Solving Sudoku

Vamsi Yerramsetti s1032599, Tim Gerritsen s1041423

February 5, 2022

Abstract

This project report documents a successful implementation of the AC3 algorithm to solve a constraint satisfaction problem "Sudoku" and showcases the results from the output. All code, figures and interpretations were developed by the two Radboud University BSc students following the AI-Principles and Techniques course (SOW-BKI259).

1 Introduction

Sudoku is a very popular game and is played through out the world. It requires basic arithmetic knowledge but depending on the difficulty the puzzle, it can be solved with good problem-solving capability. However in the eyes of a programmer or an AI engineer, Sudoku is looked at as a constraint satisfaction problem which can resolved by implementing algorithms designed to solve such problems and satisfy the constraints. Such an algorithm is AC-3. This project report gives you an in-depth insight into the solving constraint satisfaction problems with such algorithms, a breakdown of the our algorithmic implementation in JAVA to solve Sudoku and a full analysis on the produced results,complexity and our perceived interpretation with reasoning [ASB09]. We were advised by our TA at the start of our project to make a hypothesis on what we expect the results would be and how they they would be under various heuristics. We hypothesized that using the AC3 algorithm, we can solve all the Sudoku puzzles using the MRV heuristic would lead to better results.

2 Constraint Satisfaction Problems

Problems that pose a set of constraints that impose conditions and limitations on a domain of variables which can only be solved by a process of satisfying the constraints imposed on the variables are known as constraint satisfaction problems. A variable is an entity which holds a value. CSPs are a unique type of "search problems" whose states are defined by values of a fixed set of variables [BO]. A CSP is:

- "Complete" when each variable has a valid value within the domain.
- "Consistent" when all constraints are satisfied.

CSPs specialise in deciding if there is a goal state and what it would look like. In the real world CSPs follow the process of resource allocation, scheduling and construction. However in our case Sudoku is a 9x9 grid with 81 variables all consisting of a value in the inclusive range of 1-9. The main constraint is that no two cells in the same row, column or sub-box can have the same value. The objective is to solve the puzzle such that all the constraints are satisfied.

There are many approaches/strategies to solve CSPs such as :

- Backtracking search: A form of depth first search where during each iteration, one variable is assigned and checked per level until there are no more valid variables to assigns which results in a backtrack to the next level.[BO]
- Solely using and following the MRV, Degree and LCV heuristics respectively to reach the desired goal state.

However, this project focuses on the working and implementation of the AC3 algorithm to solve the CSP: Sudoku, which we will now dive into.

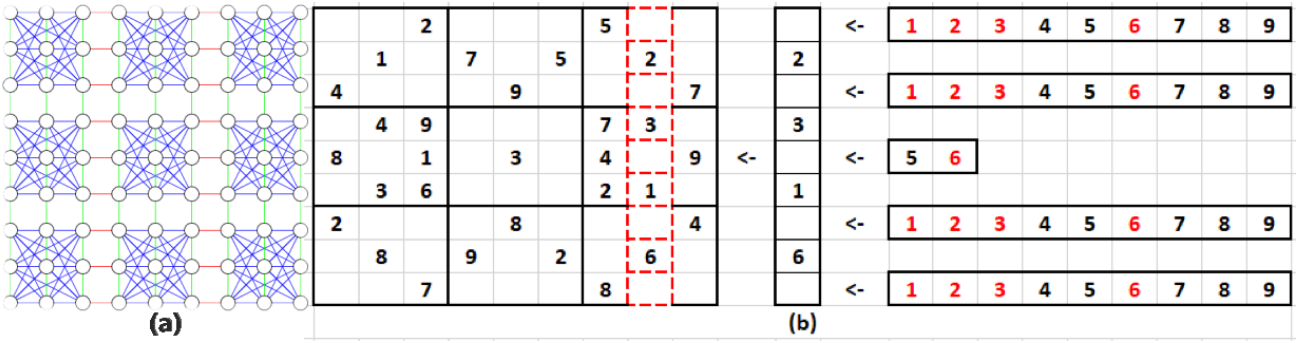


Figure 1: (a) Shows the various constraints in the puzzle. Blue portrays the constraints in each sub-box, red shows the row-constrains and green shows the column constraints. (b) Shows the variables in an example Sudoku-row and how their domains and values are affected by the various constraints

3 Arc Consistency 3 algorithm

A strategy to solve these constraint satisfaction problems is the usage of the AC-3 algorithm. The AC-3 algorithm is the most efficient algorithms from a series of CSP-solving algorithms developed by Alan Mackworth [ASB09].

