CS4402-P2

200024015

November 2020

1 Introduction

This project will be divided into 2 sections looking at 2 types of constraint problem solving algorithms for the n-queens problem: Forward Checking and Maintaining Arc Consistency. Each section will provide a pseudocode for the algorithm followed by a more detailed analysis of the Java code which implements the algorithm. The are some features which apply to both algorithms which will be described briefly here.

Both algorithms support both the static heuristic of ascending ordering and the dynamic heuristic of smallest domain first. The type of heuristic is constant throughout the solver once declared in the command line. The algorithms require the queens csp files (e.g. 4Queens.csp) to be stored locally.

To handle variables and thier domains the algorithms use binary arrays. Each algorithm uses an nn matrix to represent the chess board. Methods to handle these arrays can be seen in `ArrayHandler.java`.
0 : represents the index of a value in not in the domain and a square on which a queen could not be placed.
1 : represents the index of a value in the domain and a square on which a queen could be placed. e.g. for a 4 queen problem

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

represents variables with the following domains,

$$\begin{aligned} x_1 &= \{0\} \\ x_2 &= \{2,3\} \\ x_2 &= \{1,3\} \\ x_3 &= \{1,2\}. \end{aligned}$$

To store the assigned variables the algorithms use a pair of lists;
`master` - to represent the assigned values
`varAssign` - to represent the assigned values
e.g. `master = {2,0}`
`varAssign = {3,0}`
represents the assignments: $x_0 = 0, x_3 = 2$.
These are reordered for the final result so: `result = {0,3,1,2}` represents $x_0 = 0, x_1 = 3, x_2 = 1, x_3 = 2$

Using the binary array method, a domain wipeout is represented by a row on the grid summing to 0 since there are no supported values.
e.g for some row n $[0, 0, 0, 0]$ representing $x_n = \{\}$.

Although not implemented in this report, a future improvement could be to exploit the use of symmetry to reduce search time e.g. for 4 queens both $\{1, 3, 2, 0\}$ and $\{2, 0, 1, 3\}$ are both solutions. Therefore once $x_0 = 0$ has been removed from search domain, so could $x_0 = 3$.

2 Forward Checking (FC)

The java code for this algorithm is based on the following psuedocode:

```
ForwardChecking(varList):
    if(assignment complete):
        print assignment
        quit()
    assign var and val according to assignment heuristic.
    branchLeft(varList, var , val)
    branchRight(varList, var , val)

branchLeft(varList, var , val):
    assign(var, val)
    if( resiveFutureArcs(varList, var)):
        ForwardChecling(varList without var):
    else:
        undo pruning()
        remove (var, val) from assignment

branchRight(varList, var , val):
    delete val from var domain
    if size{var domain} != 0:
        if( reviseFutureArcs(varList, var)):
            ForwardChecking(varList)
        else:
            undo pruning()
    else:
        return val to var domain

reviseFutureArcs(varList , var):
    consistency = true
    For all futureVar > var:
        if size{checkSupport(arc(futureVar, var))} = 0:
            return false
        else:
            continue
```

To from command line for ascending heuristic and forward checking 4 queens eg.`java Forward 4 asc`

The java code in fig(1) shows the attributes (lines 8 - 12) and the main (lines 15 - 21) for the Forward class. The class has an attribute of two lists `master` and `assigVar` to track the assigned variables and ensure they are consistent throughout recursions, see explanation in introduction. The ordering type and board dimension n is passed as parameters when called on command line. The ArrayHandler object is imported from `ArrayHandler.java` and is used for array operations in this code.

If the second argument for ordering type isn't `sdf`: smallest domain first or `asc`: ascending then the class returns an error message. The main takes the command line parameters and passes them to the Forward initialization. Forward starts by setting up all of the parameters needed for the algorithm (including filling the board with 1's because at the start all arcs are consistent and each variable has a domain (0..n)). The print statement on line 44 will print out `return` an array of the assigned values to the command line, in the format discussed in the introduction.

