A Comparison of MAC3 and Forward Checking Algorithms on Binary CSP Problems

CS4402
November 27th, 2020

Key Data Structures and Methods

The following list provides a basic understanding of the Objects and Functions critical to the software which will be useful in reading this report. There are omitted Objects and Functions that can be found documented in the /src folder of the CS4402-p2 folder.

- **ForwardSolver**
  - *forwardCheck(ArrayList<Variable> varList)*
    - Begins a recursive call for forward checking on unassigned variable list
  - *branchLeft(ArrayList<Variable> varList, Variable var, int val)*
    - Begins recursive call on branching left after assigning var to val.
  - *branchRight(ArrayList<Variable> varList, Variable var, int val)*
    - Begins recursive call on branching right after unassigning val from var.
  - *reviseFutureArcs(ArrayList<Variable> varList, Variable var)*
    - Checks if all unassigned variables in varList have support in the domain of assigned variable var.
  - *private boolean reviseArc(Variable currentVar, Variable futureVar)*
    - Called on all variables in varList in *reviseFutureArcs*
    - Checks that all domain values in an unassigned variable have support in the assigned variable
    - Prunes values that do not from domain
    - Will indicate if the domain of futureVar has been exhausted
  - *undoPruning(ArrayList<Variable> varList, Variable var)*
    - Undoes any pruning done by assigning a value to var
    - Every variable in (varList-var) does *undoPrune*(var)
  - *selectVar(ArrayList<Variable> varList)*
    - Uses desired heuristics to select the next variable for assignment from unassigned list of variables varList
- **MACSolver**
  - *MAC3(ArrayList<Variable> varList)*
    - Begins recursive MAC3 call on an unassigned list of variables varList
  - *initAC3()*
    - Performs an initial AC3 algorithm before search to establish global arc consistency and node consistency.
  - *initQueueAllArcs()*

        - Builds the initial queue of all possible arcs, avoiding repeated arcs. This queue is used for *initAC3()*

    - *AC3(Variable var)*
        - Performs AC3 following the assignment of variable var.

    - *buildQueue(Variable v)*
        - Builds the initialized queue for all arcs incident on variable v. Used for *AC3(*v*).*

    - *undoPruning(ArrayList<Variable> varList, Variable var)*
        - Undoes any pruning done by assigning a value to var
        - Every variable in (varList-var) does unpruneAll*();*

    - *selectVar(ArrayList<Variable> varList)*
        - Uses desired heuristics to select the next variable for assignment from unassigned list of variables varList

- **Variable**
    - deletedVals = HashMap<Variable, ArrayList<Integer>>
        - Mechanism for storing pruned values and accessing them to unpruned
        - Key of the HashMap is the variable who's assignment pruned a value from this integer's domain
        - Value of the HashMap is an ArrayList of all pruned domain values that were pruned by the assignment of a value to the Key.

    - *assign(int i)*
        - Assigns a value i from the variable's domain to be the selected value
        - Prunes all other variables in the variable's domain
        -

    - *unassign(Integer i)*
        - Sets the variable's selected value to null
        - Unprunes all values from variable's domain that were pruned by assigning i

    - *deleteValue(Integer val)*
        - Removes a value val from the variables domain that can be restored, but not unpruned

    - *restoreValue(Integer val)*
        - Adds a value to the variable's domain
        - Val is a domain value that was deleted, not pruned

    - *hasSupport(int value, Variable other)*

        - Checks if there is a domain value in other variable's domain that satisfies a constraint with value from this variable

    - *pruneValue(Variable v, Integer value)*
        - Adds the pruned value to the ArrayList of pruned values associated with variable assignment for v in deletedValues

    - *undoPrune(Variable v)*
        - Adds all domain values pruned by the assignment of variable v back to the domain of this variable

    - *unpruneAll()*
        - Undoes the pruning of all domain values, regardless of the variable assignment responsible.

    - *hasArc(Variable otherVar)*
        - Checks if a binary constraint exists between this variable and otherVar indicating an Arc.

    - *establishNodeConsistency()*
        - Establishes node consistency for this variable by ensuring all domain values satisfy unary constraints for upper and lower bounds.

    - *public int compareTo(Object o)*
        - Used for smallest-domain variable ordering
        - Compares variable objects by the size of their active domains

- **Arc**
    - *matches(Arc other)*
        - Checks whether two arcs represent the same pairing of variables, regardless of order (v1,v2).