3.1 The working of the algorithm

1. We turn each binary constraint into two "arcs". Ex: $A = B \rightarrow A = B$ and $A \neq B$
2. We add the arcs to priority queue.
3. We take an arc from the queue and check it with all the constraints keeping in mind that every left hand side of an arc should have at least one valid value from the domain range.
4. We remove values from the domain of that variable that when assigned to the variable do not satisfy all the constraints.
5. If the domain of the variable has changed then we add all arcs that associate with that variable to the end of the queue for a consistency recheck.
6. We repeat the above three steps until the queue is empty and by then we would have solved our CSP.

In our case we used AC-3 to solve the CSP:Sudoku. For any two variables that have mutual constraints (any two squares in the same row, or the same column, or in the same large square in Sudoku), the AC3 is used to reduce the domain of each grid and hence reduce subsequent space and time consumption. In the beginning, the known squares in the title are filled in. At this time, using AC3 algorithm to judge the arc consistency can greatly reduce the domain of some squares. For some simpler (more known values) Sudoku, you can get the solution directly without continuing the subsequent search. AC3 is also used in backtracking, maintaining the arc consistency every time a value was assigned.

AC3 applied on a sample CSP with strating domains{1,2,3} for all variables			
Variables & Domains	**Queue**	**Arcs**	**Constraints**
A={ ,2,3}	A>B	A>B	A>B
B={1,2, }	B<A	B<A	B=C
C={1,2,3}	B=C	B=C	
	C=B		
	A>B		

Figure 2: This table represents a midway state of AC-3 solving a simple CSP with three variables all initially having a domain of 1,2,3 and two constraints. The cells in red are the arcs in the queue that are satisfied. The algorithm is finished once the whole queue is red.

4 Implementation

4.1 Setup

We start by setting up some variables. In order to setup the constraints, we must first establish all the neighbours. As shown in figure 1a, each cell has the following neighbours:

1. all other cells in the same row
2. all other cells in the same column
3. all other cells in the same box

Our implementation of finding such neighbours for each cell is as follows:

```
private static void addNeighbours(Field[][] grid) {
    for(int row = 0; row < 9; row++){
      for(int col = 0; col < 9; col++){
        ArrayList<Field> neighbours = new ArrayList<>();

        //Add all neighbours from the same row
        for(int i = 0; i < 9; i++){
          //Don't add itself to it's neighbours
          if(i==row) continue;
          neighbours.add(grid[i][col]);
        }

        //Add all neighbours from the same column
        for(int i = 0; i < 9; i++){
          //Don't add itself to it's neighbours
          if(i==col) continue;
          neighbours.add(grid[row][i]);
        }

        int box = getBox(row,col);
        //Add all neighbours from the same box
        for(int i = 0; i < 3; i++){
          for(int j = 0; j< 3; j++){
            int rowOnGrid = box - (box % 3);
            int colOnGrid = (box % 3) * 3;
            //Don't add itself to it's neighbours
            if(i+rowOnGrid==row && j+colOnGrid == col || neighbours.contains(grid[i+
    rowOnGrid][j+rowOnGrid])) continue;
            neighbours.add(grid[i+rowOnGrid][j+colOnGrid]);
          }
        }

        grid[row][col].setNeighbours(neighbours);
      }
    }
  }
```

Next, we need to setup arcs for all the constraints, if we put this into perspective, it would be exactly like figure 1a. Our implementation is as follows:

```
void setupArcs(){
    //For each field, go through all it's neighbours and add an arc between the field
    and the neighbour
    for(int i = 0; i < 9; i++){
        for(int j = 0; j < 9; j++){
            Field field = fields[i][j];
            for (Field neighbour : field.getNeighbours()) {
                if(neighbour.getDomainSize()==0 ) continue;
                arcs.add(new Arc(field, neighbour));
            }
        }
    }
}
```

Where Arc is a simple class:

```
class Arc{
    Field x;
    Field y;

    Arc(Field f1, Field f2){x = f1; y=  f2;}
}
```

And arcs is a PriorityQueue of Arc's. The setupArcs function runs through each cell on the field and checks whether the domain has a size larger than 0 (if not, the cell has a fixed value, and therefore should not be added) and adds it to the 'worklist' (the PriorityQueue).

