Laboratory 2

Introduction to Artificial Intelligence

Constraint Satisfaction Problems

by Ernesto Vieira Manzanera. Student id: 251663

1. Goal of the project

The purpose of this paper is analysing the resolution of different types of problems that can be solved with a Constraint Satisfaction Problem (CSP) approach. The algorithms used here will be properly described, and the methods used for getting the results that are presented.

2. Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 subgrids that compose the grid (also called "boxes", "blocks", or "regions") contain all of the digits from 1 to 9.

In order to approach this problem as a CSP we first need to identify its elements:

Variables:

A variable x_{ij} $i, j \in \{0...9\}$ placed at row i and column j is defined as a blank cell that is yet to be assigned a value.

Constraints:

Condition that affects a variable and affects its domain. In this problem, we can distinguish among 3 different types of constraints:

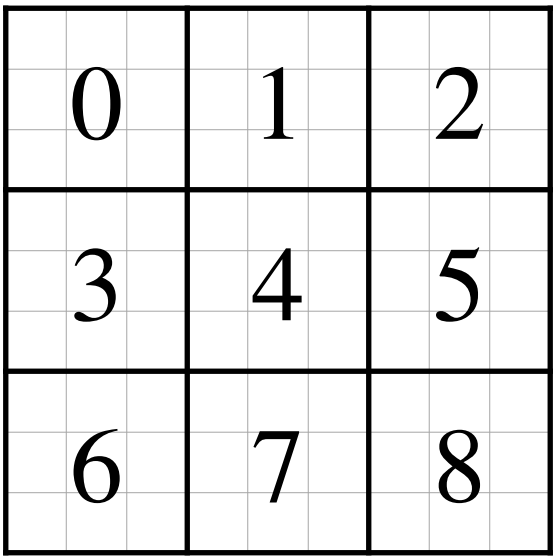
- Row-constraint: A variable x_{ab} is row-constrained such that it cannot be assigned a value already in the row.

$$x_{ab} \neq x_{aj} | j \in \{0...9\} - \{b\}$$

- Column-constraint: A variable x_{ab} is column-constrained in such that it cannot be assigned a value existent in the column.

$$x_{ab} \neq x_{ib} | i \in \{0...9\} - \{a\}$$

- Grid-constraint: A variable x_{ab} is grid-constrained such that it cannot be assigned a value existent in the same subgrid. Each sub grid is indexed with a number following the diagram:



And the indexes are calculated with the formula

$$g(n) = (n \mod 9)//3 + ((n//9)//3) * 3$$

We define then the constraint as:

$$x_{ab} \neq x_{ij} \forall i, j \mid g(i * 9 + j) = g(a * 9 + b)$$

Domains:
Set of values that are assignable to a variable. Initially, the domain for each variable is the general domain:

$$G = \{1,2,3,4,5,6,7,8,9\}$$

We will denote the domain of the variable x as D_x. Applying the constraints previously described, we can reduce the domain of each variable to be consistent at each iteration. In such situation, the domain of a variable x would be defined as:

$$D_x = G - R_i - C_j - S_{ij}$$

where R_i is the set of values already assigned in the i-th row, C_j is the set of values already assigned in the j-th column, and S_{ij} is the set of values assigned the sub grid the variable x_{ij} belongs to. Such formula will be used to apply the Forward Checking method.

A. Backtracking:
Backtracking makes use of a depth-first search algorithms to iterate over the space of solutions and find a consistent one. At each iteration the algorithm will test a new variable and iterate over its domain, making an assignment that is consistent with the constraints affecting such variable, and then iterating again over the next variable. If the domain of a variable is checked on the whole and no feasible partial solution has been found, it means that a previously made assignment makes the problem inconsistent and the algorithm backtracks to that point and assigns a new value to the variable.

A raw backtrack algorithm has been implemented to this problem, with the following schema:

```
backtrack (sudoku, variables, i):
      _sudoku = copy(sudoku)
      if i = len(variables):
            if checkSolution(sudoku): return sudoku
            else: return None

      variable = variables[i]
      for j in {0…9}:
            if isConsistent(sudoku, var = j):
                  sol = backtrack(_sudoku, variables, i+1)
                  if sol != None: return sol
```

We will run the program on the following sample files:

Puzzle	Difficulty
Puzzle14	2.0
Puzzle24	4.0
Puzzle36	7.0
Puzzle42	8.0
Puzzle45	9.0