    - *isIncident(Variable v)*
        - Checks whether an arc is incident on a variable by returning whether the second variable in the arc pair (v1,v2) is v.

    - *revise() throws MACDomainException*
        - Checks whether every domain value for v1 in the arc has support in the domain of v2.
        - Will prune any values of the domain of v1 if there is no support
        - Will throw a **MACDomainException** if the domain of v1 becomes exhausted during revising.

- **MACQueue**
    - *add(Arc a)*
        - Adds an arc to the back of the queue

- *pop()*
  - Returns the arc at the head of the queue and then removes it from the queue.

- *containsMatchingArc(Arc a)*
  - Checks if an arc that matches *a* (regardless of ordering) already exists in the queue.

- **MACDomainException**
  - Thrown when a domain has been exhausted in AC3. This allows for the algorithm to end early and for backtracking to begin.

Forward Checking

The base code (omitting counters/ prints etc.) for Forward Checking:

```
public void forwardCheck() {
   ArrayList<Variable> varList = new ArrayList<Variable>(getVarList());
   forwardCheck(varList);
   return;
}
public void forwardCheck(ArrayList<Variable> varList) {
   Variable var = selectVar(varList);
   int val = selectVal(var);
   branchLeft(varList, var, val);
   branchRight(varList, var, val);
}

public void branchLeft(ArrayList<Variable> varList, Variable var, int val) {
   var.assign(val);
   if(reviseFutureArcs(varList, var)) {
      ArrayList<Variable> remainingVars = new ArrayList<Variable>(varList);
      remainingVars.remove(var);
      forwardCheck(remainingVars);
   }
   undoPruning(varList, var);
   var.unassign(val);
}

public void branchRight(ArrayList<Variable> varList, Variable var, int val) {
   var.deleteValue(val);
   if(var.getDomain().size() > 0) {
      if(reviseFutureArcs(varList, var)) {
         forwardCheck(varList);
      }
   undoPruning(varList, var);
   var.restoreValue(val);
}
```

The forward checking algorithm was implemented using the **ForwardSolver** class. The class contains a recursive *forwardCheck(ArrayList<Variable>)* method that handles recursive

branching via the *forwardCheck(ArrayList<Variable>, Variable)* and *forwardCheck(ArrayList<Variable>, variable).*

The forward checking algorithm makes use of **Variable** objects, but does not need **Arc** objects for its implementation. For each recursive assignment and un-assignment of values to a variable, the *reviseFutureArcs(ArrayList<Variable>, Variable)* method is called which checks whether or not each value in the domain of any unassigned variables has support in the domain of the assigned variable. If it does not, the value is pruned. This is done via the helper method *reviseArc(Variable, Variable)* which performs all pruning and will indicate whether a domain has been exhausted and backtrack is needed.

Pruning and un-pruning of values in the Forward Checking algorithm is done in the **Variable** class via a HashMap of entries <Variable, ArrayList<Integer>> called *deletedVals*. When a value (Integer) from the domain of a variable is pruned at an assignment of another variable (assignedVar), an entry to the HashMap for assignedVar is amended to include the pruned Integer. This allows for the pruned variables to be associated with a variable assignment which lends itself to an easy undo – pruning process. To undo pruning on a variable level, the assignedVar which is being undone is accessed in the HashMap keyset and all of the elements in the corresponding ArrayList value are added back to the domain. This HashMap method for pruning is also useful in assigning and unassigning values to a given variable. An entry is made in the HashMap for that variable that references itself (this). When an un-assignment occurs, all of the Integers in the corresponding ArrayList are added back to the domain of that value.

When the *undoPruning(ArrayList<Variable>, Variable)* method is called in the **ForwardSolver,** all of the variables that are in the list of unassigned variables will un-prune the values that were pruned by the assignment of the most recently assigned variable.

The forward checking algorithm uses two helper methods – *branchLeft()* and *branchRight()*. For each assignment to a variable, the left branching recursively branches left until a complete assignment is made, or a domain of a future variable is exhausted and un-pruning occurs. At this point, the rest of the values that have not yet been assigned to the original value are added back to the domain of that variable and the value that led to a dead-end is removed. A new value is then selected from the new remaining values if the domain has not been exhausted, and left branching begins again.

