Constraint Solver Implementation

1. Introduction

The assignment asks to implement and compare two constraint solving algorithms for binary Constraint Satisfaction Problems (CSP). This submission achieves both of these as well as three extensions. The system can be used to solve Binary CSPs using both Forward Checking (FC) and Maintaining Arc Consistency (MAC). It is easily extendible, a desired number of solutions can be specified. It can also parse intentional constraints, transform them into BinaryCSP. It can also solve CSP expressed with intentional constraints without transforming them into BinaryCSP. The last two algorithms are not efficient, however they have proved useful in understanding the problems a Constraint Solver runs into and in appreciating all the work a solver does behind the scenes.

The source code for the basic requirements is primarily in the package “binary”. The source code for the extensions is the package “nary” (stands for N-ary, as in N dimensions, rather than binary). Usage instructions can be found in the User Manual (Section 8).

2. Problem Representation

The abstract class “Variable” represents a decision variable in a Binary Constraint Satisfaction Problem (BCSP). It stores its ID, current assigned value (if any), and its domain. It provides the majority of utility functionality used by the solving algorithms. When arcs are revised, the domains of variables are pruned. A BCSP is consistent if all of its variables are consistent. A variable is inconsistent if its domain is empty.

Variable domain pruning is implemented in the method “retainAll”. It accepts a set of values that are supported. It iterates the variable’s domain, removing all values that do not appear in the supported set. Finally, a set containing all the removed values is returned. It can later be used to undo the pruning.

The heuristic by which a value is selected is defined by subclasses of this abstract class. This way, multiple heuristics can easily be implemented and only the Variable instantiation in the BinaryCSP class will have to change. The class “AscendingVariable” implements the “***ascending value ordering***” heuristic. It represents its domain via a hash set. A hash set iterates its elements by their hash code. An integer’s hash code is itself, so they are always iterated in ascending order. Therefore, this data structure provides constant time complexity for adding, removing, and retrieving the smallest integer value.

The class “BinaryConstraint” provides only two methods. They iterate all tuples and determine what values are supported by a given set. One of the methods does this for the left side of the tuples, the other – for the right side. This way, given a set of values that one variable can take, it can return the set of values that the other variable is permitted to take.

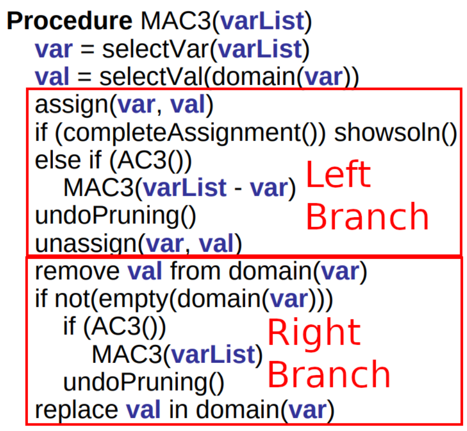
The class “BinaryConstraintStorage” is dedicated to the storage of BinaryConstraints. It uses nested hash maps to associate a pair of variable ids with a constraint. The prime use of this class is to index binary constraints by the IDs of variables. This way the retrieval of a BinaryConstraint instance, based on the variables it affects, happens in constant time.

The class “BinaryCSP” simply provides methods to access binary constraints in constant time and iterate all variables. Solutions to constraint problems are represented and stored as maps between variable IDs and values.

3. Solving Algorithms

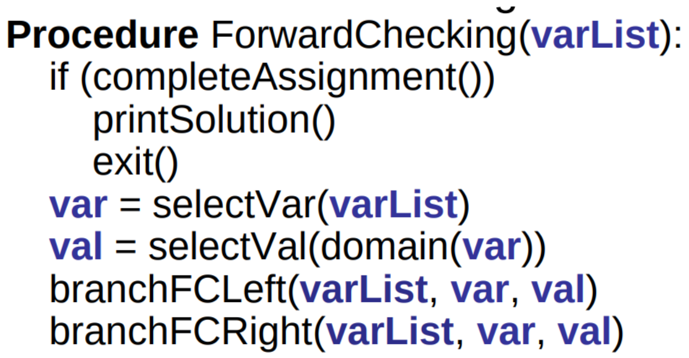
Constraint solving algorithms have a common design: make an assignment, propagate changes through the constraints, repeat and backtrack as needed. Additionally, the two implemented algorithms use 2-way branching. The only difference between the two is how they revise arcs when propagating changes. The abstract class “BinarySolving” in the package binary.algorithms implements the common aspects. It relies on subclasses to implement arc revision.

The algorithms differ when briefly comparing the pseudo code provided in the lectures. However, upon closer review they are the nearly identical.



```
Procedure MAC3(varList)
  var = selectVar(varList)
  val = selectVal(domain(var))
  assign(var, val)
  if (completeAssignment()) showsoln()
  else if (AC3())
    MAC3(varList - var)
  undoPruning()
  unassign(var, val)
  remove val from domain(var)
  if not(empty(domain(var)))
    if (AC3())
      MAC3(varList)
    undoPruning()
  replace val in domain(var)
```

Left Branch
Right Branch



```
Procedure ForwardChecking(varList):
  if (completeAssignment())
    printSolution()
    exit()
  var = selectVar(varList)
  val = selectVal(domain(var))
  branchFCLeft(varList, var, val)
  branchFCRight(varList, var, val)
```

Figure 1. MAC3 pseudo code

Figure 2. Forward Checking pseudo code

The abstract class “SolvingAlgorithm” keeps track of statistical data such as number of nodes, number of arcs, total solving time.