4.2 AC3 Algorithm

Now for the main algorithm, the pseudo code of AC3 is as follows:

```
Input:
   A set of variables X
   A set of domains D(x) for each variable x in X. D(x) contains vx0, vx1... vxn, the
    possible values of x
   A set of unary constraints R1(x) on variable x that must be satisfied
   A set of binary constraints R2(x, y) on variables x and y that must be satisfied

 Output:
   Arc consistent domains for each variable.

 function ac3 (X, D, R1, R2)
     // Initial domains are made consistent with unary constraints.
     for each x in X
         D(x) := { vx in D(x) | vx satisfies R1(x) }
     // 'worklist' contains all arcs we wish to prove consistent or not.
     worklist := { (x, y) | there exists a relation R2(x, y) or a relation R2(y, x) }

     do
         select any arc (x, y) from arcs
         arcs := arcs - (x, y)
         if arc-reduce (x, y)
             if D(x) is empty
                 return failure
             else
                 arcs := arcs + { (z, x) | z != y and there exists a relation R2(x, z)
     or a relation R2(z, x) }
     while arcs not empty

 function arc-reduce (x, y)
     bool change = false
     for each vx in D(x)
         find a value vy in D(y) such that vx and vy satisfy the constraint R2(x, y)
         if there is no such vy {
             D(x) := D(x) - vx
             change := true
         }
     return change
```

And our implementation of this algorithm is as follows:

```
public boolean solve(){
    while(!arcs.isEmpty()){
        Arc arc = arcs.remove();
        if(arcReduce(arc)){
            if(arc.x.getDomainSize()==0) return false;
            for(Field neighbour : arc.x.getOtherNeighbours(arc.y)){
                arcs.add(new Arc(neighbour, arc.x));
            }
        }
    }
    return true;
}
private boolean arcReduce(Arc arc){
    boolean change = false;
    ArrayList<Integer> domain = new ArrayList<>(arc.x.getDomain());
    for(Integer vx : domain){
        boolean hasDifferentValue = false;
        for(Integer vy : arc.y.getDomain()){
            if(vx != vy){
                hasDifferentValue = true;
                break;
            }
        }

        if(!hasDifferentValue){
            arc.x.removeFromDomain(vx);
            change = true;
        }
    }
    return change;
}
```

5 Verification of Output and Testing

5.1 Variables, Domains and Constraints

In our Sudoku CSP we have a 9x9 grid so 81 cells. These are the variables. Each of these variables initially have a domain size of nine from values ranging from 1-9. There are three constraints. All variables in the same row,same column and same 3x3 sub-boxes must have unique values from their domain. This part of our report is explained in the beginning and in other parts but we are formally stating them here as well.

5.2 Heuristics

Heuristics are used to estimate the most efficient ordering of the constraints. There were two main heuristics that we implemented and tested: MRV and LCV-Degree.

1. Minimum remaining values (MRV) heuristic works by choosing the variable with the fewest possible values in its domain at said time. The idea is to assign most constrained variable first and then prune the impossible assignments fairly early.

2. Least-constraining-value (LCV) heuristic works by selecting one value for a current variable that can provide the maximum choices for all neighboring variables. LCV chooses a value that rules out the smallest number of values in variables connected to the current variable by constraints.

For the MRV heuristic we have to setup a function which finds an unassigned node (meaning, having domain size > 0) with the minimum remaining value in the field.
For the LCV we have to setup a function which determines the least constraining value in a node. This means, that we have to go through each value of the domain of that given node, then go through it's neighbours and keep track of the times that the other domains (the neighbours domains) have that value too. The more times a value is found in the neighbour's domains, the more it is constrained. By keeping a HashMap, we can determine what the LCV is. We add the size of every

neighbour's domain and subtract the values that match the value. At the end of the function, return the key that has the biggest value in the HashMap, as it is least constrained.

