# CS4341 Project 4 - Report

*Constraint Satisfied Problem By Alexander G. Bennett and Paulo Carvalho*

## Introduction

Project 4, aimed at the development of a constraint satisfaction solving algorithm. The context of which is given in the attempt to optimize the packing of items that thieves are planning to take from a store. In this scenario, the items are the variables and each bag a value. For matters of preference and physical background, the packing has to follow constraints which include:

1. Unary Inclusion: A certain item can only be placed in one of the listed bags.
2. Unary Exclusion: A certain item can not be placed in a listed bag.
3. Binary Equality: A pair of items have to be placed in the same bag.
4. Binary Inequality: A pair of items can not be placed in the same bag.
5. Binary Simultaneity: A pair of items cannot be placed in a given pair of bags.

Bags are also subject to a minimum and maximum capacity constraint as well as a lower and upper bound on the number of items per bag. The algorithms used to solve this problem will be presented and analyzed throughout the report.

## Instruction on Running Code

For detailed instruction on running the python script see the README.txt file included with the report.

## Search Algorithm and Heuristics Used

The algorithms used within this project include the following: Backtracking search algorithm, minimum remaining value, least constrained value heuristic, and forward checking.

### Backtracking Search Algorithm:

The backtracking search algorithm is a variant of depth first search. In backtracking, only one successor is generated at a time rather than all successors, each partially expanded node remembers which successor to generate next. The backtracking search algorithm uses only O(m) memory versus the O(b^m) that is used in depth-first search. The pseudo code algorithm used within the software is based on that of the textbook Artificial Intelligence a Modern Approach (3rd Edition).

```
function backtrackingSearch(world)
        return Recursive-Backtracking(world)

function backtracking(world) returns the resulting world
        if assignment is complete then return assignment
        var ← minimumRemainingValue()
        for value in leastConstrainingValue()
                add assignment to world
```

```
add forwardChecking to world

if world remains valid return world
else remove all assignments and return failed world
```

### Minimum Remaining Value Heuristic:

The minimum remaining value heuristic performs one essential function, it chooses the variable with the fewest “legal” values. This heuristic has also been called the most constrained value heuristic for this very reason. Implicitly what it is saying is that; If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately, thus avoiding any pointless searches through other variables which always will fail when X is finally selected.
The minimum remaining value pseudo-code is available below.

```
function MinimumRemainingValue() returns a Variable
    test statements
    for all variables in a list of variables:
        if the constraints of this variable are >= the current maximum constraints
            then:
                change the MaximumConstraints to that of the current variable
                set the MaximumConstraint variable to the current Variable
```

### Least Constraining Value Heuristic

The minimum remaining value heuristic selects a variable in the decision tree. Once this variable has been selected it is important to decide what is the order of the values that will be examined. This is done using the least-constraining value heuristic. The LCV prefers the value that rules out the fewest choices for the neighboring variables. In general this heuristic is trying to leave the maximum flexibility for subsequent variable assignments. The pseudo code for this algorithm is available below:

```
function LeastConstrainingValue() returns an array of values
    if statement for testing
    create a list of values including their constraints
    Sort this list
    for all values in this list
        if these values exist in a list of unassigned values
            then:
                add them to the list of ordered values
    return the list of ordered values.
```

### Forward Checking (Arc Consistency)

Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that has a constraint in common with X and if this relation results in a single assignment location possible it makes the assignment. In practice, this algorithm results in multiple assignments being made in a single round whenever these assignments are the only "inference" possible given the binary constraints to which the most recently assigned variable is connected to.

## Tests to try out program

In order to try out the program there were many tests that were run. The table below illustrates what happens when different heuristics are used together and separately. The values in the table represent the number of iterations that were needed to make it to a final solution. There were seven input files used for this analysis. Only four provided any relevant data, the other three were of practically little use. This is because of the size of the models in both one and two made their solution easy. The third model did not have any actual data in it, although it was provided for use during this project. The next four models represent models of different sizes, input files four through six were provided for this project and input model seven was developed separately. As can be seen several combinations of heuristics were used. The first column after the input column is the backtracking algorithm, without a single other heuristic. Notice that all performance data is provided in terms of iteration required before final assignment.

| Input File | BT | BT + MRV | FC | FC + MRV | LCV | LCV+ MRV | LCV+ MRV+ FC |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 3 | NA | NA | NA | NA | NA | NA | NA |
| 4 | 36 | 15 | 36 | 15 | 19 | 13 | 13 |
| 5 | 1262 | 947 | 1262 | 947 | 1116 | 951 | 951 |
| 6 | 42 | 134 | 24 | 55 | 30 | 115 | 52 |
| 7 (self made) | 10996 | 32786 | 3668 | 1482 | 12118 | 38116 | 1700 |

