# AI PRINCIPLES & TECHNIQUES

Sudoku Assignment

Burak Balci s1073728
Radboud University 24/11/2023

## Contents

**1 Introduction** **2**
1.1 Sudoku . . . . . . . . . . 2
1.2 AC-3 Algorithm . . . . . . . . . . 2
1.3 Goals . . . . . . . . . . 2

**2 Design** **2**
2.1 Design of the App Class . . . . . . . . . . 2
2.2 Design of the Field Class . . . . . . . . . . 2
2.3 Design of the Sudoku Class . . . . . . . . . . 3
2.4 Design of the Game Class . . . . . . . . . . 4
2.4.1 Design of the AC-3 Algorithm . . . . . . . . . . 4
2.4.2 Design of the Verification Function . . . . . . . . . . 5
2.5 Design of the Arc Class . . . . . . . . . . 5
2.6 Design of the Heuristics . . . . . . . . . . 5
2.6.1 Design of the Simple Heuristic . . . . . . . . . . 5
2.6.2 Design of the Minimum Remaining Values Heuristic . . . . . . . . . . 6
2.6.3 Design of the Priority to Constraints With Finalized Arcs Heuristic 6

**3 Implementation** **6**
3.1 Implementation of the App Class . . . . . . . . . . 6
3.2 Implementation of the Field Class . . . . . . . . . . 6
3.3 Implementation of the Sudoku Class . . . . . . . . . . 7
3.4 Implementation of the Game Class . . . . . . . . . . 8
3.4.1 Implementation of the `solve()` Function . . . . . . . . . . 8
3.4.2 Implementation of the `revise()` Function . . . . . . . . . . 9
3.4.3 Implementation of the `validSolution()` Function . . . . . . . . . . 10
3.5 Implementation of the Arc Class . . . . . . . . . . 10
3.6 Implementation of the Heuristics . . . . . . . . . . 11
3.6.1 Implementation of the Simple Heuristic . . . . . . . . . . 11
3.6.2 Implementation of the Minimum Remaining Values Heuristic . . 11
3.6.3 Implementation of the Priority to Constraints With Finalized Arcs Heuristic . . . . . . . . . . 12

**4 Tests** **12**

**5 Complexity** **13**
5.1 Complexity Results of the AC-3 Algorithm With/Without Heuristic . . 13
5.2 Complexity Analysis . . . . . . . . . . 13

**6 Conclusion** **13**

## 1 Introduction

In this section, small description of Sudoku and goals of the assignments are given.

### 1.1 Sudoku

Sudoku is a game that contains 81 fields in a 9×9 grid, divided to nine 3×3 subgrids. The goal is to fill the grid so that each row, each column, and each of the nine ×3 subgrids contain all the numbers from 1 to 9 without repetition.

### 1.2 AC-3 Algorithm

AC-3 (Arc-Consistency Algorithm 3) is a constraint satisfaction algorithm used in artificial intelligence and constraint programming. Its primary purpose is to reduce the search space for constraint satisfaction problems by using arc consistency. The algorithm works by examining the constraints between variables, removing inconsistent values from their domains.

### 1.3 Goals

The primary goals of this assignment are:

- Understand the AC-3 algorithm.
- Find a good representation and initialize all constraints.
- Implement AC-3 algorithm and a function that can verify the output.
- Implement multiple heuristics.
- Experiment with different heuristics and compare complexity.

## 2 Design

In this section, we will look through the design of each class. Changes and additions in existing classes and newly created classes.

### 2.1 Design of the App Class

I made some changes in the `App` class to make testing easier for me. Now it asks the user for an `int` input to choose which Sudoku to solve, so that way I don't have to change `filePath` every time I want to test a new Sudoku file. Although it is not relevant for the purpose of the assignment, it makes testing way easier and faster. Implementation of it can be found in **Section 3.1**.

### 2.2 Design of the Field Class

The `Field` class was already mostly designed and implemented. There are three attributes given, value, domain list and neighbours list, and two constructors for both when field is known and unknown. There are functions for the given attributes like getter and setter functions for value and neighbours, and getter function, getting size function and remove function for domain. But since I needed to add a class for arcs, I had to make a few changes.
First thing I did was to add two attributes. List of arcs and a boolean named `added`.