Maintaining Arc Consistency

The base code (omitting counters/ prints etc.) for MAC3:

```
public void MAC3() {
   ArrayList<Variable> varList = new ArrayList<Variable>(getVarList());
   MAC3(varList);
   return;
}

public void MAC3(ArrayList<Variable> varList) {
```

```
    if(varList.size() == 0) {
       solutions.add(solution);
       return;
    }
    Variable var = selectVar(varList);
    int val = this.selectVal(var);
    var.assign(val);
    if(AC3(var)) {
       ArrayList<Variable> remainingVars = new ArrayList<Variable>(varList);
       remainingVars.remove(var);
       MAC3(remainingVars);
    }
    undoPruning(varList, var);
    var.unassign(val);
    var.deleteValue(val);
    if(var.getDomain().size() > 0) {
       if(AC3(var)) {
          MAC3(varList);
       }
       undoPruning(varList, var);
    }
    var.restoreValue(val);
 }
```

The base code (omitting counters/ prints etc.) for AC3:

```
 public boolean AC3(Variable var) {
    queue.clear();
    buildQueue(var);
    while(! queue.isEmpty()) {
       Arc currArc = queue.pop();
       try {
          if(currArc.revise()) {
             for(Arc a : arcList) {
                if(a.isIncident(currArc.var1) && !this.queue.containsMatchingArc(a) && !a.matches(currArc)) {
                   queue.add(a);
                }
             }
          }

       catch(MACDomainException mde) {
          return false;
       }
    }

    return true;
 }
```

The MAC3 algorithm also utilizes recursion to complete the search. MAC3 makes use of an AC3 algorithm which is performed prior to any search (*initAC3()*) and then following any variable assignment. AC3 uses a queue **MACQueue** which is ultimately a queue of **Arc** objects. MAC3 utilizes **Arc** objects which indicate an arc between two **Variables**. At each variable assignment, AC3 is run using an initial **MACQueue** of all **Arc** objects incident on the assigned **Variable**.

The AC3 algorithm runs until the queue is emptied and a solution is reached, or a new assignment needs to be made. The AC3 algorithm begins with the variable specific queue initiated with all incident arcs. The arcs are popped from the queue one by one and if there is a revision made (domain value removed), then all additional arcs within the constraint graph which are incident on that revised variable are added to the back of the queue (as long as there is not a matching arc already in the queue). If a variable's domain is exhausted before the queue is emptied, a **MACDomainException** is thrown by the *revise()* function, letting AC3 exit and un-pruning occur.

Un-pruning in the MAC algorithm must occur differently than in Forward Checking. In FC, the pruned variables are stores as values with the assigned variable that resulted in pruning were the key. This key is always equivalent to the assigned variable that began the recursive call for left branching. Un-pruning and branching right involved accessing the values associated with just that single key. In MAC, a value can be pruned by the assignment of a subsequent arc revision so it will be stored under a key value that is not the assigned variable which triggered AC3. Therefore after AC3 fails due to a domain wipeout, pruning is undone using *unpruneAll()* which restores the domains of all unassigned variables to their original domain by un-pruning any pruned values that would have occurred in the AC3 algorithm, regardless of the arc revision that pruned it.

Empirical Evaluation

Solver Correctness and Effectiveness

Our solver was able to produce correct results for N-Queens, Langford's(2,n), Langford's(3,n) and sudoku problem classes. While solutions were reached within a reasonable timeframe (10 minutes) for most provided and generated test instances, there were certain instances that did not reach a solution in a reasonable amount of time. These instances are denoted by "-" in Tables 1 and 2.

It appears that the implementation of MAC was often more efficient than Forward Check in terms of revisions made and nodes searched, but took disproportionately more time to solve. This is most likely due to the use of additional data structures such as **Arc** and **MACQueue** as well as the time cost associated with AC3. Therefore, there are some results for MAC which were not reachable within an allotted 10 minute time period. The analysis will attempt to understand the dynamics of the search algorithms despite these missing results.

