CS4402

REPORT(P2)

220026989

I. Introduction

This program implements Forward Checking and Maintaining Arc Consistency with a given basic pseudo code. The solver methods created is then tested with some given instances.

The submitted file has 3 java files, created for each functionalities. The main file is the Main.java, which only takes in the arguments and calls the respective java files:- FC.java for forward checking or MAC.java that implements AC3 algorithm. A jar file is added to the submitted file and can be executed as:

- ***java -jar P2.jar <filename.csp> <fc|mac> <asc|sdf>***

  An extra asc extension can be given on the command file. But since it is the only method given to implement selection of values, it is made to be default.

II. Model

1. Forward Checking

The data structures used for implementing the FC are described here:

- An arrayList of integers called unaAllocated variables are declared initially that has all the initial variables in it. The code snippet of function setVars() is used to allocate the unallocated arraylist is shown in the method description:

Apart from the unallocated variable arrayList another arrayList of allocated variable is also used to keep track of the same.

- A HashMap, named varDomains is used to store the domains of the variables. It uses the variable number as the key for retrieving the domain

of that variable. Moreover, All the values for a domain are stored within an ArrayList of Integers. This hashmap is filled with the setVars() function.

- Method description.

1.1) **setVars() :** This method is only used to populate the unassigned ArrayList with all the variables of the instance, and the varDomains HashMap with all of the domains for each variable, before initialising the Forward Checking algorithm.

It should be noted that the functionality of variables being in a specific order can be incorporated into this function although it is not. The same functionality is implemented in the getVars() function that takes one variable at a time.This initial list will be passed into the Forward checking function.

```
    /**
     * Method to populate the unallocated list with all the
variables and varDomains.
     * Hash Map with the corresponding domains of each variable.
     */
    private void setVars() {
        for(int i = 0; i < binaryCSP.getNoVariables(); i++) {
            int variable = i;
            ArrayList<Integer> domain = new ArrayList<>();
            for(int j = binaryCSP.getLB(i); j <
binaryCSP.getUB(i)+1; j++) { //creating the domain list.
                domain.add(j);
            }
            //after the domain is filled, variable is added to the
unallocated var list and is placed in the hashmap of variable and
domain.
            unallocatedVars.add(i);
            varDomains.put(variable, domain);
        }
    }
```

1.2)

forwardChecking() – This method, which is the main one for the forward checking operation, accepts as an argument an ArrayList of Integers that is initially an unallocated ArrayList. First, it determines whether the algorithm has produced a solution, in which case it prints the answer and then moves on. If the problem hasn't been solved, it will use the methods selectVar and selectVal to choose a variable from the varList and a value from the variable's domain. The branchFCLeft and branchFCRight methods are then accordingly invoked.

```
/**
     * Method that implements forward checking.
     * @param varList The list of variables required for FC.
     */
    private void forwardChecking(ArrayList<Integer> varList) {

        numberOfNodes++;

        if(allocatedVars.size() == binaryCSP.getNoVariables() && !solutionFound)
{  //Print the first solution found.

            System.out.println(numberOfNodes); //printing the number of nodes is
printed first.
            System.out.println(arcRevisions); //the number of arcs revised is
printed second.
            long finish = System.currentTimeMillis();
            long timeElapsed = finish - start;

            System.out.println(timeElapsed); //time taken to solve third
            System.out.println();
            printSolution(); //finally printing the values of each variable
serparately.
            solutionFound = true;
        }
        else if(!solutionFound) {   //If a solution was not found, we proceed.
            int var = selectVar(varList, varOrder); //Selecting a variable
according to the varOrder requested.
            int val = selectVal(var);   //Select a value from the variable's
domain(always in ascending order).

            branchFCLeft(varList, var, val);    //Branching left.
```

```
            branchFCRight(varList, var, val);   //After branching left returns,
branch right.
        }
```