Arcs are designed in such way that the left hand side will be the current field and the right hand side will be the neighbours of that field. Then I needed a function to add arcs to the list. I used `boolean added` here since if a field is already known, we don't need arcs of that field. With that, the Field class was done.
Here is the pseudo code for `addArcs()` function:

```
function addArcs():
    if not this.added then:
        for each Field f in this.neighbours do:
            this.arcs.add(new Arc(this, f))
```

Implementation of this and changes made in constructors can be found in **Section 3.2**.

## 2.3 Design of the Sudoku Class

`Sudoku` class only has one attribute and it is a 2D Field array named board. Of course, this is our 9×9 grid. Class has a `toString()` functions that prints the Sudoku board, also function `readsudoku()` which reads the Sudoku from a `.txt` file to our grid.
To solve a Sudoku using AC-3 algorithm, we need to find neighbours of each field so we can initialize the arcs for each field. For each neighbour, an arc will be created 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.
In Sudoku, each row, each column, and each of the nine 3×3 subgrids contain all the numbers from 1 to 9 without repetition. That means that a fields neighbours are other fields that are in the same row, same column and same subgrid.
There is a function called `addNeighbours()` which adds every neighbour of each field to their neighbour list. Here is the pseudo code:

```
function addNeighbours(grid):
    for each f in grid do:
        neighbours = rowNeighbours(f) + columnNeighbours(f) +
            squareNeighbours(f)
        f.setNeighbours(neighbours)
```

Basically, a field list `neighbours` is created, and all the neighbours of a field are added to that list for all fields. I used three functions for row, column and square neighbours. Row and column functions share a similar logic. We iterate through the row/column of a field and return every field except the field we are finding neighbours of. Here are the pseudo codes for both functions:

```
function rowNeighbours(grid, field):
    for row 0 to 9 do:
        if row != field.getRow() then:
            rowList.add(grid[field.getCol()][row])
    return rowList

function colNeighbours(grid, field):
    for col 0 to 9 do:
        if col != field.getCol() then:
            colList.add(col]grid[field.getRow())
    return colList
```

Square neighbours function is a bit more complex. The logic is to find the top-left field of the subsgrid that given field belongs to. For example, if given field is `grid[5][5]`, then it should belong to the subgrid that's right in the middle, and top-left of that subgrid is `grid[3][3]`. Now a nested for loop, iterating 3 times for row and column, can add all the fields in the subgrid. To find the starting row and starting column, we can simply divide the indices by 3, and then multiply by 3. That way, when we divide an indice by 3, we will find which sub-row/column that our field is in, and we multiply it by 3, we find the starting indices. Here is the pseudo code:

```
function squareNeighbours(grid, field):
    startRow = (i / 3) * 3
    startCol = (j / 3) * 3
    for row startRow to startRow + 3 do:
        for col startCol to startCol + 3 do:
            if row != field.getRow() and col != field.getCol() then:
                sameSquare.add(grid[row][col])
    return sameSquare;
```

### 2.4 Design of the Game Class

This class is where AC-3 algorithm and the function that verifies the output is implemented. I will explain both separately. Before that, the class contains only one variable, Sudoku, and there is a given function `showSudoku()` which simply prints the Sudoku. Other functions are implemented by me.

#### 2.4.1 Design of the AC-3 Algorithm

AC-3 is an Arc-Consistency Algorithm. Main goal is to eliminate value possibilities for each field, so removing values from their domain. If Sudoku is solvable, each field will have domain size of 1 in the end.
The algorithm first finds arcs for each field on the board, and adds all to a queue and a list. Then until the queue is empty, takes the first arc from the queue and removes it and revises this arc.
Revise operation takes an arc, checks if any value in the domain of the left hand side is equal to a value from the domain of the right hand side. If there is a value that is equal to every value of the right hand side, then that value is removed from the domain of the left hand side. For that to happen, domain size of the right hand side has to be 1. That means we only remove a value from left hand side when a field is known.
Here is the pseudo code for `revise()` function:

```
function revise(arc(Xm, Xn)):
    for each xm in Xm do:
        found = false
        for each xn in Xn do:
            if xm != xn then:
                found = true
                break
        if not found then:
            Xm.remove(xm)
```