The code extract in fig(2) shows the java code implementation for the forward checking algorithm. The forward checking algorithm is recursive as it calls BranchLeft and BranchRight which in turn call ForwardChecking again. Lines 54 - 67 assign the next variable according to the chosen heuristic. The `MinDomainIn` for smallest domain first ordering is a method from `ArrayHandler` and finds the index of the unassigned variable with the smallest domain. e.g if $varList = \{2, 1\}$ and $x_2 = \{3\} x_1 = \{0, 1\}$ then `MinDomainIn` $= 0$.

For ascending ordering, since the varList is initiated in ascending order and variables are always replaced in ascending order, then the algorithm simply pops the first value off the list for the variable.

For assigning the value, the algorithm simply takes the smallest value in variables domain using `FirstOne` method in line 68. `FirstOne` is from `ArrayHandler.java` and returns an integer for the first occurrence of the number 1 in the binary array. If there is no '1' in the array, i.e the domain is empty, then the result is -1. This should not happen if the pruning and branching has been implemented correctly however there is an exception thrown at 69 to prevent the algorithm continuing if this occurs. e.g. FirstOne $([0, 0, 1, 1]) = 2$. The final loop on lines 80 - 82 reorder the variables and their assigns for the result array so that result is in the format discussed in the introduction.

The methods for BranchLeft (left branching) and BranchRight (right branching) (the Java code as shown in fig(3) are complementary and will be discussed together. The left branching is called in line 72 of fig(2) and implements the assignment of a value to a variable and a left branch of the binary tree. Once the left branching algorithm has terminated, the right branching algorithm is called in line 73 of fig(2). The right branching algorithm instead removes the value from the variable's domain. Both left and right branches call the ForwardChecking algorithm (lines 101 and 125) and passes the updated list of unassigned variables with

```java
/**
 * Forward checking constraint solver algorithm for n-queens problem.
 */
public class Forward {

    ArrayList<Integer> master; //master represents the order of assigned values so far.
    ArrayList<Integer> assigVar; //list of assigned variables
    final int n; //dimensions of the nxn chess grid.
    int [][] board; // represents playing board. Binary values where 0 represents a unsupported node and 1 represents an supported node.
    String ordering; //true if smallest domain first, false if
    ArrayHandler arh; //ArrayHandler object for array operations.

    public static void main(String[] args) {

        if(args[1] != "sdf" && args[1] != "asc") {
            System.out.println("Please enter <sdf> (smallest domain first) or <asc> (ascending) for ordering.");
        }
        new Forward(Integer.parseInt(args[0]), args[1]);
    }

    /**
     * Initializes class and starts forward checking algorithm. Also assigns class attributes.
     * @param n - dimensions of nxn grid.
     * @param ordering - ordering type for variables: smallest domain first or ascending.
     */
    public Forward(int n, String ordering) {

        this.n = n;
        this.master = new ArrayList<>();
        this.assigVar = new ArrayList<>();
        this.arh = new ArrayHandler();
        this.ordering = ordering;
        ArrayList<Integer> varList = new ArrayList<>();
        for(int i = 0; i<n ; i++) {
            varList.add(i); //populates initial varList with all variables
        }
        for (int i = 0 ; i < n ; i++) {
            for (int j = 0 ; j < n ; j++) {
                board[i][j] = 1; //fills board with 1's at start.
            }
        }
        System.out.println(ForwardChecking(varList));
    }
```

Figure 1: Forward checking initialization,

pruned domains. Before pruning they store a version of old_board (representing variables domains) which can be restored incase of a domain wipe out in forward checking.

In the BranchLeft method, lines 101 - 106 handles backtracking by restoring the old board as discussed above, then removing the most recent variable assignment. It also removes the value from the variable's domain (line 105) so that variable is not assigned again on this branch to prevent trashing.

In the BranchRight method, lines 101 - 106 handles backtracking by again restoring old_board and returning the value to the variable's domain (line 132).

ReviseFutureArcs in fig(4) iterates through the unassigned arcs, checking that they are consistent with the most recent variable assignment by using the revise method. This is called by BranchRight and BranchLeft fig(3) before calling the ForwardChecking algorithm.

ReviseFutureArcs returns false if there is a domain wipeout on one of the future arc revisions, therefore the Branch algorithm must backtrack and undo latest variable assignment (left) or domain pruning (right). ReviseFutureArcs returns true if it is able to iterate through all the arcs without a domain wipeout, therefore all the future arcs are consistent with the latest assignment.This means that the branching algorithm can continue to forward checking with the new varList.

