AI PRINCIPLES & TECHNIQUES

Sudoku Report

Ege SARI *s1034535*
Radboud University *30 November 2021*

Contents

1 Introduction

In this section we will explain the goals of the project and give a brief description of Sudoku. Also we will explain the constraint satisfaction problems and AC-3 algorithm briefly.

1.1 Goals of the Project

The main goal of the project is understand the constraint satisfaction problems and solutions for these problems. Using arc consistency is one of the solutions and it can be improved by some heuristics. In this project we will solve a constraint satisfaction problem, Sudoku, by using one of the arc consistency algorithm, AC-3. Also we will enhance our algorithm by using some heuristics such as minimum remaining value and we will test the heuristics to draw results. These results will illustrate a comparison of complexities of different heuristics.

1.2 About Sudoku

Sudoku is a game contains 81 fields in a 9×9 square area. Also there are 9 3×3 sub-squares contains 9 fields. To solve the game, the player must place a number between 1 and 9 to every field. But there are some constraints. In each row and each column no number can be repeated, they must appear exactly once. The same applies for sub-squares, in each sub-square all the numbers in [1,9] must appear exactly once.

1.3 Constraint Satisfaction Problems and AC-3 Algorithm

The constraint satisfaction problems are interested in a complete and consistent assignment of values to the variables. Each variable have a value assigned within the domain of the variable and each constraint must hold. For our case, each field is a variable and their domain is the numbers in the range [1,9]. The constraints are described in the previous section. In each row, column and sub-square all the numbers must appear exactly once.
These problems can be solved by using arc consistency. Arc is a part of a binary constraint, we need two arcs for one binary constraint. For instance, if there is a binary constraint for variable A and B such as $A \neq B$, then there are two arcs namely : $Arc(X, Y)$ and $Arc(Y, X)$. Here $Arc(X, Y)$ is consistent if and only if for every value in the domain of left hand side X, there is an allowed value in the domain of right hand side Y.
AC-3 algorithm is an algorithm used for these constraint satisfaction problems by using arc consistency. In AC-3 algorithm there is a queue contains all arcs. The first element of the queue is taken, and is applied to revise function. For revise function, the consistency of the arc is checked by checking the domain values of the left hand side and the values that contradicts for consistency are deleted. If there is no value remaining in the domain of the left hand side variable, then the algorithm returns false as indicating there is no valid solution for this constraint satisfaction problem. If there is a reduction in the domain, then the algorithm adds the arcs which contains the left hand side of initial arc as the right hand side of the arc to be added if it is not in the queue already. The algorithm continues as taking the next element from the queue, until the queue is over. If the queue is empty, the algorithm stops. The verbal expression can be hard to be interpreted. The AC-3 algorithm is explained in details in next sections.

2 Design

In this section we will explain the design of new features for the given classes and new classes. Also we will explain how the constraints and arcs are initialized and what is the design of AC-3 algorithm in details.

2.1 Design of Field Class

The Field class was given half-implemented. This class is used for containing the attribute values of every field in Sudoku. There are attributes such as domain and neighbour list. The functions about domain and domain size were also given. But for neighbours of a field, the function *addNeighbours*() was not implemented and it was one of the tasks in this assignment. Its design is illustrated in section 2.3.
For the Field class we have to re-design it by using more attributes and suitable functions for AC-3 algorithm. In the initial version of Sudoku, some fields are already initialized with a value and since their domain contains only that value, they are finalized. For AC-3 algorithm they can kept away from our constraints i.e. there is nothing to do with the finalized fields since their domain size is already 1 and we can not change them. The only important thing about them is their passive effect on constraints. The neighbours of these finalized fields, the fields that are in the same sub-square, same row or column, are affected by finalized fields, but they can't make a change in the domain of finalized fields. So it is important to make a distinction between initialized(thus finalized) and uninitialized fields. For this aim, a boolean attribute can be added to the Field Class. Another good design choice is to create two distinct constructor for initialized and uninitialized fields. By these constructors we can set separate values for the attributes of the field. The boolean attribute design with related getter function can be found in Algorithm 1. The constructors are given at the end of this section.

Algorithm 1 Boolean Attribute and Getter Function For Booelan value

```
    boolean initialized
procedure getInitial()
    return field.initialized
end procedure
```

In addition to these we should also add another needed feature, a list that contains arcs related of that field. For this design the arc list will carry all the arcs that the left hand side of the arc is the field. Also there is a need for a function that initializes this arc list. For this need, a function, namely *initializeArcs*() can be designed. For each neighbour, n of the field, f it will add an Arc(f, n) to the arc list of the field f. The pseudo-codes of designs can be found in Algorithm Algorithm 2 .
Now the constructors for each kind of field, one for uninitialized and one for initialized can be illustrated. The design of both constructors can be found in Algorithm 3.

2.2 Design of Arc Class

The Arc class is designed for containing needed attribute values for arcs. This class is used for initializing arcs. An arc is consist of two values : Left hand side variable and right hand side variable. The design for Arc class has these main attributes, namely **leftHandSide** and **rightHandSide**. The class also designed to have getter and setter functions for these main attributes. Giving the pseudo-code seems inefficient and not needed. The implementation of Arc class is given in Section 3.