After revising the arc, algorithm checks if there is an inconsistency first. That happens when domain size of a field is 0 and algorithm returns false in that case. If there is no inconsistency, it checks if the domain size of the left hand side has changed. If that's the case, adds every arc that has the same right hand as the main arcs left hand to queue if it's not already there. The main reason for that is to make sure there are no errors in the progress. For example, if we have an arc $A = B$, we want to have $B = A$ in the queue because if domain of $B$ changes, we also need to update domain of $A$. Of course that kind of an example not exactly applies to Sudoku, but that's just a small example to make things more clear.
Until the queue is empty, algorithm keeps doing that for each arc in the queue. When it's empty, algorithm returns true.
Here is the pseudo code for AC-3 algorithm:

```
function AC-3(grid, heuristic):
    PriorityQueue queue = PriorityQueue(heuristic)
    for each f in grid do:
        queue.add(f.getArcs)
    while queue is not empty do:
        arc(Xm, Xn) = removeFirst(queue)
        revise(arc)
        if new size of Dm = 0 then:
            return false
        if Dm has been changed then:
            for each Xk in {neighbours of Xm} - Xn do:
                if (Xk, Xm) not in queue then:
                    queue.add((Xk, Xm))
    return true
```

To test different heuristics, priority queue has been used. The reason for that is to sort the elements in a defined order and place the element with the highest priority in the first place in the queue. There are three heuristics implemented, design of these can be found in **Section 2.6**, and implementation can be found in **Section 3.6**.

### 2.4.2 Design of the Verification Function

Purpose of this function is to check if AC-3 algorithm did find the solution or not. Simply, if domain size of each field is not 1, then it's not a valid solution. Here is the pseudo code:

```
function validSolution(grid):
    for each f in grid do:
        if f.getDomainSize() != 1 then:
            return false
    return true
```

## 2.5 Design of the Arc Class

The `Arc.java` class contains two attributes, left hand side and right hand side, both being a field. Also there is a constructor, along with getter and setter functions. Purpose of the class is to define the interaction and constraint between the left hand side field and the right hand side field.

## 2.6 Design of the Heuristics

In AC-3 algorithm, priority queue is used, and it takes a heuristic as parameter. So depending on the heuristic used, queue priority can be changed.
I decided to create a `Heuristic` class that implements `Comparator<Arc>`. There are 3 classes that extending the `Heuristic` class. I designed 2 heuristics that are mentioned in the assignment pdf, and added one which doesn't prioritize any arcs, hence doesn't make any change to the queue.
**Note:** I also tried to implement Degree Heuristic and Least Constraining Value Heuristic, however I couldn't managed to do it. Probably I needed to add extra function(s) to count constraints, but I didn't have time to do that. That's why I only have heuristics mentioned in the assignment pdf.

### 2.6.1 Design of the Simple Heuristic

There is no prioritization on any arcs. Just returns 0 for its compare function.

### 2.6.2 Design of the Minimum Remaining Values Heuristic

Minimum Remaining Value Heuristic picks the variable that has the fewest (remaining) possible values. Basically, arcs with a smaller domain size in their right hand sides will be prioritized.

### 2.6.3 Design of the Priority to Constraints With Finalized Arcs Heuristic

Priority to Constraints With Finalized Arcs Heuristic picks the variable that is finalized. Basically, arcs with a domain size of 1 in their right hand sides will be prioritized.

## 3 Implementation

In this section, implementation of classes, functions are explained. Documentation of the code will not be included here, however they are added to the actual code.