The implementation of “BinarySolving” extends “SolvingAlgorithm” and stores the past and future variables in two respective hash sets. This provides constant time removal and addition – two operations that are performed at every node in the search tree. When a variable is assigned, it is moved from the future to the past set. When back tracking, unassigned variables are moved from the past to the future set. The sets are stored at an instance level, therefore, they do not have to passed as arguments.

The heuristic “***smallest domain variable first***” is implemented with a linear search through the set of future variables. The method keeps track of the variable with the smallest domain and that domain’s size. In the case of ties, the variable that was iterated earlier is chosen. Because the variables are stored in a hash set, they get iterated in order of ascending IDs. Finally, if the loop reaches a variable with a domain size of 1 it breaks early, as there will be no variable with a smaller domain. Consistency is guaranteed before entering selecting a variable.

Binary branching (or 2-way branching) is implemented similar to Figure 2. The main solving method “solve” increments the node counter, selects a variable and a value (using the heuristics described in the previous paragraph and in Section 2), branches left, checks if still needs to search for solutions, then branches right.

The algorithm can arrive at a solution only upon branching left (making an assignment and effectively emptying the future set). If it has not arrived at a solution, it propagates the assignment through the constraints and recurses. When branching right, it uses the utility methods from the Variable class to remove the selected value and check consistency. It then propagates the change in domain through the constraints and recurses. When backtracking, the left branch unassigns the variable, while the right branch returns the value to the domain. This way backtracking several steps will correctly restore the variable states.

To support ***arc revision*** and ***variable domain restoration***, two auxiliary classes were implemented. A “Prune” instance stores a Variable and a set of values that have been removed from the variable’s domain. Therefore, undoing a prune is simply re-adding these values to the variable’s domain. An “Arc” instance stores a *dependent* and a *supporter* variable. It provides a method that reduces the *dependent*’s domain based on the *supporter*’s current domain. It uses the BinaryCSP’s constant time constraint retrieval to find the appropriate constraint between the two. Then, it retrieves all values that the dependent is allowed to take, based on the domain of the supporter (see Section 2). Finally, It returns a Prune object, describing the pruning that has been performed.

Forward Checking (FC) and Maintaining Arc Consistency (MAC) revise arcs differently (see Section 3.1 and Section 3.2). However, they both store Prune instances in a set provided by the “BinarySolving” class. When they return, the common solving algorithm recurses with the pruned domains. Upon backtracking, it uses the stored Prune objects to undo the prunes. The revise method of both FC and MAC accepts a variable and a set to store Prunes. They assume that the domain of the given variable has changed and they prune the domains of other variables respectively.

3.1. Forward Checking

FC revises arcs which are directly connected to the given variable *v*. For each future variable *f*, if there is a constraint between *f* and *v*, an Arc is created such that *f* depends on *v*. That arc is revised (the domain of *f* is pruned) and the Prune is stored. If *f* is inconsistent, the method returns early. As constraint retrieval is done in constant time, the slowest point in the revision is the domain iteration.

3.2. Maintaining Arc Consistency

MAC3 is implemented in this submission, as it is objectively more optimal than MAC1. MAC3 maintains a queue of Arcs that need to be revised. This queue initially contains the Arcs that FC would revise. After every revision, if the domain has been changed, a new Arc is created from every future variable to the one that got changed. That arc is added to the end of the queue. The queue is implemented as a LinkedHashSet. This data structure maintains insertion order while ensuring that all contained elements are unique. The front of the queue is retrieved by getting the first element from the set’s iterator. This way the set is used a FIFO data structure with constant time enqueue and dequeue. It is worth noting that an Arc is created and enqueued only if a constraint between two variables. This avoid object creation overhead and iteration through empty arcs.

4. Empirical Evaluation

Empirical evaluation was performed via the class Benchmark. It runs both Forward Checking and Maintaining Arc Consistency a specified number of times. The mean execution time, number of nodes, and number of revisions are recorded. The data was generated by running the benchmark on N-Queens and Langford's Number Problem instances. The specified sample size was 10. That is, each algorithm was run 10 times on each problem instances and the average results were taken. Full details on how to use the Benchmark class can be found in the User Manual.

Initially I thought that MAC would always outperform FC. However, the generated data proved otherwise. The solving time depends two major factors – variables' domain sizes and number of constraints.

On the one hand, MAC performs more arc revisions, so it prunes the domains more thoroughly. It spends less time iterating search tree nodes but more time revising arcs. On the other hand, FC performs fewer arc revisions, so it leaves more values in the variables' domains. It spends less time revising arcs but more time iterating search tree nodes.

This means that the performances of FC and MAC depend on the CSP instances. FC will perform better on a problem with few any values for the variables but with many constraints. It will quickly go through some of the constraints and iterate the few values. In this case, MAC will spend a lot of time revising arcs for a small reduction in domain size (if any). A prime example of this is the N-Queens problem. Each variable has a domain size of N, and there are a total of $N*(N-1)/2$ constraints.