Algorithm 2 Arc list, Initializing function and Getter&Setter Methods

```
List arcs
procedure initializeArcs()
    if not field.initialized then
        for n in field.neighbours do ▷ For each neighbour in the neighbour list of this field
            field.arcs.add(Arc(field, n))
        end for
    end if
end procedure
procedure getArcs()
    return arcs
end procedure
procedure setArcs(newArcs)
    arcs ← newArcs
end procedure
```

Algorithm 3 Field class Constructor Methods

```
procedure Field()                                   ▷ constructor for uninitialized fields
    for i in [1,9] do
        field.domain.add(i)
    end for
    field.initialized ← false                       ▷ Initialized value is equal to false
end procedure
procedure Field((int initialValue)                  ▷ constructor for initialized fields
    field.domain.add(initialValue) field.initialized ← true   ▷ Initialized value is equal to true
end procedure
```

2.3 Design of Constraints Initializing

For the queue in AC-3 algorithm we have to initialize the constraints. And from these constraints, we have to initialize the arcs. For this purpose, first, the neighbours of an uninitialized field must found. From these neighbours, for each neighbour, an arc must be made and be added to the arc list of each field. Later on these arc lists are to be used for the queue in AC-3 algorithm.

For the first purpose, finding the neighbours of an uninitialized field, we should look what the neighbour means and what is the algorithm to find it. A neighbour of a field is every field in the same row, column and sub-square with that field. It can be also said that every field that affects our field is the neighbour of our field. It can be understood better by the illustration in Figure 1 In the figure, blue fields represents the neighbours in the same row, yellow fields represents the neighbours in the same column and green fields represents the neighbours in the same sub-square.

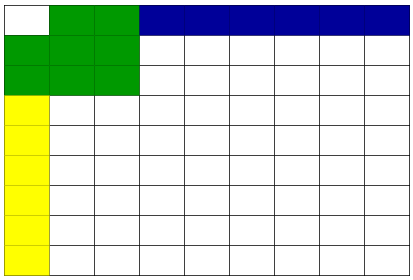

Figure 1: An example for neighbours of the first field.

To find all the neighbours of our field, an algorithm can be developed which has mainly 3 parts. The first part is about to find the neighbours in the same sub-square and the other two parts are about to find the neighbours in the same row and column. The latter two can be merged in one part. Also as we have seen in Figure 1, some of the neighbours overlaps. Then it is best to find the non-overlapping neighbours in the same square. Even though finding the neighbours in the same row and column is relatively easy, finding the neighbours in the same sub-square is hard. Because for every row-column combination in the position of the field, there are different cases. For example if we are looking for the neighbours of a field in row 1, column 1, then the (non-overlapping) neighbours will be in (2,2), (2,3), (3,2) and (3,3) or we can formalize it as (i+1, j+1), (i+1, j+2), (i+2,j+1) and (i+2, j+2) where i represents the row number of our field and j represent the column number of our field. We can see that same formalization applies for the fields in position (1,4) and (1,7) where the column number changes. Or even we can see that same applies for the fields in position (4,1) and (7,1). We can find other formalizations for row and column numbers 2,5,8 and 3,6,9. It can be realized that every time there is a difference of 3 in the numbers e.g. 3-6-9. This occurs because every sub-square has edge length of 3 fields. In Algorithm 4 the formalizations can be better understood with the pseudo-code. The algorithm returns all non-overlapping neighbours in a same sub-square of a field. It gets the row and column index of the field and the table of all Sudoku as named grid. Also the algorithm defines the first row and firs column indices as zero and the last indices are 8.
For the second part of finding neighbours algorithm, finding neighbours in the same row and same column, the designed algorithm is easier. All the fields shares the same row index with our field, but not the same column index, are the row neighbours of our field. In the same way, all the fields shares the same column index with our field, but not

Algorithm 4 Algorithm for Computing Non-overlapping Neighbours In A Sub-Square

```
procedure sameSubSquare(int i, int j, Field[][] grid )
    List subSq = ∅
    if (i + 1)%3 == 1 then                                          ▷ Rows 1,4,7
        if (j + 1)%3 == 1 then                                      ▷ Columns 1,4,7
            subSq.addAll(grid[i+1][j+1], grid[i+1][j+2], grid[i+2][j+1], grid[i+2][j+2])
        else if (j + 1)%3 == 2 then                                 ▷ Columns 2,5,8
            subSq.addAll(grid[i+1][j-1], grid[i+2][j-1], grid[i+1][j+1], grid[i+2][j+1])
        else                                                        ▷ (j+1)%3 ==0 column 3,6,9
            subSq.addAll(grid[i+1][j-1], grid[i+2][j-1], grid[i+2][j-1], grid[i+2][j-2])
        end if
    else if (i + 1)%3 == 2) then                                    ▷ Rows 2,5,8
        if (j + 1)%3 == 1 then                                      ▷ Columns 1,4,7
            subSq.addAll(grid[i-1][j+1], grid[i-1][j+2], grid[i+1][j+1], grid[i+1][j+2])
        else if (j + 1)%3 == 2 then                                 ▷ Columns 2,5,8
            subSq.addAll(grid[i+1][j-1], grid[i+1][j+1], grid[i-1][j+1], grid[i-1][j-1])
        else                                                        ▷ (j+1)%3 ==0 column 3,6,9
            subSq.addAll(grid[i-1][j-1], grid[i-1][j-2], grid[i+1][j-1], grid[i+1][j-2])
        end if
    else                                                            ▷ Rows 3,6,9
        if (j + 1)%3 == 1 then                                      ▷ Columns 1,4,7
            subSq.addAll(grid[i-1][j+1], grid[i-1][j+2], grid[i-2][j+1], grid[i-2][j+2])
        else if (j + 1)%3 == 2 then                                 ▷ Columns 2,5,8
            subSq.addAll(grid[i-1][j-1], grid[i-1][j+1], grid[i-2][j+1], grid[i-2][j-1])
        else                                                        ▷ (j+1)%3 ==0 column 3,6,9
            subSq.addAll(grid[i-1][j-1], grid[i-1][j-2], grid[i-2][j-1], grid[i-2][j-2])
        end if
    end if
    return subSq
end procedure
```