Results

Table 1: Solver Results for Forward Checking Algorithms

			Forward Checking				
			Ascending			Smallest Domain	
	Problem	**Solutions**	Nodes	Revisions Made	Solver Time (ms.)	Nodes	Revisions Made	Solver Time (ms.)
	N-Queens							
*	3 Queens	0	10	11	22	8	8	14
	4 Queens	2	30	37	20	30	37	21
	6 Queens	4	244	448	27	226	410	24
	8 Queens	92	3,134	6,315	39	2,634	5,111	53
	10 Queens	724	50,274	119,049	177	38,718	86,100	150
*	12 Queens	14,200	1,145,464	2,938,054	1,822	809,586	1,913,009	1,379
	Langfords							
	L(2,3)	2	48	92	38	48	89	22
	L(2,4)	2	198	470	38	186	424	34
*	L(2,5)	0	822	2,266	32	750	1,980	32
*	L(2,7)	52	20,122	65,822	164	177,744	55,597	157
*	L(3,8)	0	185,722	1,044,094	3,304	82,086	494,710	1,968
	L(3,9)	6	1,107,350	6,659,206	27,722	462,518	2,987,564	15,082
	L(3,10)	10	6,823,174	44,022,011	261,999	2,670,018	18,521,008	144,865
	Sudoku							
	Finnish	1	9,211,586	69,783,882	46,714	22,360	106,606	473
	Simonis	1	670	5,058	152	162	810	76
*	Test 3 (32 Hints)	1	808	6,028	153	162	810	74
*	Test 4 (32 Hints)	1	1,028	8,680	154	162	810	73

Table 2: Solver Results for MAC3 Algorithms

			MAC3				
		Ascending			Smallest Domain		
Problem	Solutions	Nodes	Revisions Made	Solver Time (ms.)	Nodes	Revisions Made	Solver Time (ms.)
N-Queens							
* 3 Queens	0	1	3	14	3	0	13
4 Queens	2	9	21	22	9	21	22
6 Queens	4	225	417	38	232	421	33
8 Queens	92	7,278	7,873	186	7,163	7,660	199
10 Queens	724	204,972	162,529	2,532	207,470	164,824	2,630
* 12 Queens	14,200	6,710,427	4,314,370	142,421	6,931,701	4,515,183	149,433
Langfords							
L(2,3)	2	14	55	26	12	27	32
L(2,4)	2	82	357	28	38	158	28
* L(2,5)	0	533	2,199	108	218	1,063	48
* L(2,7)	52	21,151	82,709	1,098	8,800	48,203	490
* L(3,8)	0	95,064	997,441	34,688	12,040	156,412	4,593
L(3,9)	6	-	-	-	62,243	876,331	35,718
L(3,10)	10	-	-	-	293,668	4,655,137	347,360
Sudoku							
Finnish	1	-	-	-	-	-	-
Simonis	1	236	283	123	81	7	86
* Test 3 (32 Hints)	1	991	1,187	272	81	10	86
* Test 4 (32 Hints)	1	1,380	2,812	284	81	5	82

* Indicates a generated test instance.

Forward Checking vs. MAC

When comparing the searches performed by MAC and Forward Checking, two things were expected:

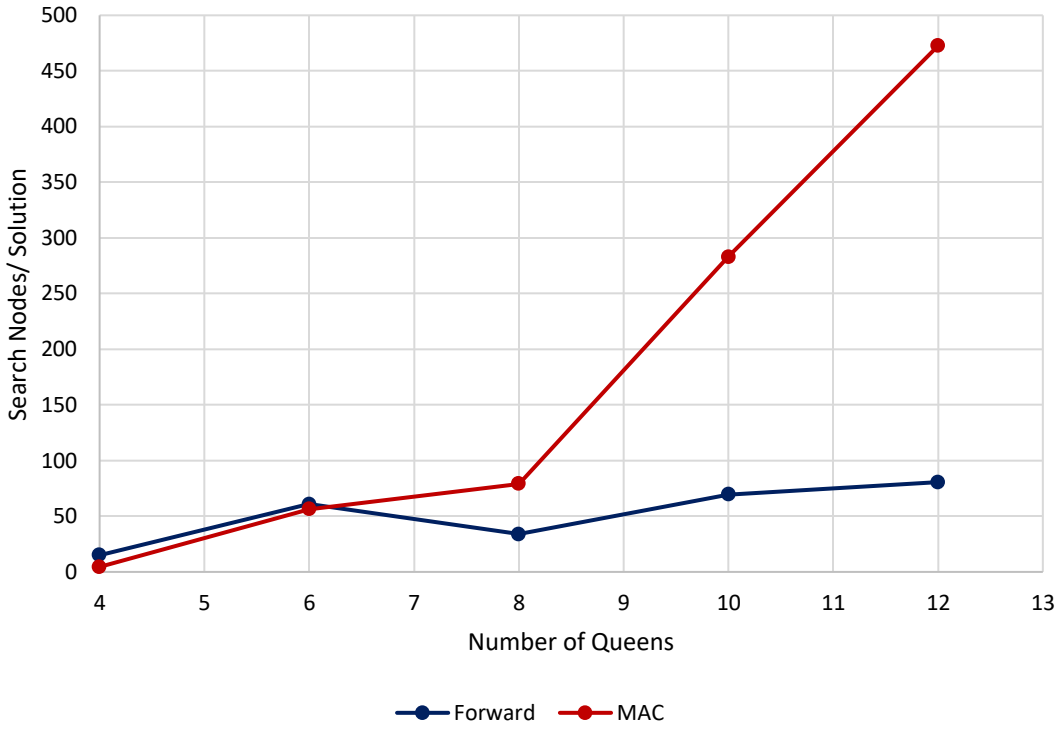
1. MAC spots dead ends quicker than Forward Checking
2. MAC performs less search as a result