1.3)

branchFCLeft() – This method also takes in the varList as a parameter, as well as the variable and value selected in the forwardCheck method. Firstly, it makes a copy of all of the current domains, which will be used for undoing the pruning(Important). The method will then be used to assign the variable and value. The reviseFutureArcs method will then be used to check the consistency of the future arcs. If it is consistent, it will use getVarList to produce the varList again and then call forwardChecking while supplying the newly generated varList to do a recursive call. Using the copy created at the start of the function, it will restore the variables domain in the event that subsequent arcs are inconsistent. The variable and value are then unassigned using the unassign method.

```
    /**
     * Branching left.
     * @param varList variable list
     * @param var current var
     * @param val current val assigned to the variable
     */
    private void branchFCLeft(ArrayList<Integer> varList, int var, int val) {

        HashMap<Integer, ArrayList<Integer>> prevDomains = new HashMap<Integer,
ArrayList<Integer>>(varDomains); //clone of varDomains
        assign(var, val);   //Assign the variable.


        if(reviseFutureArcs(varList, var)) {    //If consistent.
            varList = getVarList();     //Re-create varList.
            forwardChecking(varList);
        }

        varDomains = prevDomains;   //Undoing pruning.
        unassign(var, val);     //Unassigning variable.
    }
```

1.4)

The same parameters as branchFCLeft are used by branchFCRight. Using the deleteValue method, it first removes the supplied value from the passed variable's domain. It then determines whether the variable's domain is empty following the preceding operation. If it is not empty, a copy of the domains will be made so that the pruning can be undone, and the future arcs will be revised. If subsequent arcs are consistent, it calls forwardChecking recursively while passing the same varList that was supplied to this method. If not, the pruning is reversed using the copy of the domains, and the restoreValue method is used to restore the value inside the variable's domain.

```
    /**
     * Branching right.
     * @param varList variable list in use
     * @param var the current variable
     * @param val the current value of the current variable
     */
    private void branchFCRight(ArrayList<Integer> varList, int var, int val) {

        deleteValue(var, val);

        if(!varDomains.get(var).isEmpty()) {
            HashMap<Integer, ArrayList<Integer>> prevDomains = new
HashMap<Integer, ArrayList<Integer>>(varDomains);    //Saving domains to undo the
pruning.
            if(reviseFutureArcs(varList, var)) {
                forwardChecking(varList);
            }
            varDomains = prevDomains;   //Undoing pruning.
        }

        restoreValue(var, val);
    }
```

1.5)

reviseFutureArcs – The varList and variable are required arguments for this procedure. First, it iterates through the varList, checking each element to see if the variable that was handed in

was there. If not, it then calls the function revise for the arc. It will return false if the arc is inconsistent; else, it will return true.

```
    /**
     * Revises if future arcs are consistent.
     * @param varList the set of variables
     * @param var the current variable
     * @return true if consistent, otherwise return false.
     */
    private boolean reviseFutureArcs(ArrayList<Integer> varList, int var) {

        boolean consistent = true;

        for(int futureVar : varList) {
            if(futureVar != var) {
                try {
                    consistent = revise(var, futureVar);
                } catch (Exception e) {
                    e.printStackTrace();
                }
                if(!consistent)
                    return false;
            }
        }

        return true;
    }
```

1.6)

revise – The present variable and the future variable are the two variables used in this strategy. The future variable's domain is then pruned after collecting all the accepted values for that arc using the getUnConstraintValues function. Any value that is not a supported value of that arc is removed from the future variable domain as part of the pruning process. It returns false since it is inconsistent if the domain being pruned is empty at the end; otherwise, it returns true.

```
    /**
     * Prunes domain of arc(var, futureVar).
     * @param var Current variable
```

```
     * @param futureVar The variable connected to the current variable that is to
be pruned.
     * @return false if there where no changes, otherwise return true.
     * @throws Exception when the pruned domain is empty.
     */
    private boolean revise(int var, int futureVar) throws Exception {
        arcRevisions++;

        ArrayList<Integer> acceptableValues = getUnConstraintValues(var,
futureVar);    //Values from supported tuples for arc(var, futureVar).
        if(!acceptableValues.isEmpty()) {
        ArrayList<Integer> newDomain = new
ArrayList<Integer>(varDomains.get(futureVar));   //Get domain.
        for(int i = 0; i < varDomains.get(futureVar).size(); i++) {
            if(!acceptableValues.contains(varDomains.get(futureVar).get(i)))
{  //If domain contains an unsupported value.
                newDomain.remove(newDomain.indexOf(varDomains.get(futureVar).get(
i)));  //Prune unsupported value.
            }
        }

        if(newDomain.isEmpty()) {  //If the new pruned domain is empty.
            return false;
        }
        else {
        varDomains.put(futureVar, newDomain);   //Replace the domain with the new
pruned domain.
        return true;
        }
    } else {
        return false;
    }
}
```