Revise finds the support for `val` on arc($x_{futurevar}, x_{var}$) as a binary array from `getBinaryArray` on lline 164.

e.g let futurevar $= x_1$ and var $= x_0$ where $x_0 = 2$ and $x_1 = \{0, 2\} \equiv [1, 0, 1, 0]$. Say supportvec $= [1, 0, 0, 0]$ then the new domain for (calculated by MutlArray in line 166) $x_1 = \{0\} \equiv [1, 0, 0, 0]$.

As discussed in the introduction if all the values for a row are zero then there has been a domain wipeout, this is handled in line 167 which checks the array sum.

```java
    public int[] ForwardChecking(ArrayList<Integer> varList){

        try {
            while (master.size() != n) { //continues until all the nodes are assigned. TODO: add a catch in case it is unsolvable.
                int var = 0; //initialize var
                if(ordering == "sdf") {
                    // for smallest domain first call MinDomainIn function to find index of variable with smallest domain
                    // and select this one.
                    var = varList.get(arh.MinDomainIn(varList,board));
                    assigVar.add(var);
                    varList.remove(arh.MinDomainIn(varList,board));
                }
                if(ordering == "asc") {
                    // for ascending variable assignment, because varList was initialized with ascending values, can
                    // take the first variable from the varList.
                    var = varList.get(0);
                    assigVar.add(var);
                    varList.remove(0);
                }
                int val = arh.FirstOne(board[var]); //assign value of first occuring '1' on board, i.e smallest value in domain of var.
                if (val == -1){
                    throw new Exception(); //sums to -1 if 1 is not on the row i.e no more possible assignments.
                }
                BranchLeft(varList,var,val);
                BranchRight(varList,var,val);
            }
        }
        catch (Exception e) {
            System.out.println(e.getMessage());
        }
        int[] result = new int[n];
        for (int i = 0 ; i < n ; i++) { //Reorders master list incase of sdf, where ith value in array corresponds to value of ith variable
            result[assigVar.get(i)] = master.get(assigVar.get(i));
        }
        return result;
    }
```

Figure 2: Forward checking algorithm.

```java
/**
 * Handles left branch of algorithm. Recursive because calls the ForwardChecking method in itself and is called by
 * ForwardChecking.
 * @param varList - list of unassigned varaibles
 * @param var - variable which is being assigned
 * @param val - value assigned to given variable
 */
public void BranchLeft(ArrayList<Integer> varList , int var , int val) {

    master.add(val); //assign value to variable and add to main.
    int[][] old_board = board; //store the previous version of the board before pruning.
    if (ReviseFutureArcs(varList, var, val)){
        ForwardChecking(varList); //recursion step
    }
    else {
        board = old_board;// resets domains if revise future arcs is unsuccessful.
        board[var][val]  = 0; //removes the value from the variables domain.
        varList.add(0,var); // returns the variable to the varList.
        assigVar.remove(assigVar.size()-1);
        master.remove(master.size()-1); //removes the variable from the master list.
    }
}

/**
 * Handles right branching of algorithm. Recursive because calls the ForwardChecking method in itself and is
 * called by ForwardChecking.
 * @param varList - list of unassigned variables
 * @param var - variable which has been assigned on left branch
 * @param val - value assigned to given variable on left branch
 */
public void BranchRight(ArrayList<Integer> varList , int var , int val){

    int[][] old_board = board; //saves old version of the board incase needs resetting for failure.
    board[var][val]  = 0; //removes value from variable domain.
    if(arh.SumArray(board[val]) != 0){ //if domain not empty
        if(ReviseFutureArcs(varList , var , val)){
            ForwardChecking(varList); //recursive step.
        }
        else{
            board = old_board; //if revising future arcs fails, this resets the board.
        }
    }
    else{
        board[var][val] = 1; // if results in empty domain then returns the value to the variable domain.
    }
}
```