### 3.1 Implementation of the App Class

Not much to mention here. I just decided to go with a slightly different design in the `main()` function to make testing easier for me, but `start()` function is the same. Here is the implementation of `main()`:

```java
public static void main(String[] args) throws Exception {
    boolean stop = false;
    Scanner scanner = new Scanner(System.in);
    while (!stop) {
        String filePath = "";
        System.out.println("Select a sudoku file(1, 2, 3, 4, 5, anything
            else to stop):");
        System.out.println("""
            1- Sudoku1.txt
            2- Sudoku2.txt
            3- Sudoku3.txt
            4- Sudoku4.txt
            5- Sudoku5.txt""");
        String s = scanner.nextInt();
        switch (s) {
            case "1" -> filePath = "Sudoku1.txt";
            case "2" -> filePath = "Sudoku2.txt";
            case "3" -> filePath = "Sudoku3.txt";
            case "4" -> filePath = "Sudoku4.txt";
            case "5" -> filePath = "Sudoku5.txt";
            default -> stop = true;
        }
        if (!stop) {
            start(filePath);
        }
    }
}
```

### 3.2 Implementation of the Field Class

Most of the `Field` class was already implemented. Newly added features are 2 attributes, an arc list and a boolean value to check if arc list is already initialized or not, getter function for arc list and a function that adds fields arcs to the arc list. Also there are some changes in the constructors.
Here is the implementation of the `Field` class:

```java
public class Field {
  private int value = 0;
  private List<Integer> domain;
  private List<Field> neighbours;
  private List<Arc> arcs;
  private boolean added;

  Field() {
    this.domain = new ArrayList<>(9);
    for (int i = 1; i < 10; i++)
      this.domain.add(i);
    this.arcs = new ArrayList<>();
    this.added = false;
  }

  Field(int initValue) {
    this.value = initValue;
    this.domain = new ArrayList<>();
    this.domain.add(initValue);
    this.arcs = new ArrayList<>();
    this.added = true;
  }

  public List<Arc> getArcs() {
    return arcs;
  }

  public void addArcs() {
    if (!this.added) {
      for (Field f : this.neighbours) {
        this.arcs.add(new Arc(this, f));
      }
    }
  }
}
```

## 3.3 Implementation of the Sudoku Class

For `Sudoku` class, I needed to implement adding neighbours for each field. I implemented this in `addNeighbours()` function with using three helper functions for finding neighbours in row, column, and subgrid.
Function `addNeighbours()` takes a grid as parameter and iterates through each field and adds their neighbours to its neighbour list. Here is the implementation of it:

```java
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(rowNeighbours(grid, i, j));
            neighbours.addAll(columnNeighbours(grid, i, j));
            neighbours.addAll(squareNeighbours(grid, i, j));
            grid[i][j].setNeighbours(neighbours);
        }
    }
}
```

The function `rowNeighbours()`, `columnNeighbours()` and `squareNeighbours()` all take a grid, and indices of a field. We are iterating through all the fields in `addNeighbours()` function and calling each of these functions by giving each fields indices along with the Sudoku grid we are working on. They all return a list of neighbours and will be added to fields neighbour list. Here is the implementation of all three functions:

```java
private static List<Field> rowNeighbours(Field[][] grid, int i, int j) {
    List<Field> sameRow = new ArrayList<>();

    for (int x = 0; x < 9; x++) {
        if (x != j) {
            sameRow.add(grid[i][x]);
        }
    }
    return sameRow;
}

private static List<Field> columnNeighbours(Field[][] grid, int i, int j)
    {
    List<Field> sameColumn = new ArrayList<>();

    for (int x = 0; x < 9; x++) {
        if (x != i) {
            sameColumn.add(grid[x][j]);
        }
    }
   return sameColumn;
}

private static List<Field> squareNeighbours(Field[][] grid, int i, int j)
    {
    List<Field> sameSquare = new ArrayList<>();
    int startRow = (i / 3) * 3;
    int startCol = (j / 3) * 3;

    for (int row = startRow; row < startRow + 3; row++) {
        for (int col = startCol; col < startCol + 3; col++) {
            if (row != i && col != j) {
                sameSquare.add(grid[row][col]);
            }
        }
    }
    return sameSquare;
}
```

## 3.4 Implementation of the Game Class

That's the class where we implement the AC-3 algorithm, one of the main goals of the assignment, and a function that checks if the result is valid.
`Game` class contains three functions implemented by me. AC-3 algorithm is basically implemented in `solve()` function and that function uses `revise()` function to revise the arcs. Idea behind those functions are explained in **Section 2.4.1**.