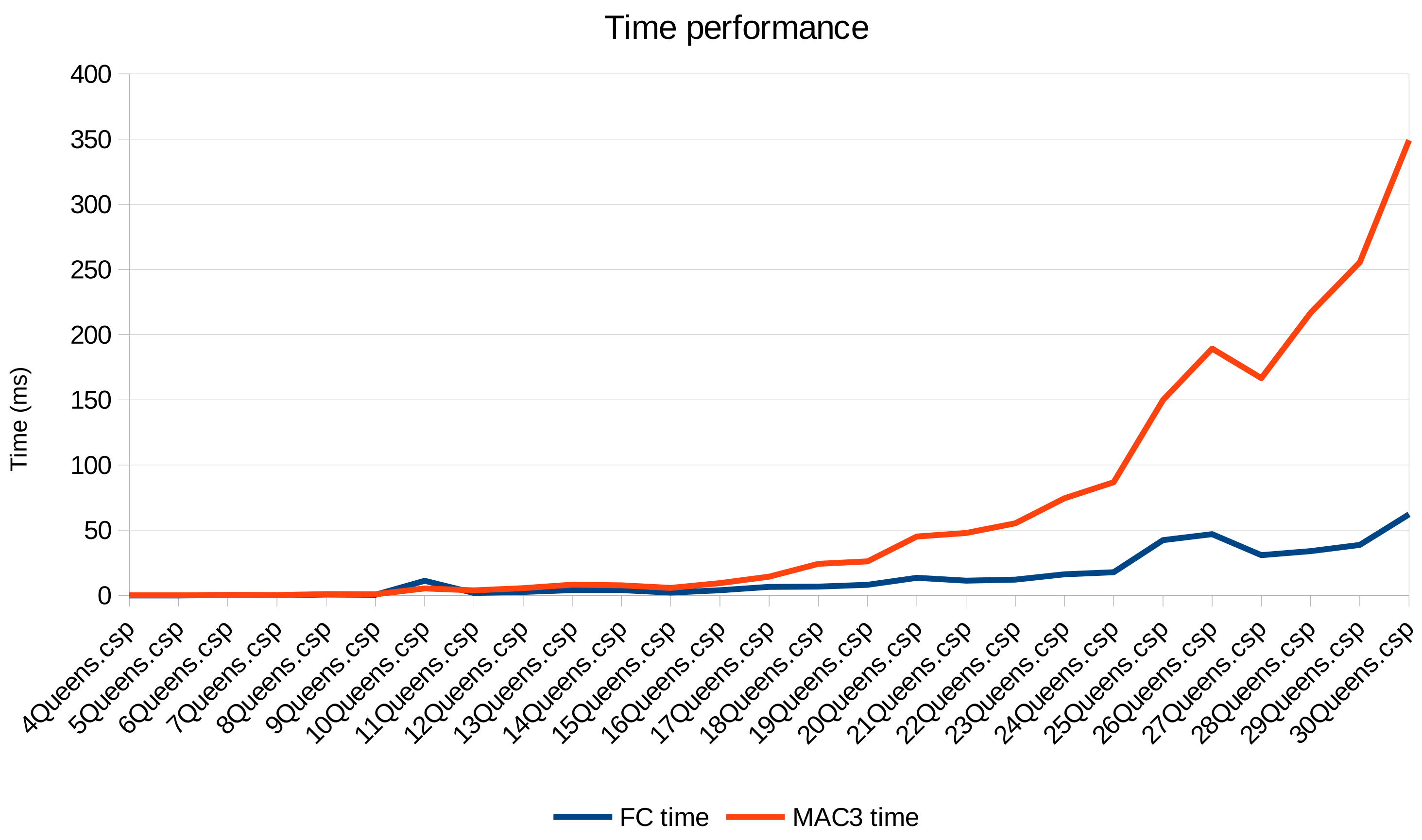


Figure 3. MAC takes more time to solve N-Queens than FC

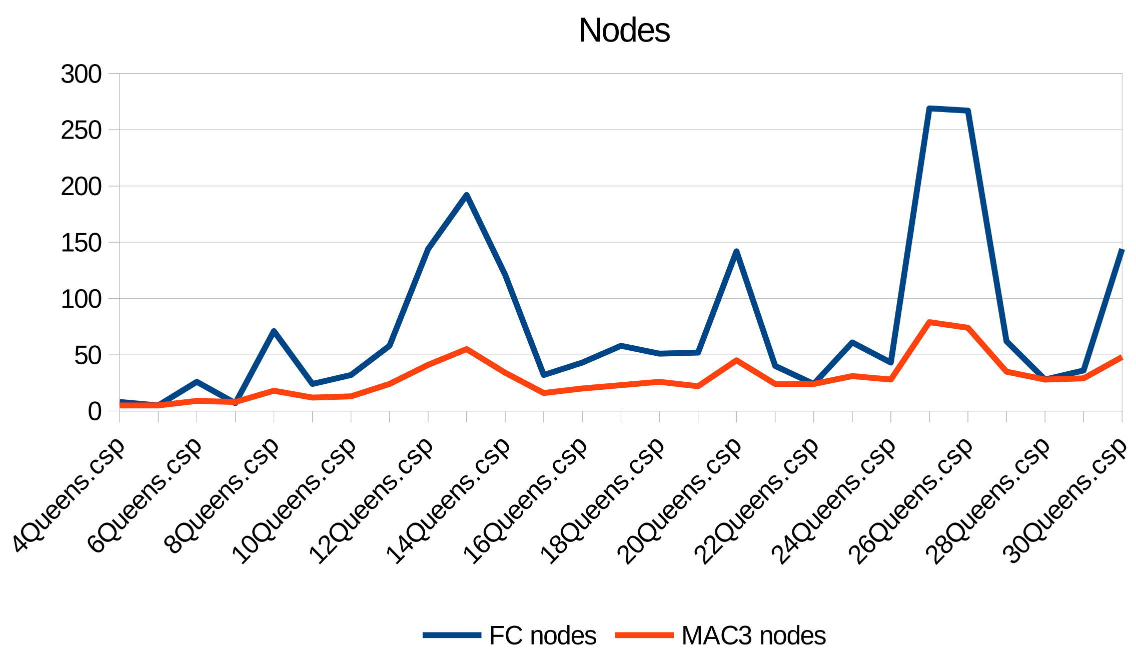


Figure 4. FC iterates more nodes than MAC (N-Queens)

