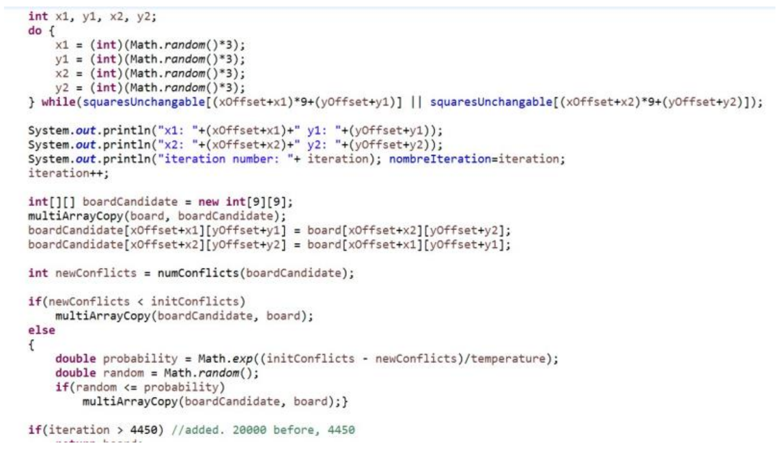
Implementation:



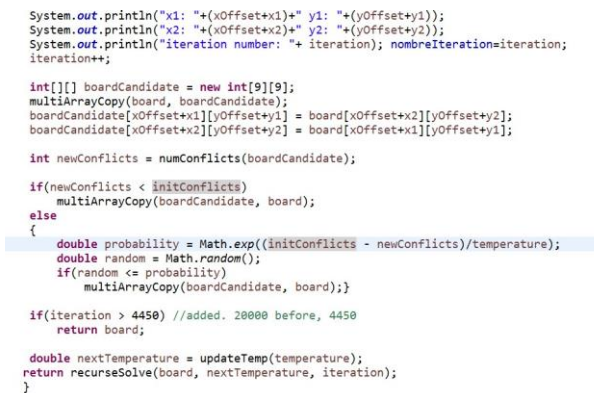
```
    // cooling constant = .7
    public int[][] recurseSolve(int[][] board, double temperature, int iteration)
    {
        int initConflicts = numConflicts(board);
        int square = (int)(Math.random()*9);
        int xOffset = 0;
        int yOffset = 0;

        if(initConflicts == 0)
        {
            return board;
        }

        switch(square)
        {
            case 0:
                xOffset = 0;
                yOffset = 0;
                break;
            case 1:
                xOffset = 0;
                yOffset = 3;
                break;
            case 2:
                xOffset = 0;
                yOffset = 6;
                break;
            case 3:
                xOffset = 3;
                yOffset = 0;
                break;
            case 4:
```

```
            case 3:
                xOffset = 3;
                yOffset = 0;
                break;
            case 4:
                xOffset = 3;
                yOffset = 3;
                break;
            case 5:
                xOffset = 3;
                yOffset = 6;
                break;
            case 6:
                xOffset = 6;
                yOffset = 0;
                break;
            case 7:
                xOffset = 6;
                yOffset = 3;
                break;
            case 8:
                xOffset = 6;
                yOffset = 6;
                break;
        }

        int x1, y1, x2, y2;
        do {
            x1 = (int)(Math.random()*3);
            y1 = (int)(Math.random()*3);
            x2 = (int)(Math.random()*3);
```

Fig.10. recurseSolve()



```java
int x1, y1, x2, y2;
do {
    x1 = (int)(Math.random()*3);
    y1 = (int)(Math.random()*3);
    x2 = (int)(Math.random()*3);
    y2 = (int)(Math.random()*3);
} while(squaresUnchangable[(xOffset+x1)*9+(yOffset+y1)] || squaresUnchangable[(xOffset+x2)*9+(yOffset+y2)]);

System.out.println("x1: "+(xOffset+x1)+" y1: "+(yOffset+y1));
System.out.println("x2: "+(xOffset+x2)+" y2: "+(yOffset+y2));
System.out.println("iteration number: "+ iteration); nombreIteration=iteration;
iteration++;

int[][] boardCandidate = new int[9][9];
multiArrayCopy(board, boardCandidate);
boardCandidate[xOffset+x1][yOffset+y1] = board[xOffset+x2][yOffset+y2];
boardCandidate[xOffset+x2][yOffset+y2] = board[xOffset+x1][yOffset+y1];

int newConflicts = numConflicts(boardCandidate);

if(newConflicts < initConflicts)
    multiArrayCopy(boardCandidate, board);
else
{
    double probability = Math.exp((initConflicts - newConflicts)/temperature);
    double random = Math.random();
    if(random <= probability)
        multiArrayCopy(boardCandidate, board);}

if(iteration > 4450) //added. 20000 before, 4450
```



```java
System.out.println("x1: "+(xOffset+x1)+" y1: "+(yOffset+y1));
System.out.println("x2: "+(xOffset+x2)+" y2: "+(yOffset+y2));
System.out.println("iteration number: "+ iteration); nombreIteration=iteration;
iteration++;

int[][] boardCandidate = new int[9][9];
multiArrayCopy(board, boardCandidate);
boardCandidate[xOffset+x1][yOffset+y1] = board[xOffset+x2][yOffset+y2];
boardCandidate[xOffset+x2][yOffset+y2] = board[xOffset+x1][yOffset+y1];

int newConflicts = numConflicts(boardCandidate);

if(newConflicts < initConflicts)
    multiArrayCopy(boardCandidate, board);
else
{
    double probability = Math.exp((initConflicts - newConflicts)/temperature);
    double random = Math.random();
    if(random <= probability)
        multiArrayCopy(boardCandidate, board);}

if(iteration > 4450) //added. 20000 before, 4450
    return board;

double nextTemperature = updateTemp(temperature);
return recurseSolve(board, nextTemperature, iteration);
}
```