the same row index, are the column neighbours of our field. If these two neighbours are merged and returned, then the second part of the algorithm is done. Thus the algorithm for finding all neighbour fields of all fields is done, because only thing to do is merge these two returned neighbour field collection and add to the neighbour list of the field and repeat this for every field. The algorithms, for finding same row-column neighbours Algorithm 5 and for finding all neighbours of all fields, Algorithm 6 is illustrated.

Algorithm 5 Algorithm for Computing All Neighbours In Same Row-Column

```
procedure SAMEROWCOLUMN(int i, int j, Field[][] grid )
    List rowCol = ∅
    for int k = 0, k in [0,9) do
        if k≠ j then
            rowCol.add(grid[i][k])
        end if
        if k≠ i then
            rowCol.add(grid[k][j])
        end if
    end for
    return rowCol
end procedure
```

Algorithm 6 Algorithm for Computing All Neighbours Of Every Field

```
procedure ADDNEIGHBOURS(Field[][] grid )
    for int i = 0, i in [0,9) do
        for int j = 0, j in [0,9) do
            List neighbours = ∅
            neighbours.addAll(sameSubSquare(i, j, grid))
            neighbours.addAll(sameRowColumn(i, j, grid))
            grid[i][j].setNeighbours(neighbours)
        end for
    end for
end procedure
```

For the last step of constraint initialization, we should build arcs for each field and add these arcs to the arc list of each field. As it mentioned in previous sections, in Sudoku, a field has constraints with each of its neighbours where the constraint is about an inequality relationship. Thus for each uninitialized field, there will be twenty arcs, for its twenty neighbours, where the left hand side of these arcs is the field and the right hand side will be its neighbours. For instance, for Field f, and its neighbours n_1, n_2, ... ,n_{20} there will be twenty arcs $Arc_1(f, n_1)$, $Arc_2(f, n_2)$, ... , $Arc_{20}(f, n_{20})$. For this design, the Arc class and the *initializeArcs*() function in the Field class can be used. The Algorithm for *initializeArcs*() function can be found again in Algorithm 2. With this function for every field, an arc list can be created and set to the arc list attribute of every field. The implementation of arc initialization of every field is given in Section 3.4 for implementing AC-3 algorithm.

2.4 Design of AC-3 Algorithm

AC-3 is an arc-consistency algorithm. Basically, it is used to eliminate some values in the domain of variables in order to leave only one (or more) variable in the domain of

all variables.
The algorithm is consist of a while loop. The loop continues until the queue, or agenda, initially the list of all arcs, is empty. It takes the first arc from the queue and removes it, say $Arc(X_m, X_n)$ and revises this arc. Here revise operation means for $Arc(X_m, X_n)$, it checks for every value in the domain of left hand side, X_m if they satisfies the constraint with some value in the domain of X_n. If a value in the domain of X_m does not satisfied the constraint, then it is removed from the domain of X_m. For instance, for our Sudoku case, for every value x_m in the domain of field X_m, there must be some value x_n in the domain of the neighbour of X_m, X_n such that $x_m \neq x_n$. If this constraint is not satisfied, i.e. x_m is equal for all the values in domain of X_n, then x_m is removed from the domain of X_m. The design of Revise operation is explained in more details in the Section 2.4.1.
After revise operation algorithm continues with checking the domain size of X_m. If it has become zero, then the algorithm stops and returns false. Because if the domain size of one variable is zero, then it can be said that there is no appropriate value for that variable to satisfy some constraints, so there will be no valid solution for that system. If the domain size is changed but it is not zero, then algorithm adds all arcs $Arc(X_k, X_m)$ to the queue where X_k is one of the field in the neighbours of X_m except X_n if $Arc(X_k, X_m)$ is not in the queue. So here, the left hand side of our arc becomes the right hand side and all the neighbours of it, except the right hand side of our arc, becomes left hand side and all arcs added to the queue if these arcs are not in the queue already.
Algorithm gets the second arc from the queue and repeat these operations until the queue is empty, or one of the domains becomes empty. If the queue is empty, then algorithm returns true.
As mentioned in the previous sections, one of the goals of this assignment is, to test different heuristics prepared for AC-3 algorithm. For this reason, it is vital to modify the queue structure described in the design of algorithm. Instead of a normal queue, a priority queue must be implemented which will sort the elements in a defined order and will place the element with the highest priority in the first place so that the first element that pulled from the queue will be the element with the highest priority. By this structure, the heuristics will be applied on the algorithm. As it is described in Section 2.6 in details, the heuristics are used to define priority for some arcs to be evaluated by the algorithm earlier. The algorithm of AC-3 will be understand better in the pseudo-code illustration given after Section 2.4.1 in Algorithm 8. In the algorithm the constraint- arc initialization is not given, it is done by default.