1.7)

getUnconstraintValues - The method takes the two arc variables and iterates through the tuples and domains to produce a list of the supported values. This list is then returned and used for pruning by the revise method.:

```
    /**
     * Method to get the values that where supported by the tuples of c(var,
futureVar).
```

```
     * @param var current variable of the node
     * @param futureVar The future variables linked.
     * @return an ArrayList with the supported values.
     */
    private ArrayList<Integer> getUnConstraintValues(int var, int futureVar) {

        ArrayList<Integer> acceptableValues = new ArrayList<>();

        for(BinaryConstraint c : constraints) {
            if(c.getFirstVar() == var && c.getSecondVar() == futureVar) {
                for(BinaryTuple bt : c.getTuples()) {
                    for(int val : varDomains.get(var)) {
                        if(bt.getVal1() == val) {
                            acceptableValues.add(bt.getVal2());
                        }
                    }
                }
            }
        }
        return acceptableValues;
    }
```

1.8)

getVarList – The varList is recreated using this method, which only returns a list of variables that have not yet been assigned.

```
    /**
     * Method to get a varList with the variables that have not been assigned
yet.
     * @return populated ArrayList varList
     */
    private ArrayList<Integer> getVarList() {
        ArrayList<Integer> varList = new ArrayList<>();
        for(int i = 0; i < binaryCSP.getNoVariables(); i++) {
            if(!allocatedVars.contains(i)) { //If variable i has not been
assigned yet.
                varList.add(i); //Add the variable to the list.
            }
        }
```

```
        return varList;
    }
```

1.9)

assign – This method is used to allocate a variable. In order to achieve this, it enters the variable into the ArrayList that was assigned to it and deletes all values from its domain other than the one that was assigned to it.

```
    /**
     * Assigns a variable by adding it to the assigned list, and prunes all
     * of the values from the domain apart from the assigned value (Domain will
only contain val).
     * @param var Variable to which value has to be assigned.
     * @param val value that has to be assigned.
     */
    private void assign(int var, int val) {
        allocatedVars.add(var);
        ArrayList<Integer> domain = new ArrayList<>();
        domain.add(val);
        varDomains.put(var, domain);
    }
```

1.10)

unassign – This is the method used for unassigning a variable, by removing it from the assigned ArrayList. It does not undo the pruning as this is already done when undoing the pruning in the branchFCLeft and branchFCLeft methods:

```
    /**
     * Unassigns a variable by removing the variable of the assigned list.
     * @param var variable whose value is to be unassigned.
     * @param val Value to be unassigned
     */
    private void unassign(int var, int val) {
        allocatedVars.remove(allocatedVars.indexOf(var));
    }
```

1.11)

restoreValue – This method takes a variable and value, and re-introduces the value into the variable's domain.

```
    /**
     * Re-introduces the value in the domain of the specified variable.
     * @param var Variable whose values has to be restored.
     * @param val Value to be re-assigned.
     */
    private void restoreValue(int var, int val) {
        ArrayList<Integer> domain = varDomains.get(var);
        domain.add(val);
        varDomains.put(var, domain);
    }
```

1.12)

deleteValue – The restoreValue method performs the opposite with this one. It deletes the value rather than adding it back to the domain.

```
    /**
     * Deletes a value from the domain of the specified variable.
     * @param var Variable whose domain value has to e removed.
     * @param val Value that has to be removed.
     */
    private void deleteValue(int var, int val) {
        ArrayList<Integer> domain = varDomains.get(var);
        domain.remove(domain.indexOf(val));
        varDomains.put(var, domain);
    }
```

1.13)

selectVal – To choose a value from a variable's domain, use this method. In order to do this, it first sorts the variable's domain into ascending order or smallest-domain-first order before returning the top-most element of the new domain.

```
    /**
     * Selects the first value from the domain of the specified variable, after
the domain
     * is ordered in ascending order.
     * @param var variable whose value is to be
     * @return val selected.
     */
    private int selectVal(int var) {
        ArrayList<Integer> domain = varDomains.get(var);
        domain.sort(null); //Sorts domain in ascending order.
        return domain.get(0);
    }
```