5.3 Verification function

In order to verify whether a solution output is correct, we have to run through each row,column and box and check that there exists at least and at most one of each of the following numbers: [1,2,3,4,5,6,7,8,9]. That would make a valid Sudoku solution. Our implementation for this validation is as follows:

```
public boolean validSolution() {
    //Check all rows
    for(int row = 0; row < 9; row++){
      ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9));
      for(int col = 0; col < 9; col++){
        if(!numbers.remove(Integer.valueOf(sudoku.getBoard()[row][col].getValue()))){
          return false;
        }
      }

      if(numbers.size() > 0) return false;
    }

    //Check all columns
    for(int col = 0; col < 9; col++){
      ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9));
      for(int row = 0; row < 9; row++){
        if(!numbers.remove(Integer.valueOf(sudoku.getBoard()[row][col].getValue()))){
          return false;
        }
      }

      if(numbers.size() > 0) return false;
    }

    //Check all 9 boxes
    for(int box = 0; box < 9; box++){
      ArrayList<Integer> numbers = new ArrayList<>(Arrays.asList(1,2,3,4,5,6,7,8,9));
      for(int row = 0; row < 3; row++){
        for(int col = 0; col < 3; col++){
          int rowOnGrid = box - (box % 3);
          int colOnGrid = (box % 3) * 3;
          if(!numbers.remove(Integer.valueOf(sudoku.getBoard()[rowOnGrid+row][
    colOnGrid+col].getValue()))){
            return false;
          }
      }
      }

      if(numbers.size() > 0) return false;
    }

    //Everything passed
    return true;
}
```

5.4 Testing

For testing, we setup a variable to keep count of the number of iterations that the algorithm performs. This way, we can keep a realistic track of our algorithm complexity and what the heuristics do to improve it (or how they don't). As a second measure, we keep track of how many arcs are created in the entire algorithm process.

6 Complexity Analysis

There are three main parts to our code which is the AC3 algorithm, the setup of the constraints, the neighbor-function and the verification function. These parts are where the main computations take

place and hence make up the overall complexity. We believe that using such approach to solve general Sudoku problems of grid size(N * N) is an NP-Complete problem as AC-3 could be used to solve other NP-complete problems as well like the "travelling-salesman problem" [Che09]. However, we are dealing with a Sudoku grid of 9x9 so even a brute-force approach can solve this problem in polynomial time. The underlying contributors to the complexity of the AC-3 algorithm is number of arcs and the domain size. We will explain this in depth in the next section along with a complexity analysis and comparison of the heuristics.

6.1 Complexity of setting up the constraints : setupArcs()

This function has three for-loops the first two run 9 times (Sudoku board size) and the third runs getNeighbours() number of times, which at max can be the number of available neighbors in the same box,column and row which is twenty. So in total this function has a complexity of $O(y)$ where $y = 9 * 9 * 20$. So it is done in polynomial-time : O(n).

6.2 Complexity of addNeighbours()

This function has two major nested for-loops which loops over the entire grid(9x9). The second for-loop has a set of for loops for the row,column and sub-box each run nine times. So in total this function has a complexity of $O(x)$ where $x = 9 * 9 * 9$. So it executes in polynomial-time: O(n).

6.3 Complexity of the verification function :

The verification function validSolution() can be interpreted in three parts: checking the rows, checking the columns and checking the sub-boxes. To check the rows we have two nested for-loops which run nine times each hence we have it running in $O(9 * 9)$.Hence in polynomial time: $O(n)$. Same applies for the column-check. To check the sub-boxes the function has three nested for-loops the first runs for each cell in the sub-box: 9, the second and third run for every row and column in the subgroup so that's three for each respectively. In total we have a complexity of $O(9 * 3 * 3)$.
So in conclusion the function: validSolution() runs in polynomial time :
$O(81) + O(81) + O(81) \rightarrow O(n)$.

6.4 Complexity of the methods for the AC-3 algorithm

The **solve()** and **arcReduce()** methods make up the AC-3 algorithm.
solve() has a while-loop which runs until all the arcs are empty, (so it has a complexity of $O(a)$ where "a" is number of arcs) and a nested for-loop which runs until there are no more neighbours to get in an arc which can be done in polynomial time: $O(domain.size)$. The function also calls the arcs.add() method which is done in polynomial time. The domain-size of each cell of a 9x9 Sudoku puzzle will always be 9 or less, as the algorithm prunes values that don't satisfy the said constraints. So the domain size cannot exceed 9 and each cell can be one of nine values ranging from 1-9. So in total the complexity of the solve() function is $O(a * 9) + O(a)$. This function computes in polynomial time.

arcReduce() has two for-loops which each run through the domain size which will give a complexity of $O(domain.size)^2$. This results to $O(k)$ where $K = 9 * 9$. A final if-statement calls the removeFromDomain() function which has a complexity of $O(domain.size)$. This function computes in polynomial time.