In measuring the nodes, revisions and solver time of both search algorithms on the given and generated test instances there were some supporting results, and some results that were not aligned with these expectations (Table 1 & Table 2).

To test the idea that MAC3 can spot dead ends quicker and avoid thrashing, three non-solvable instances were implemented. The 3 Queens, L(2,5) and L(3,8) problem instances have 0 solutions so it is easy to see which algorithm can spot that quicker. In the case of the 3 Queens problem, Forward Checking required 10 nodes with ascending variable ordering and 8 with smallest domain. MAC3 required 1 node with ascending variable ordering and 3 with smallest domain. For the L(2,5) problem, it took ascending order Forward Check 822 nodes and 2,266 revisions to realize there was no solution. MAC3 was able to achieve the result with 65% of the nodes, but a similar amount of revisions. With smallest domain value ordering, MAC3 required over 900 less search nodes to realize the zero solution. For the L(3,8) problem instance, it took 49% less solver nodes and 4% less revisions to complete search in MAC with ascending variable ordering vs. FC with ascending variable ordering. With smallest domain variable ordering heuristics, MAC used 85% fewer search nodes and 68% fewer revisions.

In instances with solutions, the efficiencies of MAC over Forward Checking appeared to be problem specific. In N-Queens instances, MAC outperformed FC in nodes checked/ solution for $N < 8$, but once $N > 8$ the MAC solver nodes increased at a far higher rate than FC and FC performed less search overall to achieve solutions.(Figure 1).

Figure 1: Solver Nodes per Solution for N-Queens Instances (Ascending Variable Ordering)



In the Langford's(2,n) instances, MAC did less search per solution for n = 3 and n=4 instances using ascending variable ordering (Figure 2). Using smallest domain variable ordering however, there was a clear advantage to MAC. MAC used 83% less search nodes and 49% less revisions on average across the 3 solvable L(2,n) instances.

In Langford's(3,n) instances, results were not achieved using ascending variable ordering for MAC. However, MAC with smallest domain variable ordering outperformed FC with smallest domain value ordering – using 88% less nodes and 73% less revisions to complete the search.

Figure 2a: Solver Nodes per Solution for L(2, n) Instances (Ascending Variable Ordering)