2.4.1 Design of Revise Function

The basic design of Revise operation is described in the previous section. The revise operation gets an arc, e.g. (X_m, X_n). For every value in the domain of X_m it checks if there is at least one value in the domain of X_n such that satisfies the constraint. If there is one, then the algorithm stops. If there is not, i.e. for the value x_m in the domain of X_m there is no value x_n in the domain of X_n such that $x_m \neq x_n$, then x_m is deleted from the domain of X_m. Thus the domain of X_m will be $-\{x_m\}$. Algorithm does this operation for every value in the domain of X_m. The pseudo-code for Revise operation algorithm can be found in Algorithm 7.
For finding neighbours, different designs are possible with different complexities. Their implementations and designs are described in Appendix part.

Algorithm 7 Algorithm for Revise Operation

```
procedure REVISE(Arc (X_m, X_n) )
    for int x_m in X_m.domain do
        for int x_n in X_n.domain do
            if x_m ≠ x_n then ▷ If there is one x_n that satisfies the constraint x_m ≠ x_n
                break                                  ▷ then terminate the revise operation
            end if
        end for
        X_m.domain - x_m          ▷ there is no x_n that satisfies the constraint x_m ≠ x_n
    end for
end procedure
```

Algorithm 8 AC-3 Algorithm

```
procedure AC-3(Field[][] grid, Heuristic heuristic)
    PriorityQueue queue = PriorityQueue(heuristic)
    for each Field f in grid do
        queue.add(f.arcList)
    end for
    while queue ≠ ∅ do
        arc ← REMOVE-FIRST(queue)
        Revise(arc)
        if arc.getLeftHand().domainSize() == 0 then
            return false
        end if
        if arc.getLeftHand().domainSize() is changed then
            for each field X_k in neighbours of arc.getLeftHand() - arc.getRightHand()
do
                if queue does not contains X_k then
                    queue.add(X_k)
                end if
            end for
        end if
    end while
    return true
end procedure
```

2.5 Design of validSolution()

The main goal of validSolution() function is to check if AC-3 algorithm indeed find a solution or not. To check this a simple approach will work. If for every field has exactly one value left in their domains, we can say that this it is a valid solution. The pseudo-code for this design can be found in Algorithm 9.

Algorithm 9 Algorithm for validSolution

```
procedure VALIDSOLUTION(Field[][] grid)
    for each Field f in grid do
        if f.getDomainSize() ≠ 1 then
            return false
        end if
    end for
    return true
end procedure
```

2.6 Design of Heuristics

Our design for priority queue takes a heuristic as a parameter. For that reason and for the goals of the project, we should design three heuristic. The first one is a normal heuristics, the other two heuristics are designed to affect the priority of the arcs in the queue so that improve the complexity of AC-3 algorithm. The details of their designs are given in the next sub-sections.

2.6.1 Design of Normal Heuristics

The normal heuristics does no affect on the queue, it shouldn't change any priority of the arcs. For this reason it can be defined as a null value.

2.6.2 Design of MRV Heuristics

MRV (minimum remaining value) heuristics is about to choose the arc where the right hand side has the minimum domain size. For this reason we should, make a comparison between the arcs, especially right hand sides of the arcs and the priority should be given to the one with the minimum domain size.

We argue that this is a good heuristic. Because we want to eliminate the variables in the domain of left hand side. I revise operation, it has been illustrated how the variables in the domain of left hand side is eliminated. If there is no variable in the domain of right hand side to satisfy the constraint, the variable in the left hand side is eliminated. If we eliminate more variables, we can get quicker results.

2.6.3 Design of PFA Heuristics

PFA(priority for finalized arcs) heuristics is about to choose the arc where the right hand side is finalized i.e. has only one variable in its domain. For this reason we should make a control in the arcs, especially about right hand sides, and we should pick the finalized arcs first.
We argue that this is a good heuristic. Because it has the same logic behind MRV heuristics, eliminate more variables in the domain of the left hand side of the arc.

3 Implementation

In this section, the implementation of design choices are explained and illustrated.

3.1 Implementation of Changes in Field Class

There were some changes in Field class. They are new features and functions of a field, a boolean value about being initialized or not, a getter function for this value, an arc list, and getter-setter functions for this list, initializing arcs and two constructor methods for each type of field, initialized and uninitialized.
The implementation is given in below :

```
public class Field {
  private int value = 0;
  private List<Integer> domain;
  private List<Field> neighbours; //A list of all fields that this field is constrained by
  private List<Arc> arcs ; // A list of all arcs of a field where the field is the left hand side of the arcs
  private boolean initialized ; //indicates if the field is initial or not.
  /*
   * ==============
   *  CONSTRUCTORS
   * ==============
   */

  // Constructor in case the field is unknown
  Field() {
    this.domain = new ArrayList<>(9);
    for (int i = 1; i < 10; i++)
    {
      this.domain.add(i);
    }
    this.arcs = new ArrayList<>();
    this.initialized = false; // the field is unknown, uninitialized
  }

  // Constructor in case the field is known, i.e., it contains a value
  Field(int initValue) {
    this.value = initValue;
    this.domain = new ArrayList<>();
    this.domain.add(initValue);
    this.arcs = new ArrayList<>();
    // does not add arc since it is not a variable to be changed.
    this.initialized = true; //the field is known, initialized
  }
  public List<Arc> getArcs()
  {
    return arcs;
  }
  public void setArcs(List<Arc> arcs)
  {
    this.arcs = arcs;
  }
  public void initializeArcs()
  {
```

```
43      if(!this.initialized) // if the field is unknown
44      {
45        for(Field n : this.neighbours)
46        {
47
48          this.arcs.add(new Arc(this, n)); // Add each neighbour as an arc
49        }
50      }
51    }
52
53    public boolean getInitial()
54    {
55      return this.initialized;
56    }
57
58      }
```