6.5 Overall complexity

So in total solve() has a complexity of $O(a * domain.size) + O(a)$ where "a" is the number of arcs and arcReduce() has a complexity of $O(d^2)$ so in total the AC-3 algorithm has a complexity of:

$$O(a * d^3) \rightarrow O(a * 729) \rightarrow O(n).$$

So the AC-3 algorithm with the additional helper functions can solve a 9x9 Sudoku puzzle in polynomial time: $O(n)$**.** [Che09]

7 Heuristic complexity analysis

We wanted to analyse how the algorithm works in terms of "complexity". So to interpret this we needed a way of measuring the workload done by the algorithm under various heuristics. We had implemented a counter in-a-loop in the AC-3 algorithm which counted the number of iterations. We ran and recorded the values under the various heuristics we used and made the following bar graph to compare the algorithm's performance in terms of complexity following various heuristics.

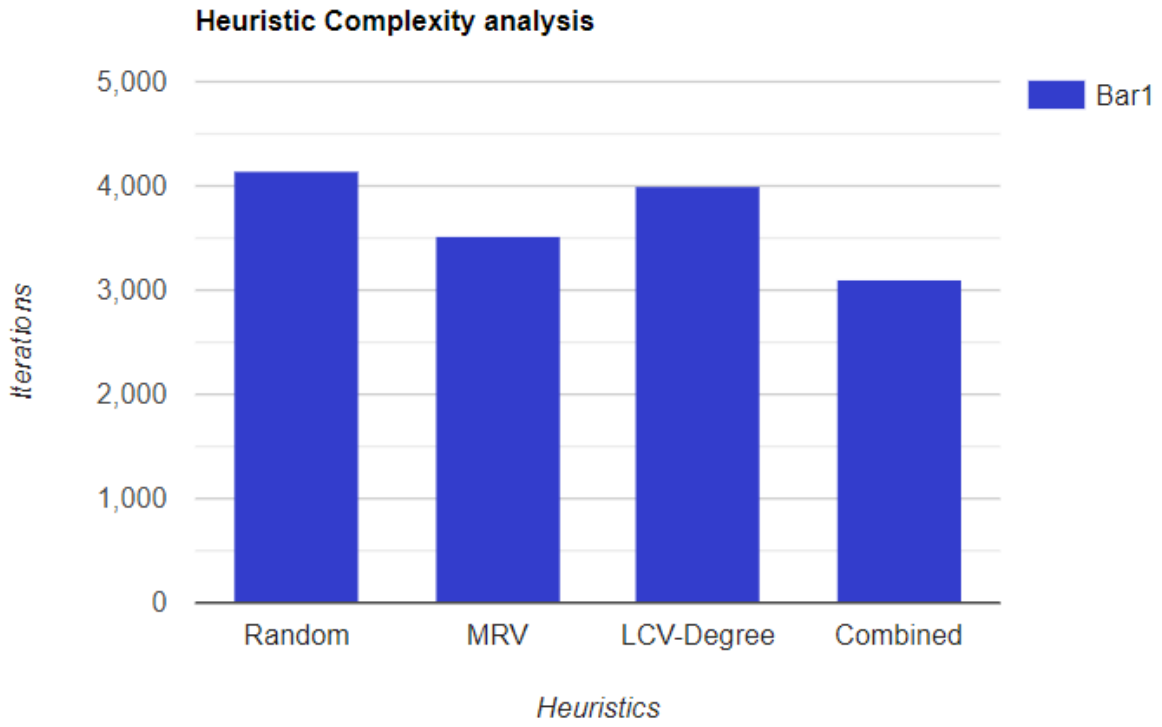


Figure 3: This graph give us a comparison of how AC-3 performs in terms of complexity under various heuristics