### 3.4.1 Implementation of the `solve()` Function

As preparation, an arc list named `arcs`, a priority queue named `queue` and an integer that will be used to compare complexity between heuristics names `count` are created. Then in a nested for loop, function adds arcs of each field by using `addArcs()` to `arcs`. Since now every field has an arc list, and they are added to `arcs`, we can add `arcs` to `queue`. With that, everything is ready to apply the AC-3 algorithm. Queue prioritize some arcs depending on the given heuristic, which is explained in sections **2.6** and **3.6**. After preparation, in a while loop that runs until the `queue` is empty, `count` is increased, and we get the first arc of our `queue` by using `poll()` function that pops the first element of a queue and removes it from the queue.
Then we save the domain size of popped arcs left hand side to an integer named `check`.

That will be used later to check if domain size has changed.
We revise the arc, design and implementation of `revise()` can be found in sections **2.4.1** and **3.4.2**, and check if domain size of the left hand side of the arc has changed after revising.
If it did not change, for every arc we added to `arcs` at the beginning, we check if the right hand side of these arcs are equal to revised arcs left hand side.
If we find an arc, then we add this to our queue if it's not already in there. The reason behind it is explained in **Section 2.4.1**.
Lastly, after `queue` is empty, function prints `count` to compare complexity of different heuristics, and returns true.
Here is the implementation:

```java
public boolean solve() {
    List<Arc> arcs = new ArrayList<>();
    Queue<Arc> queue = new PriorityQueue<>(new SimpleHeuristic());
    int count = 0;
    for (Field[] fields : sudoku.getBoard()) {
        for (Field f : fields) {
            f.addArcs();
            arcs.addAll(f.getArcs());
        }
    }
    queue.addAll(arcs);

    while (!queue.isEmpty()) {
        count++;
        Arc arc = queue.poll();
        int check = arc.getLeftHand().getDomainSize();
        revise(arc);
        if (arc.getLeftHand().getDomainSize() == 0) {
            return false;
        }
        if (check != arc.getLeftHand().getDomainSize()) {
            for (Arc a : arcs) {
                if (arc.getLeftHand() == a.getRightHand() && !queue.
                    contains(a)) {
                    queue.add(a);
                }
            }
        }
    }
    System.out.println("Complexity: " + count);
    return true;
}
```

### 3.4.2 Implementation of the `revise()` Function

`revise()` function takes an arc, and saves its left hand side domain and right hand side domain in `Integer` lists. Then if an item in left hand side is not equal to at least one of the items in the right hand side, we set `found` to true and break the loop, hence we don't remove that item. If we never break the loop, so an item from left hand side is equal to every item from the right hand side, then we add that to `valuesToRemove` to remove all values from the domain of left hand side. Actually for that to happen, right hand side domain size has to be 1. Logic of the function is explained in **Section 2.4.1**.
Here is the implementation:

```java
public void revise(Arc arc) {
    List<Integer> leftDomain = arc.getLeftHand().getDomain();
    List<Integer> rightDomain = arc.getRightHand().getDomain();
    List<Integer> valuesToRemove = new ArrayList<>();
    for (int l : leftDomain) {
        boolean found = false;
        for (int r : rightDomain) {
            if (l != r) {
                found = true;
                break;
            }
        }
        if (!found) {
            valuesToRemove.add(l);
        }
    }
    for (Integer val : valuesToRemove) {
        arc.getLeftHand().removeFromDomain(val);
    }
}
```

I couldn't find a way to just remove a value from left hand domain that I should remove the moment I find them. So instead, I found this solution where I add those values to an `Integer` list and then remove it. Somehow it worked but to be honest, I don't know why it did and why it didn't the other way.