Figure 3: Forward checking left and right branch.

```java
    /**
     * Called in left and right branch for checking the future arcs for unassigned variables after an assignment.
     * @param varList - list of unassigned variables.
     * @param var - variable which has just been assigned.
     * @param val - value which has just been assigned to that value.
     * @return
     */
    public boolean ReviseFutureArcs(ArrayList<Integer> varList , int var, int val) {

        boolean consistent = true; //intialization
        for(int i = 0 ; i < varList.size(); i++){ //loop iterates ot check new assignment against all unassigned variables.
            consistent = revise(varList.get(i) , var , val);
            if(!consistent){
                return false;
            }
        }
        return true;
    }

    /**
     * Revises a specific arc to find supported nodes for given value from binary csp.
     * @param futureVar - from revisefuturearcs, forms one end of the arc.
     * @param var - variable being assigned, forms other end of arc.
     * @param val - value assigned to variable, looking for support on arc.
     * @return
     */
    public boolean revise(int futureVar , int var , int val) {

        BinaryCSPReader bscp = new BinaryCSPReader();
        int[] supportvec = bscp.getBinaryArray(n+"Queens.csp" , var , futureVar , val,n);
        board[futureVar] = arh.MultArray(board[futureVar] , supportvec); //finds intersection of two arrays.
        if (arh.SumArray(board[futureVar])== 0) { //if all filled up
            return false;
        }
        return true;
    }
}
```

Figure 4: Forward checking future arcs revision.

3 Maintaing Arc Consistency

The code for this section is based on MAC3 algorithm and the implementation of AC3. The MAC general algorithm is similar to the Forward Checking algorithm seen previously with the key difference being the use of AC3 for enforcing Arc Consistency instead of left and right branching. The pseudocode for this is as follows,

```
MAC3(varList):
     var = varList(0);
     val = min{vardomain};
     if( assignment complete ):
          print assignment
          quit()
     else if (AC3(varList)):
          MAC3(varList without var)
     else:
          undo pruning()
          remove (var, val) from assignment
          delete val from var domain
     if size {var domain} != 0:
          if AC3(varList):
               MAC3(varList)
          undo pruning()
     replace val in domain(var)

AC3(varList):
     Ensure node consistency on all arcs
     Queue = AllArcs(x_i , x_j) for i = (j+1 ..n)
     while size(Queue) != 0:
          remove some arc(x_i , x_j)from Queue
          revise(x_i , x_j):
               for d_i in domain of x_i:
                    if d_i has no support in d_j:
                         remove d_i from domain of x_i
                    if size{d_i} = 0:
                         return false
          add to Queue all arcs(x_h , x_i) (where h != j and h = (j+1..n))
     return true
```

Running from command line for MAC checking for 8 queens using smallest domain first heuristic: `java MAC 8 sdf`.

It is usually essential to check arc consistency at initialization of MAC algorithm, however for the queens problem this is not required because a the start, before a queen is placed, all of the arcs on the chess board are inherently consistent.

The code extract in fig(5) shows how the class MAC was initialized for the algorithm. It is similar to the forward checking algorithm above as it has an attribute of two lists `master` and `assigVar` to track the assigned variables and ensure they are consistent throughout recursions. The ordering type and board dimension n is passed as parameters when called on command line. The `ArrayHandler` object is imported from `ArrayHandler.java` and is also used for array operations in this code.

If the input type isn't `sdf`: smallest domain first or `asc`: ascending then the class returns an error message. The main takes the command line parameters and passes them to the initialization MAC. MAC starts by setting up all of the parameters needed for the algorithm (including filling the board with 1's because at the start all arcs are consistent and each variable has a domain (0..n)). The print statement on line 44 will print out the `result` array of the assigned values to the command line, in the format discussed in the introduction.

The code extract in fig(6) and fig(7) show the java implementation of the MAC3 algorithm. As explained in the code comments this is recursive as it calls AC3 algorithm and then itself (MAC3 algorithm) again as seen in lines 85 and 96. In lines 59 and 67 we can see the implementation of the 2 heuristics smallest domain first and ascending for variable selection as seen in the Forward checking algorithm above.

Lines 87 - 91 and 98 -103 show the steps taken to back track in the event of a domain wipeout (see introduction for sum to zero explanation). These involves removing the value from the master list, assigned variables list and returning to the varList. As seen in the forward checking algorithm, an old version of the board is stored and restored to undo pruning if necessary.