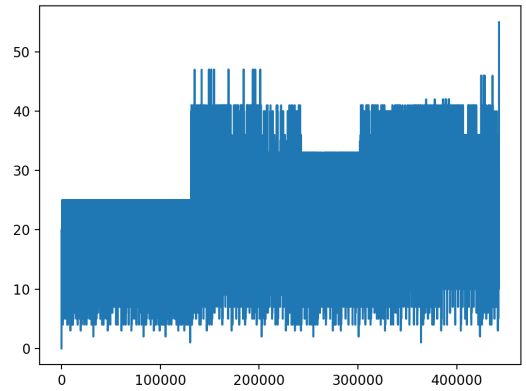
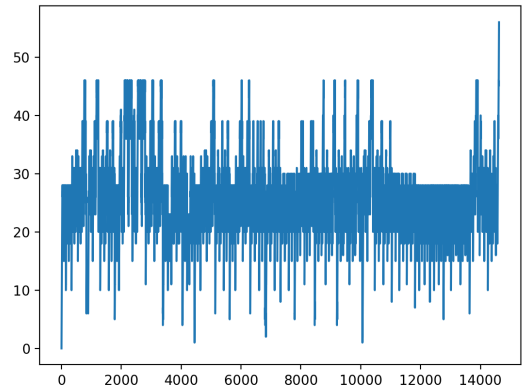
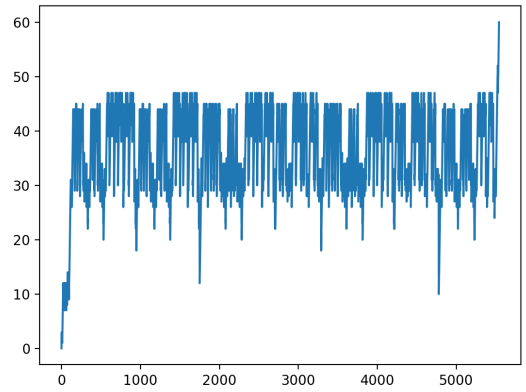
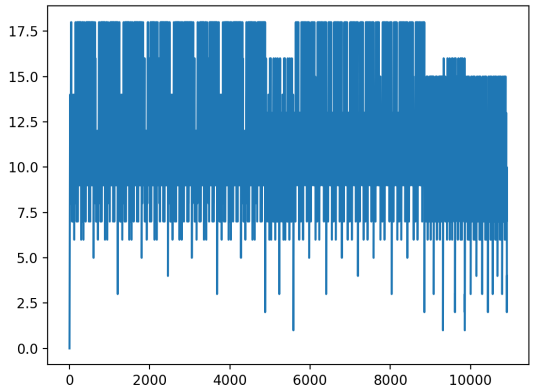
Obtaining the following results:

```
Sudoku number 14 Difficulty 2.0
Elapsed time: 1.2025010585784912

Sudoku number 24 Difficulty 4.0
Elapsed time: 14.855921030044556

Sudoku number 36 Difficulty 7.0
Elapsed time: 0.5016789436340332

Sudoku number 42 Difficulty 8.0
Elapsed time: 0.18674397468566895

Sudoku number 45 Difficulty 9.0
No solution found
Elapsed time: 0.4556150436401367
```

We can observe the range of time values that the simulation yields. Having performed a simulation on the whole set of puzzle, I have decided to take these 5 samples corresponding to different difficulties.

One value that stands out it the one associated with the puzzle 24. Such a high value means that the algorithm finds it difficult to evolve towards a solution. If we graph the

number of variables that have been assigned a value at each iteration, we can see that the simulation por puzzle24 gets stuck at finding a best solution from a certain point.

Puzzle 24	Puzzle 36
	
Puzzle 42	**Puzzle 45**
	

These situations are represented with the intervals where the graph is flattened because it is checking the same variable (or close ones) in a huge range of its domain.

With this visualisation, a good algorithm would be the one that detects inconsistencies early in the simulation so as to avoid returning back to previous variables very often. This can be seen as the peaks in the graphs: the higher the number of peaks is, the more often the algorithm finds an inconsistency and backtracks to previous variables. Such an optimal scenario must look like a set of peaks at earlier iterations, and then a linear-like curve for the latest ones. We will see examples of this phenomenon later in this paper.

Lastly, it is also interesting to point out the efficiency in determining that a problem is not solvable. These kind of situations forces the algorithm to find an inconsistency in the whole board trying numerous combinations, which can turn out to be hazardous for big problems.