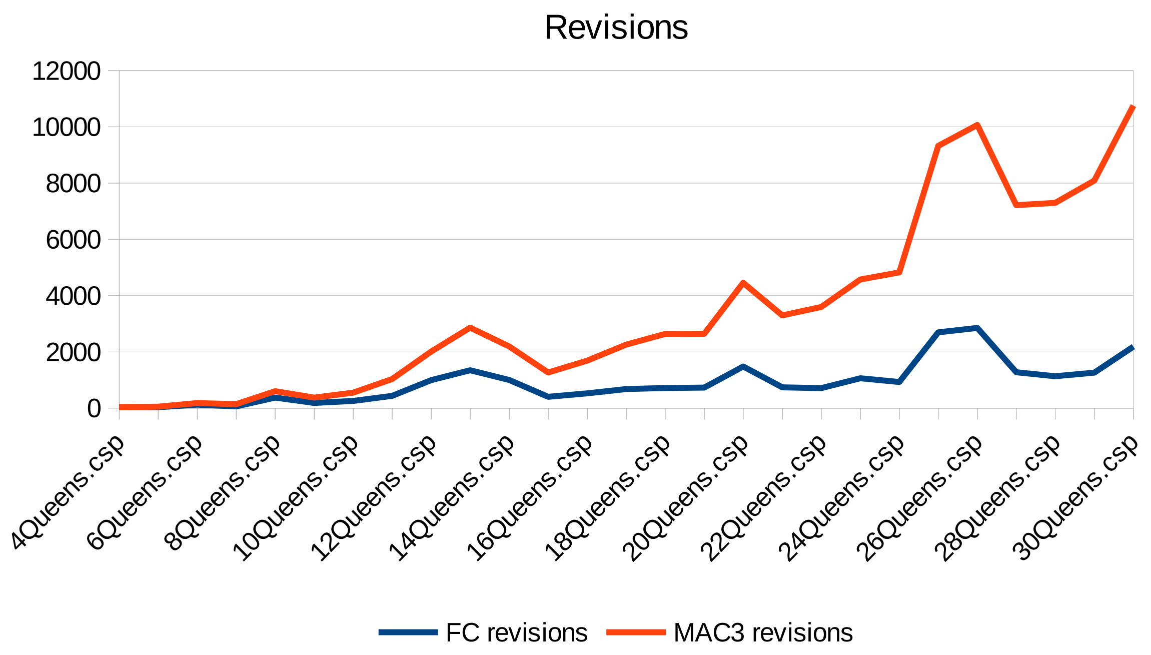


Figure 5. MAC revises more arcs than FC (N-Queens)

Figures 3, 4 and 5 show that MAC spends more time revising arcs and FC spends more time iterating nodes.