3.2 Implementation of Arc Class

The Arc class is implemented as mentioned in Section 2.2. Although it is an entire class, and maybe it is wrong to illustrate it in this report, I will present it because of its brevity. It can be seen that indeed the arc class is based on two important attributes, the left-Handside and the rightHandside. All other methods and data structures are in a helper position for these attributes. These methods and data structures are getter and setter methods for each attribute and also a constructor for the class. The implementation is illustrated below.

```
1   public class Arc {
2
3       private Field leftHandSide ;
4       private Field rightHandSide;
5
6       public Arc(Field leftHandSide, Field rightHandSide)
7       {
8       Implementation}
9
10      public Field getLeftHandSide()
11      {
12          return this.leftHandSide;
13      }
14
15      public Field getRightHandSide()
16      {
17          return this.rightHandSide;
18      }
19
20
21  }
```

3.3 Implementation of Constraint Initializing

The design of constraint initializing is described in Section 2.3. For implementing this design by using Java, again we will follow the design method and implement it in three parts. The first part is about implementation for the non-overlapping neighbours in the same sub-square. The Algorithm 4, offers a good design choice and it is very neat to implement it in Java. The implementation is given by *sameSubSquare* function below. The function gets two integers, as index numbers and the Sudoku grid, consists of all 81 fields. It returns a list of all non-overlapping neighbours of the field with index(i,j) in the grid.

```
1  private static List<Field> sameSubSquare(int i, int j, Field[][] grid)
2    {
3          List<Field> inSameSubSquare = new ArrayList<>();
4      if ((i+1)% 3 ==1 ) // row 1,4,7
5      {
6        if((j+1) % 3 ==1) //column 1,4, 7
7        {
8         inSameSubSquare.addAll(Arrays.asList(grid[i+1][j+1], grid[i+1][j+2],
9         grid[i+2][j+1], grid[i+2][j+2]));
10       }
11       else if ((j+1) % 3 ==2) //column 2,5,8
12       {
13        inSameSubSquare.addAll(Arrays.asList(grid[i+1][j-1], grid[i+2][j-1],
14        grid[i+1][j+1], grid[i+2][j+1]));
15       }
16       else //(j+1)%3 ==0   column 3,6,9
17       {
18        inSameSubSquare.addAll(Arrays.asList(grid[i+1][j-1], grid[i+2][j-1],
19        grid[i+2][j-1], grid[i+2][j-2]));
20       }
21     }
22     else if ((i+1)%3 == 2) // row 2,5,8
23     {
24       if((j+1) % 3 ==1) //column 1,4, 7
25       {
26        inSameSubSquare.addAll(Arrays.asList(grid[i-1][j+1], grid[i-1][j+2],
27        grid[i+1][j+1], grid[i+1][j+2]));
28       }
29       else if ((j+1) % 3 ==2) //column 2,5,8
30       {
31        inSameSubSquare.addAll(Arrays.asList(grid[i+1][j-1], grid[i+1][j+1],
32        grid[i-1][j+1], grid[i-1][j-1]));
33       }
34       else //(j+1)%3 ==0   column 3,6,9
35       {
36        inSameSubSquare.addAll(Arrays.asList(grid[i-1][j-1], grid[i-1][j-2],
37        grid[i+1][j-1], grid[i+1][j-2]));
38       }
39     }
40     else // row 3,6,9
41     {
42       if((j+1) % 3 ==1) //column 1,4, 7
43       {
```

```
      inSameSubSquare.addAll(Arrays.asList(grid[i-1][j+1], grid[i-1][j+2],
      grid[i-2][j+1], grid[i-2][j+2]));
     }
     else if ((j+1) % 3 ==2) //column 2,5,8
     {
      inSameSubSquare.addAll(Arrays.asList(grid[i-1][j-1], grid[i-1][j+1],
      grid[i-2][j+1], grid[i-2][j-1]));
     }
     else //(j+1)%3 ==0   column 3,6,9
     {
      inSameSubSquare.addAll(Arrays.asList(grid[i-1][j-1], grid[i-1][j-2],
      grid[i-2][j-1], grid[i-2][j-2]));
     }
   }
   return inSameSubSquare;
 }
```