## Results of Analysis

The first column in the table represents running the software in its most basic operation, only using the backtracking depth first search algorithm. As can be seen the barebones backtracking is one of the most inefficient methods for coming to a CSP solution, but is not the most inefficient method. This may seem counter intuitive but adding the MRV heuristic (only) to the backtracking algorithm in many cases decreased the efficiency of the algorithm. For input files six and seven, which were both larger and more complicated models than five or six the BT + MRV performed worse than the BT alone. It may be too early to make any conclusive results now, but it seems that different heuristics perform better or worse depending on the size or complexity of the model. There is also a chance that the implementation of the MRV heuristic was not optimal thus leading to errors in the data. These possibilities exist for each of the heuristics that were developed.

The third combination of heuristics was backtracking with forward checking. The forward checking heuristic performed substantially better on models six and seven than it did on models four and five. There is a sneaking suspicion that with larger models forward checking performs better than with smaller models. The reasoning for this is not yet certain as the sample size is rather small. Compared to the base model of just backtracking, the addition of forward checking either makes things more efficient or keeps them the same depending on the size of the model and complexity of the model.

The next heuristic to compare on the list is forward checking with the addition of the MRV heuristic. It can be seen from the data that the combination of these two heuristics does substantially better than just backtracking alone. The only place in which this is not true is in input file 6 which most likely has to do with how the MRV heuristic is being implemented and how this implementation handles the complexity of the file.

The heuristic LCV is tested in the sixth column, this heuristic represents the ordering of values once a variable has been chosen. The data shows that the LCV heuristic is an improvement to the backtracking algorithm in all but one case. This is the final input file, file 7, that was developed as an addition to the input files provided. It may be that in the case of this file the complexity and the implementation of the LCV do not correspond well. The combination of LCV and MRV fares better in some categories but much worse in others. For models four and five the combination of these two heuristics does a much better job than backtracking, but when it comes to models six and seven these two heuristics show a drastic decrease in performance.

When combining all three heuristic together, the data shows that the overall performance is dramatically increased. This proves that regardless of the model, it is always best to have more heuristics to choose from, as each heuristic has its own strength and weaknesses.

# Appendix:

## Input Files:

Input file 1

##### - variables
C 14
D 14
##### - values
p 15
q 15
##### - fitting limits
##### - unary inclusive
C p
##### - unary exclusive
C q
##### - binary equals
##### - binary not equals
##### - binary simultaneous

Input file 2

##### - variables
C 10
##### - values
p 10
q 10
##### - deadline constraint
1 1
##### - unary inclusive
C p
##### - unary exclusive
C q
##### - binary equals
##### - binary not equals
##### - binary simultaneous

Input file 3

##### - variables
##### - values
##### - fitting limits
##### - unary inclusive

##### - unary exclusive
##### - binary equals
##### - binary not equals
##### - binary simultaneous

Input file 4

##### - variables
C 6
D 3
E 5
F 8
G 15
H 12
I 7
J 19
K 4
L 9
##### - values
p 40
q 55
##### - fitting limits
5 5
##### - unary inclusive
##### - unary exclusive
##### - binary equals
##### - binary not equals
##### - mutual exclusive

Input file 5

##### - variables
C 6
D 3
E 5
F 8
G 15
H 12
I 7
J 19
K 4
L 9
##### - values
p 32

q 25
r 20
x 14
##### - fitting limits
2 3
##### - unary inclusive
C q r
D p q
H q r x
K p q
##### - unary exclusive
##### - binary equals
##### - binary not equals
##### - mutual exclusive

Input file 6
##### - variables
C 6
D 3
E 5
F 8
G 15
H 12
I 7
J 19
K 4
L 9
##### - values
p 32
q 25
r 20
x 14
##### - fitting limits
2 3
##### - unary inclusive
C q r
D p q
H q r x
K p q
##### - unary exclusive
E p q r
F q

G p r
I x
##### - binary equals
L J
I H
##### - binary not equals
D K
I E
H F
##### - mutual exclusive
D K p r
I H q r
L F q x

Input file 7

##### - variables
A 12
B 15
C 6
D 3
E 5
F 8
G 15
H 12
I 7
J 19
K 4
L 9
M 12
N 20
O 2
P 3
##### - values
p 32
q 25
r 20
s 5
x 14
y 27
z 32
##### - fitting limits
2 3

##### - unary inclusive
C q r
D p q
H q r x
K p q
##### - unary exclusive
E p q r
F q
G p r
I x
A P z
##### - binary equals
L J
A B
I H
M N
P O
##### - binary not equals
D K
I E
H F
##### - mutual exclusive
D K p r
I H q r
L F q x
A D x y
B O p y