B. Backtracking with Forward Checking:

Having a raw Backtracking implementation of the sudoku solver, I included a Forward Checking implementation for the same problem. The algorithm behaves as follows:

```
ForwardChecking(variable):
      for v in row(variable):
            newDomain = createDomain(v)
            if newDomain == {}: return -1
      for v in column(variable):
            newDomain = createDomain(v)
            if newDomain == {}: return -1
      for v in square(variable):
            newDomain = createDomain(v)
            if newDomain == {}: return -1
```

The algorithm is implemented using the function createDomain(variable), which iterates over the variables constrained by the value of variable in order to reduce their domains. If a domain is set empty, the algorithm assumes it has found an inconsistency and return -1. Backtrack is then performed with a different value.

The functions row(variable), column(variable) and square(variable) return the location of the current variable in terms of its row, column and sub grid.

To create the new domain, we apply the constraints previously defined:

$$D_x = G - R_i - C_j - S_{ij}$$

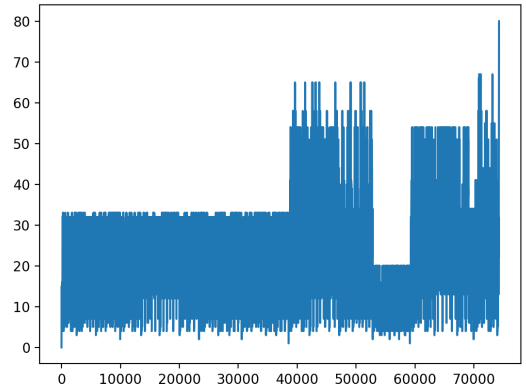
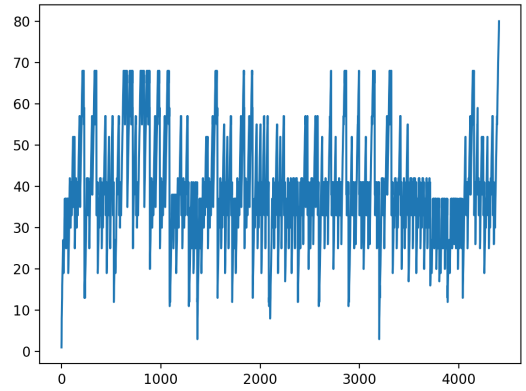
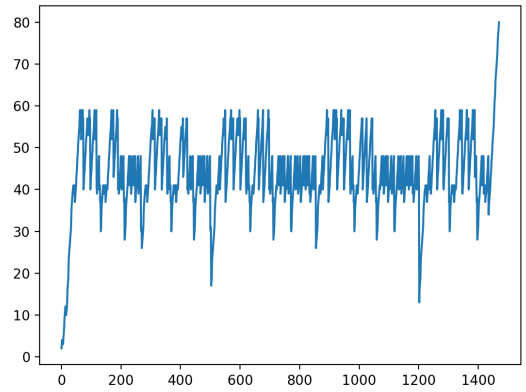
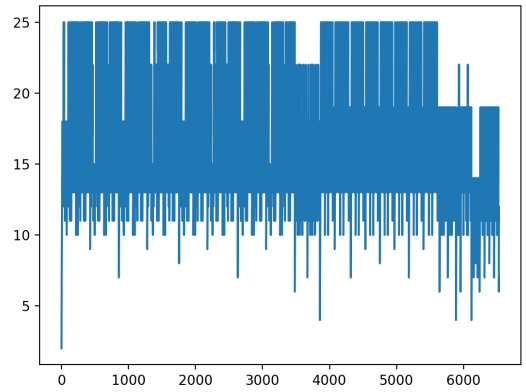
Given that a variable has already a defined domain, we only need to delete the value that has been assigned to the current variable:

$$D_x^{'} = D_x - \{a\}$$

where a is the value assigned to the variable.In order to keep track of the elements affecting each variable, a dictionary is kept for the rows, columns and sub grids with the elements already set in them.

The performance results for the chosen puzzles are the following:

```
Sudoku number 14 Difficulty 2.0
Elapsed time: 18.698641777038574

Sudoku number 24 Difficulty 4.0
Elapsed time: 137.6243052482605

Sudoku number 36 Difficulty 7.0
Elapsed time: 6.912040948867798

Sudoku number 42 Difficulty 8.0
Elapsed time: 1.9577219486236572

Sudoku number 45 Difficulty 9.0
No solution found
Elapsed time: 12.891703128814697
```

Puzzle 24	Puzzle 36
	
Puzzle 42	**Puzzle 45**
	

Comparing the results with the previous point, we can observe that the time statistics are quite worse. In examples such as puzzle 24, the time has multiplied by 9,78, which at first looks like it is a downgrade of our algorithm. However, we must also consider the number of iterations made.