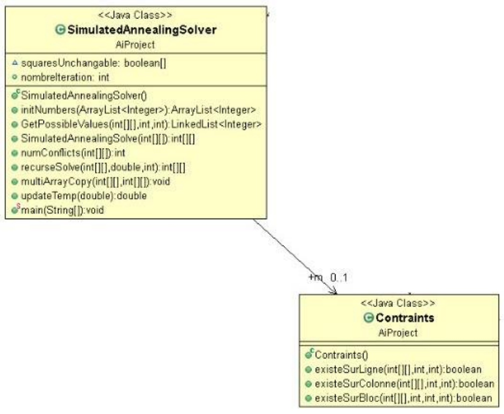
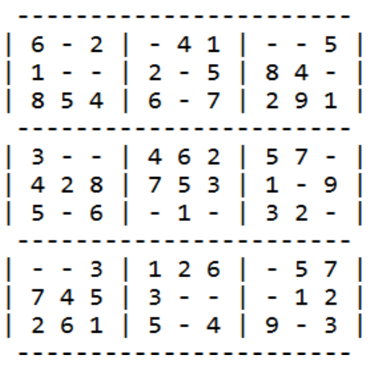

<<Java Class>>
SimulatedAnnealingSolver
AiProject
squaresUnchangable: boolean[]
nombreIteration: int
SimulatedAnnealingSolver()
initNumbers(ArrayList<Integer>):ArrayList<Integer>
GetPossibleValues(int[][],int,int):LinkedList<Integer>
SimulatedAnnealingSolve(int[][]):int[][]
numConflicts(int[][]):int
recurseSolve(int[][],double,int):int[][]
multiArrayCopy(int[][],int[][]):void
updateTemp(double):double
main(String[]):void
+m 0..1
<<Java Class>>
Contraints
AiProject
Contraints()
existeSurLigne(int[][],int,int):boolean
existeSurColonne(int[][],int,int):boolean
existeSurBloc(int[][],int,int,int):boolean

Fig.11. class SimulatedAnnealingSolver & constraints

In Fig. 11 , the SimulatedAnnealingSolver class is displayed and its relationship with class constraints, and contains the class nine method : - SimulatedAnnealingSolver() -initNumbers(ArrayList<Integer>) - GetPossibleValues(int[][],int,int) -SimulatedAnnealingSolve(int[][]) -numConflicts(int[][]) - recurseSolve(int[][],double,in) -multiArrayCopy(int[][],int[][]) - updataTemp(double) -main(String[])
and class four method - contraints() - existeSurLigne(int[][],int,int) -existeSurColonne(int[][],int,int) - existeSurBloc(int[][],int,int,int)

Results:

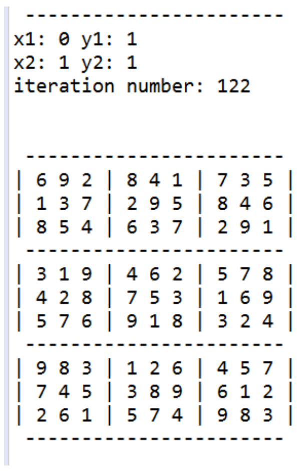
- **Test Board**



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

Fig.12. Test Board

- **Solved Board**



```
 -----------------------
x1: 0 y1: 1
x2: 1 y2: 1
iteration number: 122


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

Fig.13. Solved Board

Conclusion

We've discussed two solutions to a sudoku puzzle with core Java. We have presented a straightforward translation of a Sudoku into a propositional formula. The translation can easily be generalized from 9x9 grids to grids of arbitrary dimension. It is polynomial in the size of the grid; we are used is different algorithm such as backtracking and forward checking and Simulated Annealing. At the end they are solved sudoku.

Source code

https://github.com/erichowens/SudokuSolver
https://www.geeksforgeeks.org/sudoku-backtracking-7/
https://github.com/mattnenterprise/Sudoku

Reference

[1] Col Allan, editor. New York Post. News Corporation, New York City, NY, USA, 2005.

[2] Giovanni di Lorenzo, editor. Die Zeit. Zeitverlag Gerd Bucerius GmbH & Co. KG, Hamburg, Germany, 2005.

[3] Bertram Felgenhauer and Frazer Jarvis. Enumerating possible Sudoku grids, June 2005. Available online at http://www.shef.ac.uk/~pm1afj/sudoku/.

[4] DeadMan's Handle Ltd. Sudoku solver, September 2005. Available online at http://www.sudoku-solver.com/.

[5] Robert James Thomson, editor. The Times. Times Newspapers Ltd., London, UK, 2005.

[6] Pete Wake. Sudoku solver by logic, September 2005. Available online at http: //www.sudokusolver.co.uk/.

[7] Wikipedia. Sudoku – Wikipedia, the free encyclopedia, September 2005. Available online at http://en.wikipedia.org/wiki/Sudoku.

[8] Abdel-Raouf , Osama. and Abdel-Baset, Mohamed. (2014). A Novel Hybrid Flower Pollination Algorithm with Chaotic Harmony Search for Solving Sudoku Puzzles

[9] Weber, Tjark. "A SAT-based Sudoku solver." LPAR-12, Short paper proc. 2005.

[10]"Constraint Guide - Constraint Propagation", *Ktiml.mff.cuni.cz*, 2019. [Online]. Available: https://ktiml.mff.cuni.cz/~bartak/constraints/propagation.html. [Accessed: 08- Apr- 2019].