The second part of the design is about finding the other neighbours, the neighbours in the same row and column, and merging all these neighbours in one list to add the neighbour list of every field. Again for this part, we will implement our design by using Java. For finding the neighbours in the same row and same column, we will implement Algorithm 5 where it will get index numbers of the field and the grid. It will return the list of all neighbours in the same row and column of the field (i,j). For the implementation of merging all these neighbours, same row, same column and same sub-square, we will use Algorithm 6. It will initialize a list and add the return values of $sameSubSquare()$ and $sameRowColumn()$ to this list. Finally it will use $setNeighbour()$ function to set the neighbour list of every field to this list. Alternatively a function can be created in Field class, which is used to append the neighbour list. By this way, there is no need to initialize a list, appending the neighbour list by this appending function would be enough. But we didn't implement it in that way, because of not needing it.
The implementations are illustrated below. The first implementation is for Algorithm 5 and the second one is for Algorithm 6.

```
private static List<Field> sameRowColumn ( int i, int j, Field[][] grid )
  {

    List<Field> inSameRowColumn = new ArrayList<>();
    for (int k =0; k<9 ; k++ ) //fields in the same row and same column
    {
      if(k!= j)
      {
        inSameRowColumn.add(grid[i][k]);
      }
      if(k!=i)
      {
        inSameRowColumn.add(grid[k][j]);
      }
    }
    return inSameRowColumn;
  }
```

```
private static void addNeighbours(Field[][] grid) {
    for (int i =0; i< 9; i++)
    {
      for(int j=0; j<9 ; j++)
      {
        List<Field> neighbours = new ArrayList<>();
        neighbours.addAll(sameSubSquare(i, j, grid));
        neighbours.addAll(sameRowColumn(i, j, grid));

        grid[i][j].setNeighbours(neighbours);

      }
    }
  }
```

The last part of the constraint initialization is can be done by implementing Algorithm2. As mentioned in Section 2.3, the implementation of initializing all arcs for every field is given in the next section. The illustration for implementing *initializeArcs*() is given in the implementation of Field class in Section 3.1.

3.4 Implementation of AC-3 Algorithm

In the assingment, we are expected to implement the AC-3 algorithm in a function named *solve*(). Since the implementation of AC-3 algorithm is the core of this assignment, the implementation is given in steps. The algorithm will be implemented in the same principles of the design, given in Section 2.4.

3.4.1 Implementation of Constraint-Arc Initialization

For AC-3 algorithm the contraints and arc lists must be initialized. It is an important preparation for the algorithm. The best implementation is to use *initializeArcs*. After every field has an arc list, the initialized fields will have an empty list. We can add every arc to our arc list. Then by copying this arc list, we can create an initial agenda, a priority queue. For priority queue, we can use *Comparator* and *PriorityQueue* data structures in Java 11. In the implementation there can be a integer variable *ct*. It is used for the complexity of the algorithm. It is explained in Section 4. The implementation is given below.

```
// For every field initialize its arcs (if it is a constant value, i.e. known then its arc list is empty)
    List<Arc> arcs =new ArrayList<>();// arc list of the game
    int ct = 0;
    for (Field[] row : sudoku.getBoard())
    {
      for (Field field :row)
      {
        field.initializeArcs();
        arcs.addAll(field.getArcs());
      }
    }

    //Then add these arcs to the agenda.
    Queue<Arc> agenda = new PriorityQueue<>(new Normal_Comparator());//agenda of arcs
    agenda.addAll(arcs);
```

3.4.2 Implementation of First Part of AC-3

In the first half of the while loop, we get the first element from the priority queue with *poll*() function that pops the first element and removes it from the queue. The domain size of the left hand side of the arc is stored for comparison in the next steps, if there is any changes in the domain size after revise operation. After revise, we check the domain size and if it is zero, then the algorithm returns false and we understand that there is no valid solution possible. The iteration of *ct* value is for complexity measurement and it is discussed again in Section 4. The implementation of revise operation is given in Section 3.4.4. The implementation for this part is illustrated below.

```
//while agenda is not empty
while(!agenda.isEmpty())
    {
      ct++;
      //take the first element first arc
      Arc arc = agenda.poll();
      int arcLeftDomainSize = arc.getLeftHandSide().getDomainSize();
      //revise the arc
      revise(arc);
      // if the new size of domain is zero return false
      if(arc.getLeftHandSide().getDomainSize() == 0)
      {
        return false;
      }
```

3.4.3 Implementation of Second Part of AC-3

The second part of the algorithm is concerned about the change in the domain size of the left hand side, but the size is not equal to zero. In this case, for every new arc $(X_k, X_m$ where X_k is one of the neighbours of X_m if this arc is not in the queue, it is added.
The implementation for finding neighbours could be interpreted a bit strange. The reason for this is in the implementation *getOtherNeighbours*() method is not used. At the first days of the assignment, I didn't realize why should I use this method. Rather than using it, I have find my own solution : I check every arc in the initial arc list, where the right hand side of the the arc is the same field as the left hand side of the arc we are using for revise operation. If this condition and the other condition, this new arc shouldn't be in our queue, we can add this new arc to our queue. After that we continue to the while loop and if the queue is empty we return true.The implementation is given below.

```
// if the domain size has changed
      if(arcLeftDomainSize != arc.getLeftHandSide().getDomainSize() )
      {

        //for all arcs contain the left hand side as the right hand side add to the list
        for(Arc newArc : arcs)    //for every arc in the arc list
        {
```

```
          if(arc.getLeftHandSide() == newArc.getRightHandSide() && (!agenda.contains(newArc)))// if the lefthandsid
          {                                                    // and if that arc is not in the agenda
            agenda.add(newArc); //add that arc to the agenda
          }
        }
      }

    }
    System.out.println(ct);
    return true;
```

After a while working on the assignment, I have understood how to use $getOtherNeighbours()$ method, but its complexity is far higher than what I have implemented. The complexity of this alternative with the given method is measured in Section 4 and the design and the implementation of this alternative solution is given in the last section, Section 7.