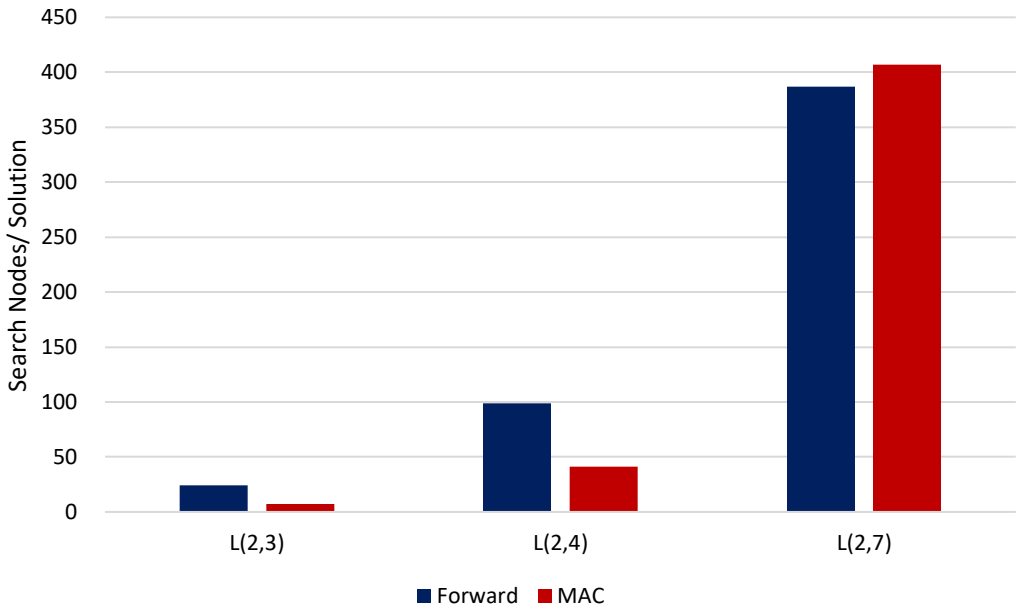
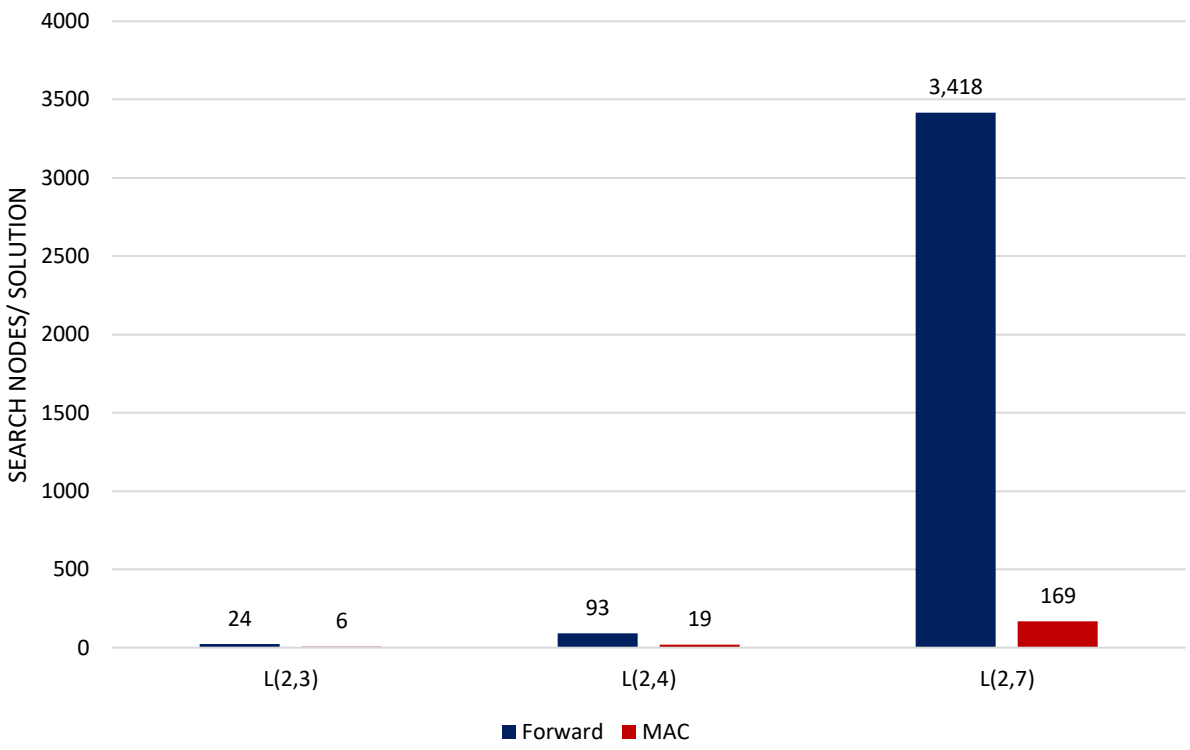


Figure 2b: Solver Nodes per Solution for L(2, n) Instances (Smallest Domain Variable Ordering)



Sudoku

In sudoku instances, it was evident that some sudoku problems were harder than others. The Finnish Sudoku problem took over 9 million nodes for Forward Check with ascending variable ordering to solve, while it only took 670 nodes to solve the Simonis instance with the same configuration. Alternatively, two additional test sudokus were created, both with 32 hints. Using ascending variable ordering, it took 220 more nodes for FC to finish search and 389 more nodes for MAC to finish search on test sudoku 4 vs. 3. Interestingly enough, using smallest domain ordering, the amount of nodes used to solve for Simonis, 3 and 4 were all equal within MAC and FC. It seems that using a heuristic for solving smallest domain first provides an efficient search tree that has the same amount of nodes across instances. The Finnish Sudoku seems to counter this argument as it required over 22,000 nodes even with smallest domain ordering and FC.