In comparison with the previous points, the reduction of iterations is very significant:
- For puzzle 24: 400.000 to 70.000. Reduction of 82,5% of iterations.
- For puzzle 36: 14000 to 4000: Reduction of 70% of iterations.
- For puzzle 42: 5000 to 1400: Reduction of 72% of iterations.
- For puzzle 45: 11000 to 6500: Reduction of 40% of iterations.

The difference in iterations turns out to be important, although the increasing of time spoils the improvement. In spite of that, I must attribute such time differences due to the implementation of the Forward Checking, not to the algorithm itself.

C. Forward Checking with Heuristic

Due to the huge amount of time consumed in the previous simulations, the inclusion of an heuristic for choosing the variables to assign may reduce the computation time. This

heuristic will consider the number of elements in the domain of each variable, and sort the variables vector in increasing number of values in domain, choosing first those variables that are most constrained.

This will be done before starting the simulation with the command:

```
variables.sort(key = lambda x: len(x.domain))
```

The statistics yielded from the simulations are as follow:

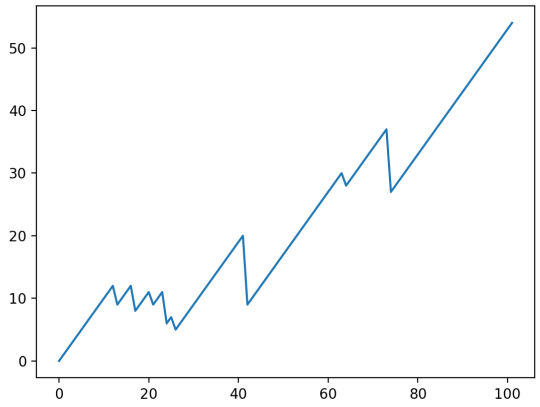
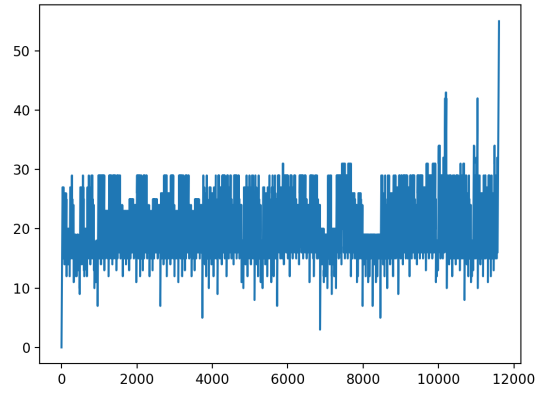
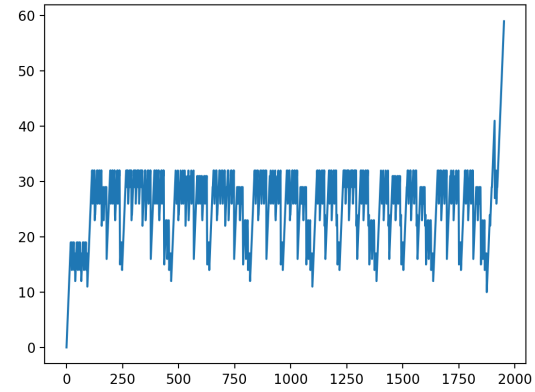
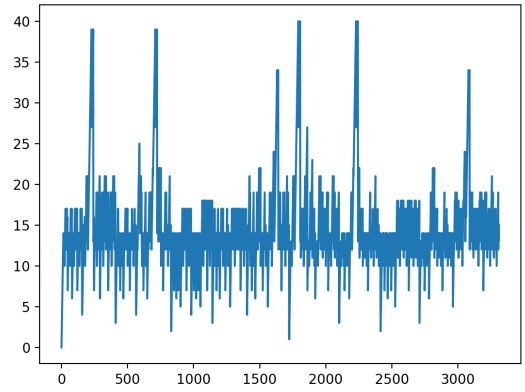
Sudoku number 14 Difficulty 2.0
Elapsed time: 18.698641777038574

Sudoku number 24 Difficulty 4.0
Elapsed time: 0.18201708793640137

Sudoku number 36 Difficulty 7.0
Elapsed time: 18.792589902877808

Sudoku number 42 Difficulty 8.0
Elapsed time: 3.5461039543151855

Sudoku number 45 Difficulty 9.0
No solution found
Elapsed time: 7.303394079208374

Puzzle 24	Puzzle 36
	
Puzzle 42	**Puzzle 45**
	

The runtime statistics have decreased drastically with respect to the bare forward checking method, but have not reached the values of the raw backtrack for some of the cases. However, we can observe that puzzle 24 has been solved in 0.18 seconds, while the backtrack had just managed to do it in 14,85 seconds.