As explained in the introduction, lines 110 - 114 show the ordering of the assigned variables for the final result such that `result = [1,3,2,0]` represents $x_0 = 1, x_1 = 3, x_2 = 2, x_3 = 0$.

The final figure for this section fig(8) shows the java implementation for the AC3 algorithm called by MAC3 above. It checks for arc consistency and prunes domains appropriately according to constraints.

Lines 126 - 130 initializes the variables for the algorithm, including taking the most recently assigned variable and its value from `master` and `assigVar`. Lines 129 - 130 then reset the row of the board according to the variable allocation. For each arc in the queue, lines 140 - 146 check for supports for each value left in the domain. It searches through each value (where board[i][j] = 1) and gets the binary array of the supports. As

```java
/**
 * MAC3 constraint solving algorithm implementing AC3 for n-queens problem.
 */
public class MAC{

    ArrayList<Integer> master; //list of values assigned to varaibles.
    ArrayList<Integer> assigVar; //list of assigned variables
    ArrayHandler arh; //ArrayHandler object for array operations.
    final int n;
    int[][] board;
    String ordering;

    public static void main(String[] args){

        if(!args[1].equals("sdf") && !args[1].equals("asc")){
            System.out.println("Please enter <sdf> (smallest domain first) or <asc> (ascending) for ordering.");
        }
        new MAC(Integer.parseInt(args[0]), args[1]);
    }

    /**
     * Initializes MAC3.
     * @param n - integer dimension
     */
    public MAC (int n, String ordering) {

        this.n = n;
        this.ordering = ordering;
        this.master = new ArrayList<>();
        this.assigVar = new ArrayList<>();
        this.arh = new ArrayHandler();
        this.board = new int[n][n];
        ArrayList<Integer> varList = new ArrayList<>();
        for(int i = 0; i<n ; i++) {
            varList.add(i); //populates initial varList with all variables (0..n) in ascending order.
        }
        for (int i = 0 ; i < n ; i++) {
            for (int j = 0 ; j < n ; j++) {
                board[i][j] = 1; //fills board with 1's at start.
            }
        }
        System.out.println(MAC3(varList).toString());
    }
```

Figure 5: Initialization of MAC3

it iterates through it forms a union of the arrays, using UnionArray from `ArrayHandler.java`. e.g Let X be the new assignment of $x_1 = 0$ and we have worked through the queue and are now checking arc(x_3, x_2) and the board state for the domains is;

$$\begin{bmatrix} X & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

the support for $(x_3 = \{0\}, x_2 = 2)$ equivalent to $[1, 0, 0, 0]$ and the support for $(, x_3 = \{0, 1\}, x_2 = 3)$ equivalent to $[1, 1, 0, 0]$. Then we have $[0, 1, 0, 0] \cup [1, 1, 0, 0] = [1, 1, 0, 0]$ which gives the supported domain of $x_3 = \{0, 1\}$. This is done recursively in lines 141 - 142. Then the intersection of the supported domain with the current domain $[1, 1, 0, 0] \cap [0, 1, 0, 0] = [0, 1, 0, 0]$ which gives the final domain $x_3 = \{1\}$ and is executed by the `ArrayHandler.java` method MultArray on line 148. (This is similar to the Forward Checking revise step in fig(4.)

If there is a domain wipeout (represented by a row summing to 0 as explained previously) on line 152 then the loop will terminate and return false, resulting in MAC3 backtracking.

If the domain has changed then the queue will be updated with new arcs, if they are not already in the queue, as seen in line 152 - 156.