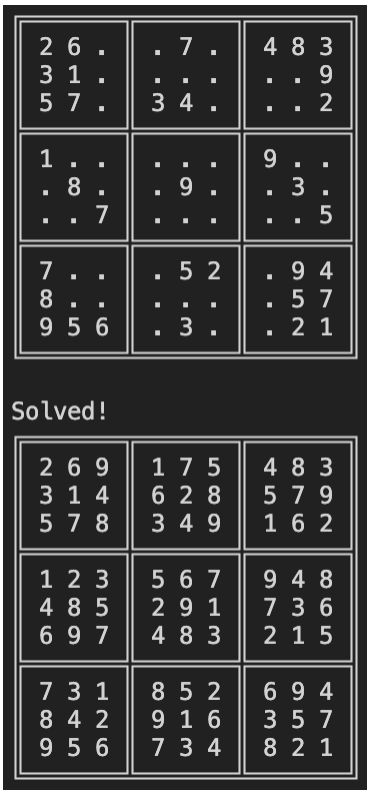
From this graph we made quite a few observations. The algorithm has the least workload when we ran it with a combination of LCV-Degree and MRV heuristics hence resulting resulting in the best run-time complexity. MRV seems to be a better heuristic to use than LCV-Degree. Running the code with no set heuristics gives us the worst run-time-complexity.

8 Results

Our algorithm was able to solve Sudoku 1, 2 and 5. It was unable to solve Sudoku 3 and 4. The table below shows the number of arcs checked for all 3 solvable Sudoku's along with the heuristic used.

	No heuristics (random)	MRV	LCV
Sudoku 1	9808	7805	6505
Sudoku 2	10088	8594	7392
Sudoku 5	9232	8504	6503

The output of our algorithm looks as follows (for Sudoku 5):



8.1 Interpretation of the results

We found the algorithm to find the right solution to the Sudoku's it was able to solve, and within very reasonable time too. AC-3 however, is not able to solve any given Sudoku. The AC-3 algorithm only checks for individual arcs and can guarentee that an arc has each individual constraint (arc consistency) has a solution but this does not guarantee that the CSP has a solution, therefore it might be the case that the algorithm can't solve a sudoku. Once one domain is completely empty, the algorithm stops, indicating there is no solution to the CSP.

So our first part of our hypothesis was wrong AC-3 does not solve all CSPs however it does make them much simpler and easier to solve as it reduced domain size and satisfies constraints with solutions. On the bright side our second part of our hypothesis turned out to be true. Using the MRV heuristic gave us better results in terms of complexity.

9 Bonus

In order to improve upon the algorithm, we would try and come up with an additional constraint to achieve more than just basic arc consistency. One way we could think of in order to do so, would be to try and come up with a constraint, that knows to prune any values that can not be assigned to anymore because they are needed for other constraints. So we tried to implement this strategy. Another strategy is implementing the strategy above for every row as well.We then tried to introduce meta-constraints such we enforce the above constraints but this time using a recursive approach. This was difficult for us to implement.

10 Conclusion

Despite not being Sudoku players we felt it was easy to interpret the puzzle from a constraint satisfaction problem perspective. Since we were working with a 9x9 Sudoku puzzle we found it fairly easy

to successfully understand, implement and test AC-3 on various versions of the Sudoku puzzle. Our testing and results section support our interpretations and overall implementation. However when exploring other Sudoku board with greater sizes we realise that the problem becomes much harder and it shows in terms of complexity. For future development of our project we would want to expand our algorithm so that it can work on various sizes. We believe we would need to use meta-heuristics and a hybrid version of AC-3 to solve such puzzles with a reasonable time and space complexity [SCG+14].

References

[ASB09] Marlene Arangu, Miguel A Salido, and Federico Barber. Ac3-op: An arc-consistency algorithm for arithmetic constraints. In *CCIA*, pages 293–300, 2009.

[BO] Benjamin Bittner and Kris Oosting. Implementing a csp solver for sudoku.

[Che09] Zhe Chen. Heuristic reasoning on graph and game complexity of sudoku. *arXiv preprint arXiv:0903.1659*, 2009.

[SCG+14] Ricardo Soto, Broderick Crawford, Cristian Galleguillos, Sanjay Misra, and Eduardo Olguín. Solving sudokus via metaheuristics and ac3. In *2014 IEEE 6th International Conference on Adaptive Science & Technology (ICAST)*, pages 1–3. IEEE, 2014.