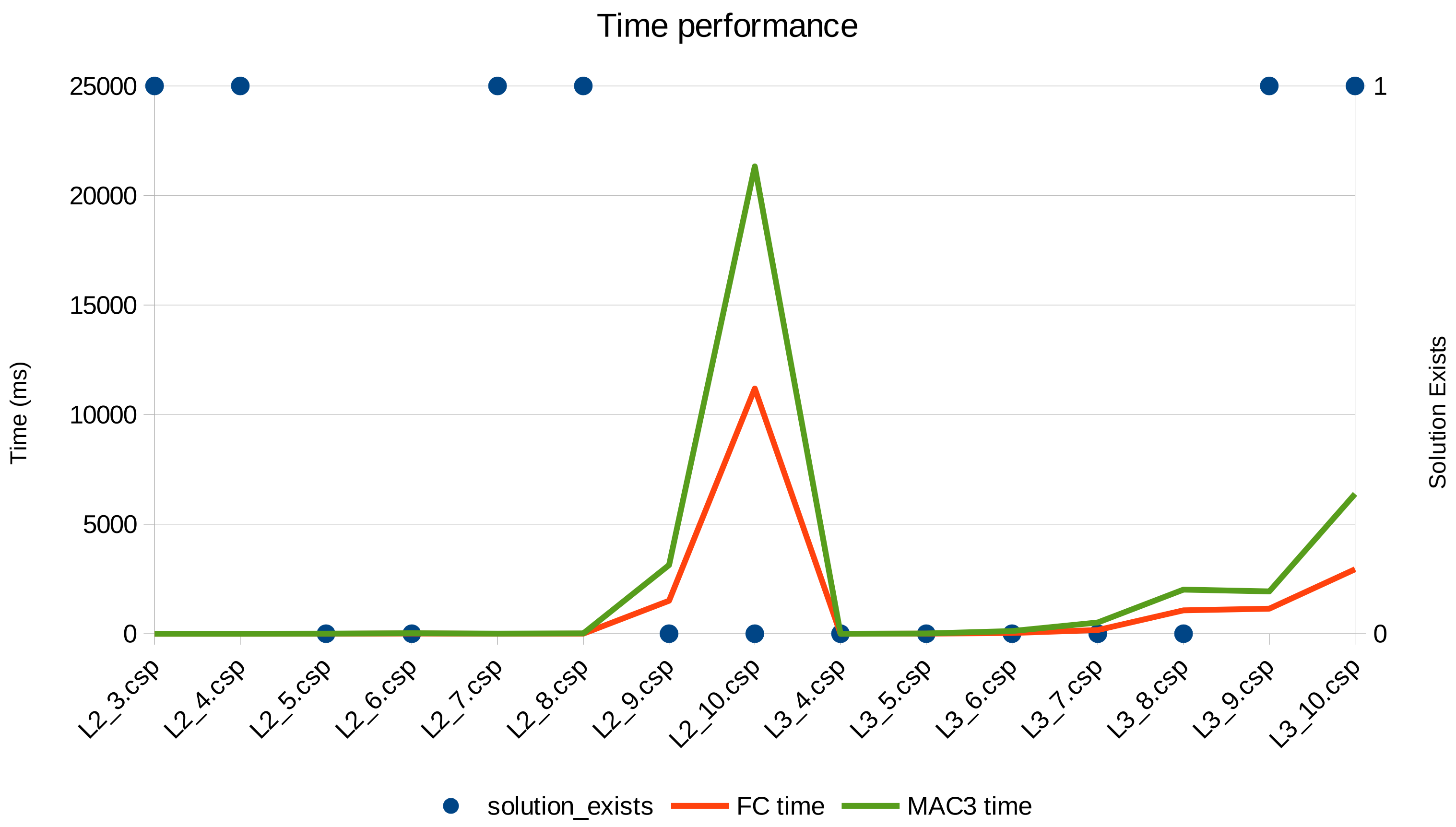


Figure 6. Time taken to solve Langford’s

Figure 6 shows that even when there is no solution (Langford's 2, 9 through Langford's 3, 8), FC still iterates the domains faster than MAC revises arcs.

However, MAC will perform better when there are many values in the variables domain, especially so if the variables are well connected. MAC will revise a lot of arcs but prune many values from the domains. In this case, FC will spend a lot of time going through search tree nodes and reaching dead ends. Those problems are much more complex. The given csp instances and generators cannot be used to show this dependency. I have not had the time to model (let alone solve) such large problems. Some such problem classes deal with non-integer, real numbers (where the domain size is limited by how many bits represent the part after the floating point) and with time constraints (where the domain values are specified in millisecond precision). These two papers[1, 2] discuss approaches to arc consistency in large to infinite-sized domains where Forward Checking falls short.

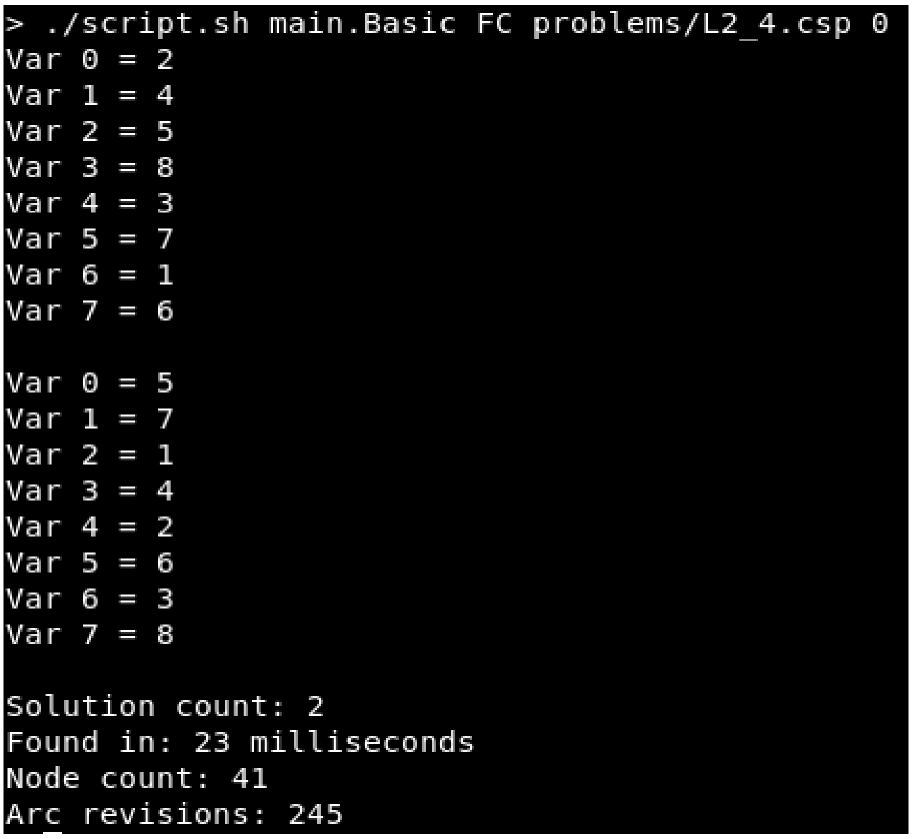
During the development of this system, I compared its performance to that of fellow classmates' systems. I discovered that some of theirs were performing several times faster than mine. After days of analysing, I and a friend of mine discovered that we choose the smallest domain variable in different ways. This system uses hash sets to store variables, so they are always ordered by id. Other implementations used lists which maintain insertion order. Effectively, ties between smallest domain variables were broken in different ways. This implementation uses smallest ID, while other use the variable that has not been selected soon. This demonstrates how useful a good heuristic can be.