1.14)

selectVar – This method is used to choose a variable. It will always return the list's first variable if it is at the start. The variable with the smallest domain or the smallest variable (ascending) within the varList will be returned if it is not at the start.

```
    /**
     * Selects a variable from the varList. At the beginning, return the first
variable.
     * Else, return the variable with the smallest domain.
     * @param varList The current list of variables
     * @return variable selected variable according to the variable order
required.
     */
    private int selectVar(ArrayList<Integer> varList, String varOrder) {
        int variable = varList.get(0);
        if(varOrder.equals("sdf")) {
        int smallestDomain = varDomains.get(variable).size();
        if(varList.size() == binaryCSP.getNoVariables()) //If at the beginning
            return variable;
        else {
            for(int i = 0; i < varList.size(); i++) {
                if(varDomains.get(varList.get(i)).size() < smallestDomain) {
                    variable = varList.get(i); //Keeps track of the variable with
the smallest domain.
                    smallestDomain = varDomains.get(varList.get(i)).size();
//Keeps track of the size of the smallest domain encountered.
                }
            }
        }
```

```
        return variable;
    } else {
        return variable;
    }
    }
```

1.15)

printSolution – This method is only used to output the solution.

```
    /**
     * Prints the solution.
     */
    private void printSolution() {
        for(int i = 0; i < varDomains.size(); i++) {
            System.out.println(varDomains.get(i).get(0));
        }
    }
}
```

2. Maintaining Arc Consistency

With just one exception, MAC makes use of the same data structures as forward checking. An ArrayList of Integer arrays called queue is utilised by MAC, more especially the AC3 method, to enter the arcs that will be changed by that method. We'll handle this ArrayList like a queue. Because of this, it will always access, delete, and add to the list's last entry.

2.1)

MAC3 – This is the main MAC method. Using selectVar and selectVal, it first chooses a variable and a value. The pruning is then undone by creating a replica of the domains. Using assign, it assigns the verifiable and determines whether a solution exists. Print the answer and stop if there is a solution. Otherwise, keep going. It then calls AC3, and if it is consistent, recreates the variable list before calling MAC3 repeatedly while handing the new variable list to it. If not, the pruning is undone using the earlier domain copies, and unassign unassigns the variable. The

"Right" branching will then be pursued, which follows the same format as the method branchFCRight but calls AC3 rather than reviseFutureArcs.

```
    /**
     * Method that implements MAC.
     * @param varList The list of variables required for MAC.
     */
    private void MAC3(ArrayList<Integer> varList) {

        numberOfNodes++;

        int var = selectVar(varList, varOrder); //Select a variable.
        int val = selectVal(var);   //Select a value from the variable's
domain.

        HashMap<Integer, ArrayList<Integer>> prevDomains = new HashMap<Integer,
ArrayList<Integer>>(varDomains); //Saving domains to undo the pruning.

        assign(var, val);

        if(allocatedVars.size() == binaryCSP.getNoVariables() && !solutionFound)
{ //Print the first solution found.

            // System.out.println("\n Solution found after " + numberOfNodes + "
nodes");
            System.out.println(numberOfNodes);
            System.out.println(arcRevisions);
            long finish = System.currentTimeMillis();
            long timeElapsed = finish - start;
            System.out.println(timeElapsed);
            System.out.println();
            printSolution();
            solutionFound = true;
        }
        else if(AC3(var) && !solutionFound) { //If a solution was not found,
proceed.
            varList = getVarList();
            MAC3(varList);
        }

        varDomains = prevDomains; //Undoing pruning.
        unassign(var, val);

        deleteValue(var, val);
        prevDomains = new HashMap<Integer, ArrayList<Integer>>(varDomains);
```

```
        if(!varDomains.get(var).isEmpty()) {
            if(AC3(var))
                MAC3(varList);
            else
                varDomains = prevDomains; //Undoing pruning.
        }

        restoreValue(var, val);
    }
```

2.2)