### 3.4.3 Implementation of the `validSolution()` Function

Basically, this function checks the domain sizes of every field and if any of them is not 1, returns false. That makes sense since for a Sudoku to be solved, every field should be known, which means domain size of those fields should be 1. Here is the implementation:

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

## 3.5 Implementation of the Arc Class

The design of this class was very simple, since it contains 2 attributes, one for the left hand side of the arc and one for the right hand side of the arc, a constructor and getter and setter functions. Here is the implementation of the `Arc` class:

```java
public class Arc {
    private Field leftHand;
    private Field rightHand;

    public Arc(Field leftHand, Field rightHand) {
        this.leftHand = leftHand;
        this.rightHand = rightHand;
    }

    public Field getLeftHand() {
        return leftHand;
    }

    public void setLeftHand(Field leftHand) {
        this.leftHand = leftHand;
    }

    public Field getRightHand() {
        return rightHand;
    }

    public void setRightHand(Field rightHand) {
        this.rightHand = rightHand;
    }
}
```

## 3.6 Implementation of the Heuristics

For the implementation of the heuristics, I used `Comparator` structure. I created a `Heuristic` class that implements `Comparator`, and three heuristic functions that extends this class. Here is the implementation:

```java
public abstract class Heuristic implements Comparator<Arc> {
    @Override
    public abstract int compare(Arc o1, Arc o2);
}
```

### 3.6.1 Implementation of the Simple Heuristic

For `SimpleHeuristic`, I used default comparator. Here is the implementation:

```java
public class SimpleHeuristic extends Heuristic {
    @Override
    public int compare(Arc o1, Arc o2) {
        return 0;
    }
}
```

### 3.6.2 Implementation of the Minimum Remaining Values Heuristic

For `MinimumRemainingHeuristic`, I override the `compare()` function such that it prioritize arcs with less domain size on their right hand side. To do that, I simply checked if one of the arcs has a smaller domain size in their right hand side. Here is the implementation:

```java
public class MinimumRemainingHeuristic extends Heuristic {
    @Override
    public int compare(Arc o1, Arc o2) {
        if (o1.getRightHand().getDomainSize() < o2.getRightHand().
            getDomainSize()) {
            return -1;
        }
        else if (o1.getRightHand().getDomainSize() > o2.getRightHand().
            getDomainSize()) {
            return 1;
        }
        else {
            return 0;
        }
    }
}
```

### 3.6.3 Implementation of the Priority to Constraints With Finalized Arcs Heuristic

For `PriorityFinalizedHeuristic` I override the `compare()` function such that it prioritize arcs with domain size of 1 on their right hand side. To do that, I simply checked if there is a case of one arc having domain size of 1 on their right hand side and the other arc doesn't. Here is the implementation:

```java
public class PriorityFinalizedHeuristic extends Heuristic {
    @Override
    public int compare(Arc o1, Arc o2) {
        if (o1.getRightHand().getDomainSize() == 1 && o2.getRightHand().
            getDomainSize() != 1) {
            return 1;
        }
        else if (o1.getRightHand().getDomainSize() != 1 && o2.getRightHand
            ().getDomainSize() == 1) {
            return -1;
        }
        else {
            return 0;
        }
    }
}
```

## 4 Tests

In this section, testing of implementation is explained.
To test my implementation of AC-3 algorithm and heuristics, I tried to solve each Sudoku with each heuristic.
In `App` class, `start()` function takes a file path and creates a `Sudoku` object from this file path, and creates a `Game` object from that. Then if solving is successful, which means all constraints are satisfied, and if the solution is valid, which means domain size of each field is 1, it prints "Solved!". Else, it prints "Could not solve this sudoku :(".
That way we can understand if Sudoku could be solved or not. Function also shows the Sudoku after reporting whether the Sudoku could be solved or not. We can also look at that result and understand if it's solved.
Algorithms step count is also printed in the process, but that will be discussed in **Section 5**.
Here are some results:

- Sudoku 1: It is solved. Final table is fully filled with correct values.
- Sudoku 2: It is solved. Final table is fully filled with correct values.
- Sudoku 3: It is not fully solved. Final table is missing some values.
- Sudoku 4: It is not fully solved. Final table is missing some values.
- Sudoku 5: It is solved. Final table is fully filled with correct values.