5. Extensions

Several extensions were successfully implemented. They improve the usability of the system and offer more insight into what makes a good constraint solver.

5.1. Number of solutions

The systems supports the finding of multiple solutions. A number of solutions to find can be provided upon execution. If not provided, it defaults to 1. If a non-positive amount is provided, the maximum integer value is used (this simulates an attempt to find all solutions). The class "BinarySolving" keeps track of all found solutions in a set. Once the size of the set reaches the user-specified amount, the algorithm terminates. Here is an example output:



```
> ./script.sh main.Basic FC problems/L2_4.csp 0
Var 0 = 2
Var 1 = 4
Var 2 = 5
Var 3 = 8
Var 4 = 3
Var 5 = 7
Var 6 = 1
Var 7 = 6

Var 0 = 5
Var 1 = 7
Var 2 = 1
Var 3 = 4
Var 4 = 2
Var 5 = 6
Var 6 = 3
Var 7 = 8

Solution count: 2
Found in: 23 milliseconds
Node count: 41
Arc revisions: 245
```

The basic (BinarySolving) algorithm with FC (forward checking) is run on Langford's (k=2, n=4), searching for all solutions.

This system assumes that no CSP will have more than $2^{32}-1$ solutions. This is obviously not enough for some CSPs (Sudoku for example has more than 6 sextillion solutions[3]). A good constraint solver will have a way of keep track of solutions that uses more than 32 bits.

5.2. Intentional Constraints

The system has been extended parse, store, transform, and solve CSPs defined with “intentional constraints”. Throughout the code (and these sections of the report), they are referred to as N-ary Constraints (to juxtapose the Binary Constraint definitions). Variables and their related classes are prefixed with “Base” to distinguish them from the binary definitions.

The library mXparser[4] has been used to parse and evaluate mathematical expressions in the input files.

5.2.1. Parsing

The accepted input language for N-ary constraints is very strict and limited. This allows for simpler parsing and object creation. Even so, many problems can be defined in a few lines, without explicitly giving all allowed tuples of values (see User Manual, ECSP syntax). Parsing happens on a line by line basis. Each constant, variable, matrix, or constraint must be specified in exactly one line. Constraints such as *for-all* and *exists* must be specified in exactly 2 lines. Blank lines and lines starting with double slash (//) are ignored.

Constants are stored in a map between names and values. While parsing, occurrences of the constants are replaced with their values. This is similar to pre-processor macros in some programming languages (e.g. C).

Variables are stored in a map between names and BaseVariable instances. This way, when constraints are processed, the respective variables can be retrieved in constant time by name. A BaseVariable is similar to a Variable from the binary representation. It has an id (String name), domain of values, and utility methods.

Matrix definitions are treated as declarations of multiple variables. All combinations of indexes are iterated and a new BaseVariable is created. The names always follow a pattern, such that when a matrix is accessed in a constraint, that access can be translated to a variable name.

When a constraint is read, the parser uses mXparser to determine how many variables appear in it. Unary constraints are constraints that act only on one variable. Nary constraints involve any number (“n”) of variables. These two constraint types are stored in separate sets.

When an *all-different* constraint is parsed, it takes any number of variables names and generates a NaryConstraint for every pair, saying that those should be different.
When a *for-all* constraint is parsed, the next line is multiplied as many times as specified by the for-all. Occurrences of the iterating variables are replaced with the given values. After that each copy of the line is parsed separately.
An *exists* constraint is parsed in much the same way, except instead of treating the line copies as separate constraints, they are joined in a single large, disjunction constraint.
This way of parsing means that *for-all* and *exists* cannot be nested because the lines won’t be multiplied correctly. A better constraint solver will have a way of doing recursive parsing and dealing uniformly with constraints.
Note: *all-different* can be placed inside a *for-all* but not inside an *exists*!

After the whole file is parsed, the BaseVariable instances, the UnaryConstraints, and the NaryConstraints are used to create an ECSProblem (stands for Extended Constraint Satisfaction Problem).

5.2.2. Dual Representation

The "ECSProblem" class provides a method that generates a BinaryCSP out of the ECSProblem instance. It follows the dual representation explained in the lectures. First, all unary constraints are applied to reduce the domains of BaseVariables. Then, for each n-ary constraint, all allowed assignments are generated and stored. Finally, BinaryConstraints are formed between every pair of NaryConstraints that share variables. The class "Assignment" maps variable names (strings) to values (integers). It provides a method to determine if two assignments are compatible – variable names that appear in both assignments must have the same values.

When generating the binary csp, each N-ary constraint is transformed into a Variable. Its domain is {0..K-1} where K is the number of Assignments that satisfy the constraint. By extension, these 'values' are indexes in the list of allowed Assignments. A BinaryConstraint between two such Variables is created by listing all pairs of compatible Assignments. If two N-ary constraints do not share BaseVariables, then a BinaryConstraint is not created.