AC3 – First, it creates an empty queue, inserts the appropriate arcs, and initialises the queue. When the queue is empty or an exception is caught, the while loop does not end. The first arc of the queue is chosen, eliminated, and then handed to revise within the while loop. New arcs are added to the queue if there were any revisions to edit. Revise throws an exception that is captured in the loop and returns false to finalise if it empty's a domain. If not, the operation continues, and if a problem was not encountered before the end of the process, true is returned.

```
    /**
     * AC3 MAIN PROCEDURE
     * @param var
     * @return true if consistent, otherwise return false.
     */
    private boolean AC3(int var) {

        ArrayList<int[]> queue = new ArrayList<>(); //Initialises Queue.

        for(BinaryConstraint c : constraints) { //Inputs arcs into the queue.
            if(c.getFirstVar() == var &&
!allocatedVars.contains(c.getSecondVar())){
                int[] arc = new int[2];
                arc[0] = var;
                arc[1] = c.getSecondVar();
                queue.add(arc);
            }
        }

        while(!queue.isEmpty()) { //While queue is not empty.
            int[] arc = queue.get(0); //Select first arc of the queue.
```

```
            queue.remove(0); //Removes arc from the queue.
            try {
                if(revise(arc[0], arc[1])) {
                        for(BinaryConstraint c : constraints) {  //Inputs new
arcs into the queue.
                            if(c.getFirstVar() == arc[1]) {
                                int[] newArc = {arc[1], c.getSecondVar()};
                                queue.add(newArc);
                            }
                        }
                }
            }
            catch(Exception e) { //If pruned domain is empty.
                return false;
            }
        }

        return true;
    }
```

2.3) Revise – This method has the same layout as the Forward Check revise. The difference is that it returns false if there has not been any change in the domain, true if there has been a change, and it throws and exception if a domain is emptied.

```
    /**
     * Prunes domain of arc(var, futureVar)
     * @param var Current variable
     * @param futureVar The variable connected to the current variable that is to
bre pruned
     * @return false if there where no changes, otherwise return true.
     * @throws Exception when the pruned domain is empty.
     */
    private boolean revise(int var, int futureVar) throws Exception {
        arcRevisions++;

        ArrayList<Integer> acceptableValues = getUnConstraintValues(var,
futureVar);    //Values from supported tuples for arc(var, futureVar).
        if(!acceptableValues.isEmpty()) {
        ArrayList<Integer> newDomain = new
ArrayList<Integer>(varDomains.get(futureVar));   //Get domain.
        for(int i = 0; i < varDomains.get(futureVar).size(); i++) {
```

```
            if(!acceptableValues.contains(varDomains.get(futureVar).get(i)))
{  //If domain contains an unsupported value.
                newDomain.remove(newDomain.indexOf(varDomains.get(futureVar).get(
i)));  //Prune unsupported value.
            }
        }

        if(newDomain.isEmpty()) {  //If the new pruned domain is empty.
            return false;
        }
        else {
        varDomains.put(futureVar, newDomain);   //Replace the domain with the new
pruned domain.
        return true;
        }
    } else {
        return false;
    }
}
```

III. Evaluation

Instances	FC		MAC	
	ASC	SDF	ASC	SDF
4Queens	(Number of Nodes) - 9 (Arcs revised) - 18 (Time Elapsed) - 1ms 1 3 0 2	Nodes - 9 Arcs - 18 Time - 1ms 1 3 0 2	Nodes - 5 Arcs - 21 Time - 0ms 1 3 0 2	Nodes - 5 Arcs - 21 Time – 1ms 1 3 0 2
6Queens	27 96 3 1 3 5 0 2	52 196 5 2 5 1 4 0	19 178 4 3 0 4 1 5	11 77 7 0 4 3 1 2

	4	3	2	4
8Queens	81 366 16 0 4 7 5 2 6 1 3	71 342 10 0 5 7 2 6 3 1 4	50 595 14 0 6 3 5 7 1 4 2	8 71 3 0 2 4 1 3 3 1 6
10Queens	81 415 15 0 2 5 7 9 4 8 1 3 6	161 909 20 0 3 8 6 9 2 5 1 4 7	49 783 20 0 2 5 7 9 4 8 1 3 6	10 129 17 0 2 4 1 3 3 1 6 1 5
FinnishSudoku.csp	109397 5395849 14625 8 1 2 7 5 3 6 4 9	82 3240 56 8 1 2 3 4 5 6 7 9	1769922 75801146 399033 8 1 2 7 5 3 6 4 9	81 3256 103 8 1 2 3 4 5 6 7 9