3.4.4 Implementation of Revise Operation

The revise operation is implemented as described in Section 2.4.1. The problem was, unfortunately, Java does not give permission to remove the values, because of concurrency. The solution I have found was, initialize a list, to store values to be removed, then for every value needed to be removed from the domain, add to that list. At the end of revise operation, remove from domain all these values in the list in a for loop works. It could be better understood by illustrating the implementation.

```
public void revise(Arc arc)
  {
    Field leftHand = arc.getLeftHandSide();
    Field rightHand = arc.getRightHandSide();
    List<Integer> toRemove = new ArrayList<>();
    for (int valueLeft : (leftHand.getDomain())) //for all values in the domain of left hand side
    {
      for(int valueRight : rightHand.getDomain()) //there exists a value meets the arc in the domain of right hand
      {
        if (valueLeft != valueRight)
        {
          break; // if there exists a value then break the inner loop
        }
        // if there is left no value that meets the arc in the domain of right hand side
        // which means the value is the last element of the domain and it does not meet the arc
        else if (rightHand.getDomain().indexOf(valueRight)== rightHand.getDomainSize()-1)

        {
          toRemove.add(valueLeft); //add to the list of values to be removed
        }
      }
    }
    for(Integer i : toRemove)
    {
      leftHand.removeFromDomain(i);//remove that value from the domain of left hand side.
    }
  }
```

3.5 Implementation of validSolution() Algorithm

The implementation of *validSolution* is consists of the design of it. As it was described in Section 2.5, it checks domain sizes of every field and if they are all exactly one, then the solution is valid. If not, the implementation returns false. Below the implementation is given.

```
public boolean validSolution() {
    for (Field[] row : sudoku.getBoard())
    {
      for (Field field :row)
      {
        if(field.getDomainSize()!=1)
        {
          return false;
        }
      }
    }
    return true;
  }
```

3.6 Implementation of Heuristics

For the implementations of the heuristics, I have used Comparator structures that imported from Java packages.

3.6.1 Implementation of Normal Heuristics

For the normal heuristics, I have implemented default comparator. The implementation is illustrated below.

```
class MRV_Comparator implements Comparator<Arc>
{

  @Override
  public int compare(Arc arc1, Arc arc2) {
    if(arc1.getRightHandSide().getDomainSize() < arc2.getRightHandSide().getDomainSize())
    {
      return -1;
    }
    else if (arc1.getRightHandSide().getDomainSize() > arc2.getRightHandSide().getDomainSize())
    {
      return 1;
    }
    else
    {
      return 0;
    }
  }
}
```

3.6.2 Implementation of MRV Heuristics

For comparator of MRV heuristics, I have defined the *compare* function such that it will create an ascending order based on the domain size of the right hand sides. The implementation is illustrated below.

```
class Normal_Comparator implements Comparator<Arc>
{

  @Override
  public int compare(Arc arc1, Arc arc2) {
    return 0;
  }
}
```

3.6.3 Implementation of PFA Heuristics

For comparator PFA heuristics, I have defined the *compare* function such that it will prioritize the arcs where the right hand sides have finalized domains. There is an important remark for the implementation. To compare the arcs, I have checked in three conditions. If the first arc is finalized, but the second is not I don't change anything. If the first arc is not finalized but the second one is, then I move the second to forward. If they are both finalized or both not finalized, I change nothing. Below the implementation is illustrated.

```
class PFA_Comparator implements Comparator<Arc>
{

  @Override
  public int compare(Arc arc1, Arc arc2) {
    if(arc1.getRightHandSide().getDomainSize()==1 && arc2.getRightHandSide().getDomainSize()!=1)
    {
      return -1;
    }
    else if (arc1.getRightHandSide().getDomainSize()!=1 && arc2.getRightHandSide().getDomainSize()==1)
    {
      return 1;
    }
    else
    {
      return 0;
    }
  }
}
```

4 Complexity

In this section, the complexity of AC-3 and the heuristics are explained. The theoretical complexity of AC-3 is not given but the focus is on the practical complexity.

4.1 Measure System for Practical Complexity of AC-3 and Heuristics

For measuring practical complexity of AC-3 algorithm and heuristics, there are some implementation methods, as illustrated but not explained in Section 3.4. For complexity measures, a counter is placed in the implementation of AC-3 algorithm. The counter is activated and iterated every time a new arc is going to be popped out of the queue. So the iteration lasts until the queue is empty and it shows how many times the algorithm works, i.e. pops out an arc, revise it and check for other neighbours. Even though we are not interested in exact numbers, this complexity measure will help to make comparison between heuristics and to prove the improvement made by application of heuristics. In the next sections, the complexity measurement results are given for normal heuristics (default AC-3), MRV and PFA heuristics for each 5 Sudoku file. The analysis of the complexity is given at the end of the section.

4.2 Practical Complexity of AC-3

The AC-3 algorithm with normal heuristics are run for each Sudoku file. The complexity results are given in the table below.