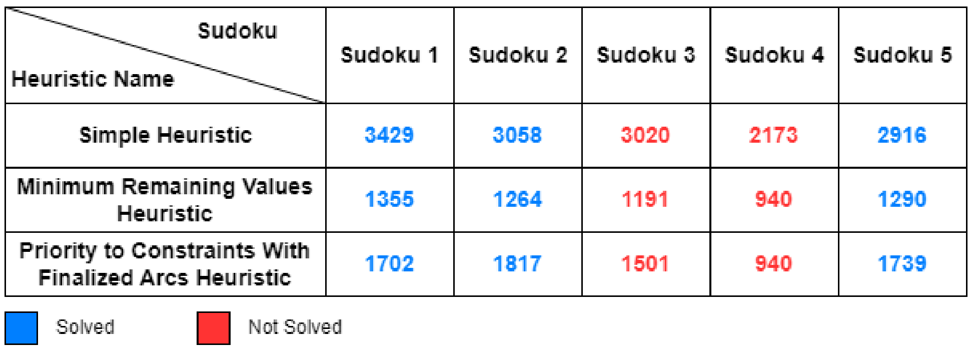
The reason that Sudoku 3 and Sudoku 4 not being solved is not actually because AC-3 algorithm is wrong. Since AC-3 algorithm eliminates variables only if the right hand side of an arc has a domain size of 1, for some Sudoku's, AC-3 algorithm can not find a full solution. I think designing and implementing a backtracking algorithm can solve these Sudoku's.

## 5 Complexity

In this section, the complexity of AC-3 and the heuristics are explained.
For measuring complexity of AC-3 algorithm and heuristics, I put a `count` integer that increases for every step of the algorithm in the implementation of the AC-3 algorithm. Counter increases every time an arc is popped from the queue. That counter will help us make comparison between AC-3 algorithm without heuristic(Simple Heuristic), and with different heuristics.
My guesses are prioritizing an arc will have a big affect on complexity, and I think Minimum Remaining Values Heuristic will be faster than Priority to Constraints With Finalized Arcs Heuristic.

### 5.1 Complexity Results of the AC-3 Algorithm With/Without Heuristic

Complexity results for AC-3 algorithm with and without heuristics are given below in a table.



| Sudoku / Heuristic Name | Sudoku 1 | Sudoku 2 | Sudoku 3 | Sudoku 4 | Sudoku 5 |
|---|---|---|---|---|---|
| Simple Heuristic | 3429 | 3058 | 3020 | 2173 | 2916 |
| Minimum Remaining Values Heuristic | 1355 | 1264 | 1191 | 940 | 1290 |
| Priority to Constraints With Finalized Arcs Heuristic | 1702 | 1817 | 1501 | 940 | 1739 |

🟦 Solved 🟥 Not Solved

### 5.2 Complexity Analysis

We can see that AC-3 algorithm with Minimum Remaining Values Heuristic or Priority to Constraints With Finalized Arcs Heuristic are both approximately 3 times faster than AC-3 algorithm without heuristic.
Also, as I thought, Minimum Remaining Values Heuristic is faster than Priority to Constraints With Finalized Arcs Heuristic.
That's because Minimum Remaining Values Heuristic prioritize arcs with smaller domain size, while Priority to Constraints With Finalized Arcs Heuristic only prioritize arcs with domain size of 1. To be more precise, Minimum Remaining Values Heuristic will still prioritize arcs with domain size of 1, if the other arcs right hand sides domain size is not 1, since 1 is the smallest domain size a field can have if there is no inconsistency. Basically, Minimum Remaining Values Heuristic will do what Priority to Constraints With Finalized Arcs Heuristic, but also prioritize arcs with smaller domain size when domain size is not 1.

## 6 Conclusion

In this project, we developed an implementation of AC-3 algorithm for solving Sudoku problems.
Overall, this project provided a deeper understanding of constraint satisfaction problems and the AC-3 algorithm but also highlighted the importance of heuristic choices in optimizing algorithms for solving constraint satisfaction problems such like Sudoku.