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

	9 7 4 3 6 8 4 3 8 5 2 6 9 1 7 7 9 6 3 1 8 4 5 2	3 3 2 5 6 8 4 2 8 5 1 2 3 1 4 3 9 2 3 6 4 4 2 3	9 7 4 3 6 8 4 3 8 5 2 6 9 1 7 7 9 6 3 1 8 4 5 2	3 3 2 5 6 8 4 2 8 5 1 2 3 1 4 3 9 2 3 6 4 4 2 3
SimonisSudoku.csp	189 8783 94 7 2 6 4 9 3 8 1 5 3 1 5 7 2 8 9 4 6 4 8	88 3820 65 7 2 6 1 3 4 8 1 3 3 1 4 7 2 8 2 5 6 4 1	211 9237 125 7 2 6 4 9 3 8 1 5 3 1 5 7 2 8 9 4 6 4 8	87 3404 94 7 2 6 1 3 4 8 1 3 3 1 4 7 2 8 2 5 6 4 1

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

	4 3 8 9 2 3 8 5 7 9 4 6 1	4 3 3 9 2 3 8 5 1 1 4 6 1	4 3 8 9 2 3 8 5 7 9 4 6 1	4 3 3 9 2 3 8 5 1 1 4 6 1
Langfords2_3.csp	7 15 1 1 3 2 5 4 4	7 15 1 1 3 2 5 4 4	6 35 2 1 3 2 5 4 6	6 30 1 1 3 2 5 2 6
Langfords2_4.csp	9 28 2 1 3 2 5 4 8 6 6	9 28 3 1 3 2 5 4 8 6 6	8 86 4 1 3 2 5 4 8 6 7	8 63 0 1 3 2 5 2 6 2 7
Langfords3_9.csp	40 550 47 1 3 5 4 7 10 8 12 16	37 536 47 1 3 5 4 7 10 4 8 12	42 3062 203 1 3 5 4 7 10 8 12 16	27 1678 141 1 3 5 2 7 10 2 6 10

	9	4	9	2
	14	9	14	7
	19	14	19	12
	11	4	11	2
	17	17	17	8
	23	23	23	14
	6	4	6	2
	13	11	13	9
	20	18	20	16
	18	4	18	2
	26	21	26	10
	2	2	27	18
	15	6	21	2
	24	15	24	11
	15	24	24	20
	21	6	15	2
	21	16	25	12
	21	26	22	22
Langfords3_10.csp	43	42	134	30
	670	686	9694	2187
	94	71	643	238
	1	1	1	1
	3	3	3	3
	5	5	5	5
	4	4	4	2
	7	7	7	7
	10	10	10	10
	8	4	8	2
	12	8	12	6
	16	12	16	10
	9	4	14	2
	14	9	19	7
	19	14	24	12
	11	4	9	2
	17	17	15	8
	23	23	21	14
	6	4	27	2
	13	11	30	9
	20	18	27	16
	18	4	18	2
	26	21	26	10
	2	29	25	18
	15	4	2	2
	24	13	11	11
	15	22	20	20
	21	4	13	2
	21	16	23	12

	21 22 22 22	26 4 30 2	29 6 17 28	22 2 13 24

IV. Conclusion

Mac 3 enforced global arc consistency and therefore is and better than forward checking. Mac3 helps identify dead ends earlier as compared to the latter.

It can also be viewed from the node traversals and time taken that order of variable selection plays an important role in getting the solution in an optimal time. This can be clearly viewed by looking at the stats of larger constraint problems.(FinnishSudoku as an example).

Although MAC3 is more an efficient way, fc leads forward in langfords constraints. This could imply that the variable selection n the left tree, is satisfying all the constraints faster.

From the above observations I could be said that, although MAC3 is more efficient in a way, FC is still faster when the constraints get satisfied faster on a specific side of the tree.