Complexity Results					
Heuristics Name	Sudoku 1	Sudoku 2	Sudoku 3	Sudoku 4	Sudoku 5
Normal(default) Heuristics	3602	3113	2922	2105	2930

4.3 Practical Complexity of AC-3 with Heuristics

The AC-3 algorithm with other heuristics are run for each Sudoku file. The complexity results are given in the table below.

Complexity Results					
Heuristics Name	Sudoku 1	Sudoku 2	Sudoku 3	Sudoku 4	Sudoku 5
MRV Heuristics	1398	1514	1261	940	1277
PFA Heuristics	1798	1853	1495	940	1748

4.4 Complexity Analysis

The results can be interpreted such that both heuristiscs, MRV and PFA are improving AC-3 alogirthm in the context o practical complexity. They make the same operation nearly 2-3 times faster than classical algorithm does.

Also it can be seen that MRV performs better than PFA. The reason for this is simple. PFA gives priority to only the arcs where the right hand side has 1 element in their domain. If they don't have 1 element left, then heuristic does not change the order. On the other hand, MRV sorts the size of the domain, so that even the variable has 2 elements in its domain, its priority is higher than the one with 3 elements in its domain. Briefly, while MRV gives priority to the arc with least domain size, PFA does not, it only gives priority to the finalized arcs, with domain size is equal to one, which leads, the arcs with less domain size should wait for the arcs with more domain size.

5 Testing

For testing our AC-3 algorithm and heuristics, we will use two measures. One of them is a visual measure, it is concerned about the output of the program and checks if every square in the Sudoku table is filled or not. The other one is an implementation and one of the goals of the project, *validSolution*() method. If the method returns true, then we can say AC-3 algorithm also gives valid solutions, not only random numbers.The second measure is more reliable than the other one.

The first Sudoku, Sudoku 1, gives both a valid visual effect and a valid solution. It completes the Sudoku table with correct values.
The second Sudoku, Sudoku 2, gives both a valid visual effect and a valid solution. It completes the Sudoku table with correct values.
The third Sudoku, Sudoku 3, does not give both a valid visual effect and a valid solution. It does not complete the Sudoku table, and returns false for *validSolution*() method.
The fourth Sudoku, Sudoku 4, does not give both a valid visual effect and a valid solution. It does not complete the Sudoku table, and returns false for *validSolution*() method.
The fifth Sudoku, Sudoku 5, gives both a valid visual effect and a valid solution. It completes the Sudoku table with correct values.

The problem with Sudoku 3 and Sudoku 4 is not about the correctness of AC-3 algorithm. Even though the AC-3 algorithm works correctly, at the end there are some variables left with the domain size is more than 1. In that case, the best way to complete Sudoku tables is designing and implementing a backtracking algorithm with possible heuristics such as degree heuristics or minimum remaining value heuristics. By this way, the Sudoku tables can be completed. Since the backtracking algorithm is out of scope for this project, it is not designed nor implemented.

6 Discussion and Results

In this section, we will evaluate the results in Section 4 and Section 5.
In Section 4, we have found that the AC-3 algorithm performs better with MRV and PFA heuristics. The reason for that, is the heuristics prioritize the arcs where the right hand side is either finalized i.e. the domain size is 1 or sorts in an ascending order based on the domain size. By this heuristics, the algorithm eliminates more values per iteration and the algorithm completes the the queue nearly 2-3 times faster. So we can say AC-3 algorithm performs better with heuristics.

Another important result is, MRV heuristics performs better than PFA does. The main reason behind this is, MRV gives priority to every arc based on the domain size of the right hand side, while PFA only gives priority to arcs where th domain size of the right hand side is one. Because of this, some arcs which should be prioritized more than the others, since they have less domain size, are not prioritized enough. So we can say that MRV heuristics is more logical and performs better than PFA heuristics.

The last result is about AC-3 algorithm and its validity. Although we are sure about the correctness of AC-3 algorithm, it does not always return complete results. Sometimes, the queue gets empty and that is why AC-3 terminates, but some variables has multiple values in their domains. In this situation a good approach is to design and implement a backtracking algorithm with appropriate heuristics.

7 Appendix

In this section we will explain an alternative implementation choice for finding and adding the neighbours of a left hand side of an arc. The idea of this implementation is given in Section 3.4.3. To remind briefly, I have implemented finding other neighbours as described in Section 3.4.3. After a while, I have discovered *getOtherNeighbours*() method given in Field class. This method could be used for finding other neighbours operation. I have made an implementation that is illustrated below.

```
for(Field f : arc.getLeftHandSide().getOtherNeighbours(arc.getRightHandSide()))
        {
          Arc newArc = new Arc(f, arc.getLeftHandSide());
          if(!agenda.contains(newArc))
          {
            newArcs1.add(newArc);
          }
```

As it can be seen in the illustration, the principle is to get every neighbour of the left hand side by using *getOtherNeighbours*() method. Then create a new arc with this found field and add to the queue if it is not already in the queue.
The problem with this implementation is about the complexity. It increases complexity somehow. A bit improvement can be made by adding another boolean checking value such as if the found neighbour f is not finalized field. The logic behind this improvement is, since the domain size of a finalized field is equal to one, there is no need to apply revise on this field. On the other hand, the improvement is not so good. So I have decided not to implement this alternative and kept the original one.
The whole implementation of this project can be found in the link :
gitlab/ege.sari/sudoku