While this approach to solving intentional constraints works for small problems, it is horribly inefficient, and does not scale at all for large problems. N-ary constraints with a large number of variables often have dozens of satisfying assignments. Generating them and storing them is a big issue. For Langford's (k=3, n=9), as defined in "eproblems/langfords.ecsp", the system exceeds the Garbage Collector's overhead limit. In other words, this dual representation has serious space complexity issues.

To solve intentional constraint problems, an algorithm similar to the one described in section 3 was implemented.

5.2.3. N-ary Constraint Solving

"NarySolving" is an amended copy of "BinarySolving". It follows all the same logic, but uses instances of the intentional definition of the constraint problem. BaseVariables are assigned and pruned. The implementing classes "NaryForwardChecking" and "NaryMaintainingArcConsistency" use also implement the same respective logics. The auxiliary class "BaseVarPrune" defines a prune for a BaseVariable.

The only key difference between N-ary Solving and Binary Solving is in the implementation of the Arc and BaseVarArc classes. While the former prunes with respect to the single BinaryConstraint between two Variables, the latter uses *ALL* NaryConstraints that apply to both the supporter and the dependent. The current Assignment is used to filter the assignments allowed by a NaryConstraint, then the domain is pruned. This operation is repeated for all relevant NaryConstraints, and the prunes are joined together. Iterating and pruning causes a significant jump in time complexity compared to BinarySolving. However, it is still an improvement over Dual Representation.

6. Conclusion

All in all, the system completes all basic requirements and several extensions. The empirical evaluation gave me great insight into how the performance of a constraint solver depends on the problem formulation. The implementation can correctly and efficiently solve Binary CSPs. It can easily be extended to use different variable- and value-ordering heuristics. It can find multiple solutions. It can also parse intentionally expressed constraint problems. It can transform them into Binary CSPs and it can solve them without transforming. The latter two algorithms are not efficient. A better implementation will find a faster domain pruning approach then iterating all assignments that satisfy a constraint.

7. References

[1] Consistency Techniques for Numeric CSPs
Olivier Lhomme, 1993
https://www.ijcai.org/Proceedings/93-1/Papers/033.pdf

[2] Complexity Classification in Infinite-Domain Constraint Satisfaction
Manuel Bodirsky, 2012
https://arxiv.org/pdf/1201.0856.pdf

[3] Mathematics of Sudoku I
Bertram Felgenhauer, Frazer Jarvis, 2006
http://www.afjarvis.staff.shef.ac.uk/sudoku/felgenhauer_jarvis_spec1.pdf

[4] mXparser
http://mathparser.org/

8. User Manual

8.1. Setup

The system requires the mXparser library to compile and run the extension part of it. The basic (binary CSP solving) can be manually compiled without that library. A copy of the jar file of “mXparser” is provided in this submission.

A bash file “script.sh” is provided to simplify the usage of the system. It assumes that the mXparser jar file is in the “libs” subdirectory. This should be the structure of the submission. Feel free to look through the commands in the file. Below is a summary of what commands are accepted.

bash script.sh compile – compiles all of the submitted source code
bash script.sh javadoc – generates JavaDoc documentation for the system, see “javadoc/index.html”
bash script.sh clean – removes the compiled class files and javadoc.
bash script.sh queens N – generates an N-Queens instance in “problems/<N>queens.csp”
bash script.sh langfords K N – generates a Langfords K, N instance in “problems/L<K>_<N>.csp”
bash script.sh sudoku N – generates a Sudoku instance in “problems/sudoku<N>.csp”
bash script.sh <anything else> - runs the system with appropriate class path. See more details in Section 8.3 and Section 8.4

8.2. Generating instances

Problem instances are placed in the directory “problems/”. A good amount of instances are generated and added in this submission. In order to test the system with more instances, use the bash script file with the appropriate arguments. This makes use of the provided java Generators while shortening the command.

8.3. Benchmark

Benchmark is performed from the respective class in the “main” package. The main executable method must be given at least 3 arguments.

The first argument is the number of iterations to perform (sample size). The benchmark will run the binary FC and binary MAC. Each will be run the specified number of times and the average results will be recorded.

The second argument is the number of solutions to find. This is passed to each execution of the solving algorithms.

The rest of the arguments are treated as file names for binary CSPs. They must be formatted such that the provided parser “BinaryCSPReader” can accept them. For each file, each of the algorithms is run the specified number of times.

When running from the command line, one can make use of the ‘*’ syntax to run the benchmark on multiple files. For example: “bash script.sh main.Benchmark 10 1 problems/*Queens.csp” will run FC and MAC ten times on each N-Queens instance in the “problems/” directory.

8.4. Individual Runs

The system can be run on a single instance to display the solution(s). Apart from the Benchmark class, all other classes extend “ArgumentParser” and thus accept the same types of arguments.

Starting with an example: “bash script.sh main.Basic FC problems/FinnishSudoku.csp”. First, “main.Basic” specifies the class that is to be executed. “FC” specifies Forward Checking. And the last argument is the file name of the problem instance.

All script arguments are passed directly to “java -cp <correct class path> <***script arguments***>”. Using the above example, to run MAC instead of FC, simply replace “FC” with “MAC” in the command: “bash script.sh main.Basic MAC problems/FinnishSudoku.csp”.

Optional command line arguments can be provided. An argument after the problem file name is parsed as an integer specifying number of solutions to find. So “bash script.sh main.Basic FC problems/4Queens.csp 10” will attempt to find 10 solutions to the given problem. “bash script.sh main.Basic FC problems/FinnishSudoku.csp 0” will attempt to find up to 2^32 – 1 solutions (see Section 5.1). By default, the system searches for 1 solution.