If this loop continues without a wipeout, until the queue is empty then the method will return true since arc consistency has successfully been enforced.

```java
/**
 * Implements the MAC3 algorithm. Recursive as calls itself if AC3 (checking for
 * arc consistency) is successful.
 * @param varList - List of unassigned variables.
 * @return - ArrayList of value assignment. Where ith value in array corresponds
 *. to value of ith variable.
 */
public int[] MAC3(ArrayList<Integer> varList) {

    try {
        int var = 0;
        while (master.size() != n) { //continues until all the nodes are assigned.
            if (ordering.equals("sdf")) {
                // for smallest domain first call MinDomainIn function to find
                // index of variable with smallest domain
                // and select this one.
                var = varList.get(arh.MinDomainIn(varList, board));
                varList.remove(arh.MinDomainIn(varList, board));
                assigVar.add(var); //adds var to assigned variables.
            }
            if (ordering.equals("asc")) {
                // for ascending variable assignment, because varList was
                //initialized with ascending values, can
                // take the first variable from the varList.
                var = varList.get(0);
                varList.remove(0);
                assigVar.add(var);
            }
            //assign value of first occuring '1' on board, i.e smallest value in
            // domain of var.
            int val = arh.FirstOne(board[var]);
            master.add(val);
            if (val == -1) {
            //sums to -1 if 1 is not on the row i.e no more possible assignments.
                throw new Exception();
            }
            int[][] oldboard = board;
            if (AC3(varList)) {
                MAC3(varList); //recursive step
            } else {
                board = oldboard; //undo pruning
                board[var][val] = 0; //assign 0 to remove this value from the variable's domain.
                varList.add(0, var); //return variable to unassigned list. Must be at start incase of asc ordering
                master.remove(master.size()-1); //remove variable from master list
                assigVar.remove(assigVar.size()-1); //removes variable from assigned variable list.
            }
```

Figure 6: MAC3 Part 1

```java
                if (arh.SumArray(board[var]) != 0) { //domain wipeout if true
                    oldboard = board; //store old board incase need to undo pruning.
                    if (AC3(varList)) {
                        MAC3(varList); //recursive step.
                    } else {
                        board = oldboard; //undo pruning
                    }
                } else {
                    varList.add(0,var); //unassign & replace val in domain. Must be at start incase of asc ordering
                    master.remove(master.size()-1); //remove variable from master list
                    assigVar.remove(assigVar.size()-1); //removes variable from assigned variable list.
                }
            }
        }
        catch (Exception e) { //exception handling.
            System.out.println(e.getMessage());
        }
        int[] result = new int[n];
         //Reorders master list incase of sdf, where ith value in array corresponds
         // to value of ith variable
        for (int i = 0 ; i < n ; i++) {
            result[assigVar.get(i)] = master.get(assigVar.get(i));
        }
        return result;
    }
```

Figure 7: MAC3 Part 2

```java
    /**
     * Handles AC3 algorithm as called by MAC3.
     * @param varList - list of unassigned variables.
     * @return - true if successful, false if failure
     */
    public boolean AC3(ArrayList<Integer> varList){

        ArrayList< int[] > queue = new ArrayList<int[]>();
        int var = assigVar.get(master.size()-1);   // variable which has just been assigned
        int val = master.get(master.size()-1); // value which has just been assigned to said variable
        board[var] = new int[n]; //sets the new board for assigned value
        board[var][val] = 0; //adds 0 to location of new placement, removes it from domain.
        for(int i = var+1 ; i <n ;i++){
            queue.add(new int[]{i, val}); //adds all subsequent arcs checking back to one which has just been assigned.
        }
        while(queue.size() != 0) {
            int[] currentarc = queue.get(0); //pop off first arc from queue.
            queue.remove(0);
            int[] oldline = board[currentarc[0]]; //stores old line to check if changes made.
            BinaryCSPReader bscp = new BinaryCSPReader();
            int[] supported = new int[n]; //storage for supported arcs
            for (int i = 0 ; i <n ;i++) {
                if (board[currentarc[0]][i] == 1 ) { //checks support for available values in domain
                    supported = arh.UnionArray(supported,bscp.getBinaryArray(n +
                            "Queens.csp", currentarc[0], currentarc[1], i , n));
                                        //fills board with supported vectors
                }
            }
            //returns binary array with 1 at index of values still supported.
            board[currentarc[0]] = arh.MultArray(board[currentarc[0]],supported);
            if(arh.SumArray(board[currentarc[0]])==0){ //if domain wipeout
                return false;
            }
            if(board[currentarc[0]] != oldline) { //checks for change in domain
                for(int i = 0 ; i < varList.size();i++ ){
                    if(i != currentarc[0]){ //repopulates queue.
                        if(!queue.contains(new int[] {i , currentarc[0]})){ //checks if value already exists in queue.
                            queue.add(new int[]{i , currentarc[0]});
                        }
                    }
                }
            }
        }
        return true; //only reaches this point if empties queue without domain wipeout.
    }
}
```

Figure 8: AC3