Interestingly, MAC outperformed FC in terms of revisions needed for a solution in all 3 sudoku instances by an average of 5,000 revisions (Figure 3b). However, the nodes searched on the 3 instances varied greatly (Figure 3b).

Figure 3a: Solver Nodes of Sudoku Instances (Ascending Variable Ordering)

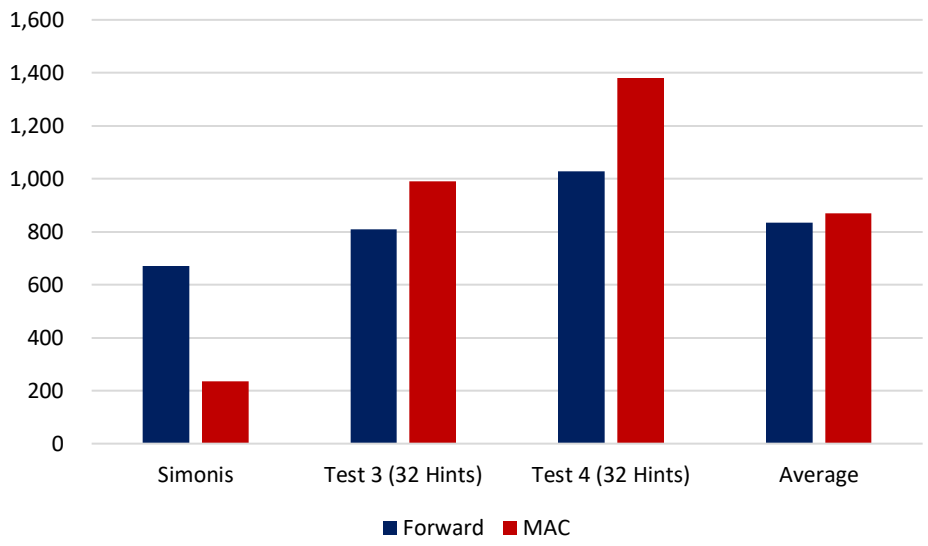
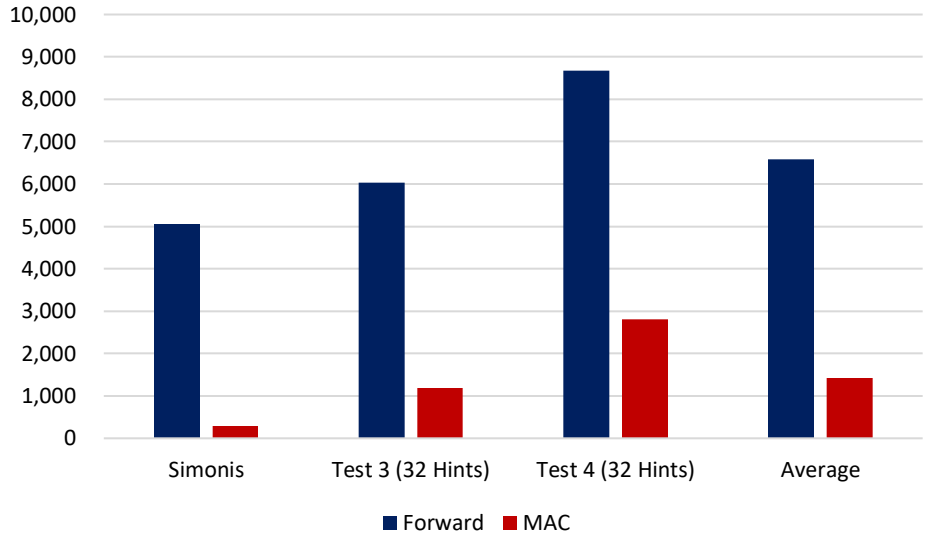


Figure 3b: Revisions Made for Sudoku Instances (Ascending Variable Ordering)



Ascending vs. Smallest Domain Variable Ordering

Both solvers implement the ability to use the dynamic smallest domain value ordering heuristic. If the option for smallest domain is selected, both solver types will re-order the unassigned variables in order of smallest domain before each assignment and subsequent search. The result is that the *.selectVar(ArrayList<Variable> varList)* will always choose the variable with the smallest remaining domain.

In implementing smallest-domain value ordering for MAC and FC, it would be expected that compared to their ascending variable ordering alternate, smallest domain would require :

1. Less time is needed to complete search
2. Less nodes needed to be searched

Heuristics can vary in effectiveness based on problem type, so the difference in impact of smallest domain will also be noted between N-Queens and L(2,n) problem classes.