An argument after the number of solutions is parsed as “logging” level. This is primarily used for debugging. It is an integer that tells what should be logged to standard output while solving. Each component is associated with a power of 2. To determine if a component should be logged, the system performs a bitwise AND on the configured logging level and the associated power. Here is a list of available loggings: BRANCHES = 1; DOMAIN_WIPEOUT = 2; VAR_STATE = 4; ARC_REVISION = 8; ARC_REVISION_CONSTRAINT = 16. So, if the system should log all branches when it takes them and the past and future variables at each node, then a logging level of 5 should be specified. It is the sum of “branches” and “var state”. This way, in bitwise AND, only these two will pass the check. All others will not be logged. By default, the logging level is 0.

The last optional argument tells whether the system should revise arcs and prune variables at the start of execution (before the first variable is selected). Any string other than “true” (case-insensitive) is treated as false. The default value is “true” which causes the system to do initial revision. Given that arc revision often takes more time than iterating nodes, this flag was provided just for completeness. When turned off, a slight boost in speed can be seen. The Benchmark keeps this flag ON, as it is part of the algorithm description from the lectures.

Here are a couple of more examples for execution:

bash script.sh main.DualRepresentation MAC eproblems/simple.ecsp 2 – transform the given ECSP file into a BinaryCSP and attempt to find 2 solutions

bash script.sh main.Nary FC eproblems/allInterval.ecsp 1 8 false – this will run the N-ary solving algorithm on the given ECSP file, looking for 1 solution, logging arc revisions (code 8), but not making the problem globally arc consistent at the start.

8.5. ECSP syntax

As mentioned in Section 5.2.1, this syntax is very limiting. Here is how it works.

If a line starts with a keyword, then keyword specific syntax is expected. Otherwise the line must be a constraint. Every mentioning of an “expression” below, refers to a mathematical expression that can be parsed with “mXparser”. I am not familiar with its full capabilities. It might be able to parse “sum” and others. A single number is also treated as an expression that evaluates to that number.

Constant definition is done with the “const” keyword. It must be followed by the constant name, a colon, and single expression. From then on, any occurrence of the constant is replaced with the evaluated expression. Make sure to put space around the constant reference, so that it can be correctly recognized.
Example: “const MAX : 10”. Defines a constant “MAX” with value 10.

Variable definition is done with the “var” keyword. It must be followed by the variable name, a colon, and two comma separated expressions.
Example: “var x : 0, MAX / 2” . Defines variable “x” with domain from 0 to (MAX/2), inclusive.

Matrix definition is done with the “matrix” keyword. It must be followed by the matrix name, a colon. Then, inside square brackets, pairs of expressions defining indexes in the matrix. Then the string “of values” then two comma separated expressions. In the square brackets, each pair defines a dimension and range of indexes for it. The dimensions must be separated by semi-colons. Each dimension is made up of two comma separated expressions. The expressions after “of values” define the domain of the matrix.
Example: “matrix grid : [1, K ; 1, N * 2] of values MIN , MAX” defines a 2-dimensional matrix, where the first dimension is indexed from 1 to K, the second is indexed from 1 to N * 2, and the domain contains all values from MIN to MAX (inclusive).
The matrix can then be accessed in a manner similar to array accesses in programming languages. “grid[2][10]”. HOWEVER, spaces can only appear inside the square brackets. There must be no spaces between matrix name and outside the square brackets. Bad example: “grid [2] [10]” will NOT be parsed. Also, a matrix cannot be used to index a matrix. The expression(s) inside the sqaure brackets are evaluated during the creation of the constraint. Bad example: “grid[row[1]][col[1]]” will NOT be parsed.
Internally, all matrix accesses are transformed into variable names by replacing the square brackets (“[” “]”) with underscores (“_”). Do not use variable names that end with an underscore “_”.

Expression with no keywords are treated as unary/n-ary constraints.
Examples: “a < 5” becomes a unary constraint; “a != b” becomes a n-ary constraint;
“grid[MIN] + grid[(MIN + MAX)/2] = grid[MAX]” becomes a n-ary constraint.

All Different definition is done with the “allDiff” keyword. It must be followed by comma separated variable names. A N-ary constraint of the type “a != b” is applies for every pair a,b in the given list. A matrix access of the form “allDiff grid[..][5]” is expanded and therefore equivalent to “allDiff grid[1][5], grid[2][5], <and so on>”. This expansion will use the correct indexes (so if the first dimension of grid is defined to be from -4 to 10, then it will go through these values.

For All definition is done with the “forAll” keyword. It’s syntax is similar to variable declaration. After the forAll, a throw-away variable name must be given, and then colon, and comma separated expression for start and end of the for all. The throw-away variable name is replaced in the subsequent line.

Example: “forAll i : 2, MAX” <newline> “arr[i – 1] + i +1 = a[i]”. The values from 2 to MAX are iterated and replaced in the expression, generating multiple constraints.

Exists definition is done with the “exists” keyword. It’s syntax then is the same as “forAll”. Example: “exists x : 1 , MAX” <newline> “arr[x] < x”. The values from 1 to MAX are iterated and replaced in the expression, generating multiple lines that get concatenated into a disjunction. Note: the number of assignments that satisfy exists are tremendous. Avoid usage.

More examples in “simple.ecsp” and “example.ecsp” files under “eproblems/”.