Table 3: Difference in Search Nodes for N-Queens Instances using Forward Checking

N Queens	Ascending	Smallest Domain	Difference in Nodes
3	10	8	2
4	30	30	0
6	244	226	18
8	3,134	2,634	500
10	50,274	38,718	11,556
12	1,145,464	809,586	335,878

In FC for N-Queens instances there was a decrease in search nodes required that increased as the number of queens (variables) increases (Table 3). However, in L(2,n) instances, there were less conclusive efficiencies in search nodes used. In fact, for the solvable instance with the highest number of variables there was a large increase in search nodes used by smallest-domain variable ordering (Table 4).

Table 4: Difference in Search Nodes for L(2,n) Instances using Forward Checking

Langfords	Ascending	Smallest Domain	Difference in Nodes
L(2,3)	48	48	0
L(2,4)	198	186	12
L(2,5)	822	750	72
L(2,7)	20,122	177,744	(157,622)

For MAC search, N-Queens instances showed small and non-uniform differences in search nodes used between ascending and smallest domain variable ordering. However, in L(2,n) instances, smallest-domain variable ordering decreased search nodes by an average of 46% (Table 5).

Table 5: Difference in Search Nodes for L(2,n) Instances using MAC

Langfords	Ascending	Smallest Domain	Difference in Nodes	% Difference
L(2,3)	14	12	2	14%
L(2,4)	82	38	44	54%
L(2,5)	533	218	315	59%
L(2,7)	21,151	8,800	12,351	58%

Interestingly, for L(3,n) instances the difference between ascending and smallest domain ordering in forward checking showed extreme improvement using MAC (Table 6). There is an evident difference in the impact of a dynamic heuristic for FC between L(2,n) and L(3,n) problem classes.

Table 5: Difference in Search Nodes for L(3,n) Instances using Forward Checking

Langfords	Ascending	Smallest Domain	Difference in Nodes	% Difference
L(3,9)	1,107,350	462,518	644,832	58%
L(3,10)	6,823,174	2,670,018	4,153,156	61%

Conclusions

Both algorithms showed different performance levels in achieving solutions for N-Queens, Langford's(2,n), Langford's(3,n) and sudoku problems. MAC seemed to outperform FC for Langford problems while FC outperformed the MAC solver as N-Queens instances increased in variables. MAC was significantly better for sudoku and L(3,n) problems in terms of reducing nodes and revisions needed, but the complexity of data structures lead to a longer runtime which effected the ability to get solutions for some larger instances.

Variable ordering heuristics also demonstrated problem-specific improvements in both MAC and Forward Checking. FC saw a large improvement in search nodes used in N-Queens when using smallest-domain heuristics versus ascending ordering. MAC showed a huge decrease in nodes needed to solve Langford's(2, n) problems as *n* increases while using a dynamic heuristic. However FC saw a significant reduction in search using dynamic smallest-domain ordering for all L(3,n) instances. All of the results for sudoku problem instances also indicated the benefits of smallest domain heuristics.

Improvements to the MAC3 model could be made to limit algorithmic complexity and overhead with respect to data structures and algorithms. This would allow It is also evident from the N-Queens and Langford's problem classes that symmetry in the solutions could be leveraged in both algorithms to improve efficiency. This marks an improvement that could be applied to the algorithms implemented.

Appendix:

Sudoku Instance for sudoku_test3.csp

_ _ 3 9 _ 8 _ 1 _

_ 2 _ 3 _ _ 9 _ _

_ 5 _ _ _ 4 _ _ 7

8 1 2 _ _ _ _ 4 _

_ 4 _ _ 7 _ 2 6 _

6 _ _ 4 _ _ 3 _ 5

1 _ _ _ _ 5 8 _ 6

_ _ 7 _ _ 1 _ _ 3

_ 9 6 _ _ _ _ 2 _

Sudoku Instance for sudoku_test4.csp

_ 8 _ 3 _ _ 2 _ _

4 _ _ _ 1 _ 7 _ 5

1 7 3 5 _ _ _ 9 4

_ _ _ _ _ _ 5 _ 1

_ _ 6 _ _ 9 _ 8 _

5 _ _ _ 8 1 4 _ _

3 4 _ _ _ _ _ _ 2

6 _ 9 _ 4 7 _ _ 3

_ _ _ 2 6 _ _ _ _