Furthermore, the non-solution puzzle i.e puzzle 45, shows a smaller rate of wide backtracking than the only-forward-checking version, as it is shown in less number of high peaks in the plot.

3. Fill-in

Fill-Ins, also known as Fill-It-Ins or Word Fills, are a variation of the common crossword puzzle in which words, rather than clues, are given. The solver is given a grid and a list of words. To solve the puzzle correctly, the solver must find a solution that fits all of the available words into the grid.

The approach to this problem as a CSP starts with a proper definition of the problem elements:

Variables:

In the fill-in puzzle, each word is considered to be a single variable. A variable is defined as a tuple of 4 values:

$$w = (l, v, t, r)$$

where:

- l is the length of the word
- v is the value assigned to the word
- t is the type of the word. We distinguish between horizontal and vertical words. Thus,

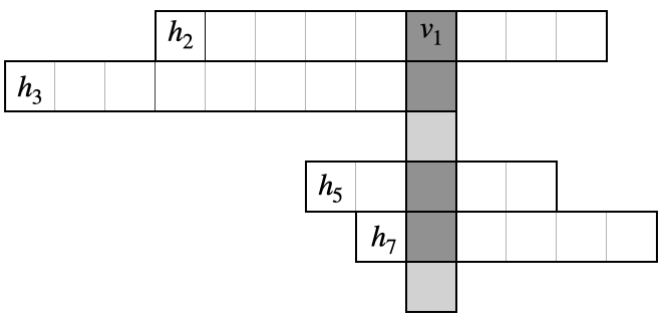
$$l \in \{'h', 'v'\}$$

- r is a two-dimensional array that sets the position of the word in the game board. r points to the first letter of the word.

Constraints:

Constraints are defined as intersection between words. Thus, every constraint will have two words in its scope: one vertical word and one horizontal word. The constraint defines which character from the first word must coincide with which character of the second one.

We will take the following subpuzzle as an example:

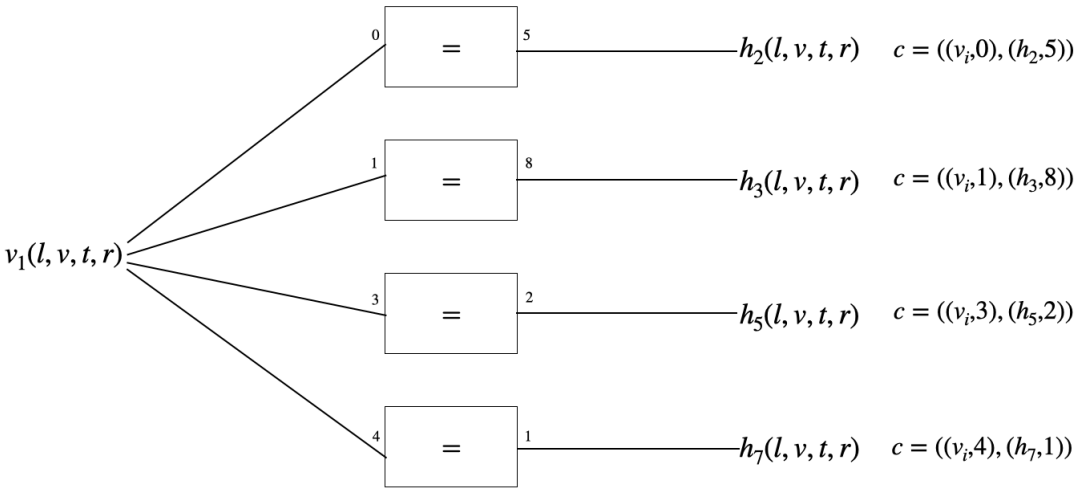


A constraint can be defined as:

$$c = ((w_1, i), (w_2, j))$$

Where each tuple of the constraint represents the word and the character affected by the constraint.

For the example above, the constraint network would look like:



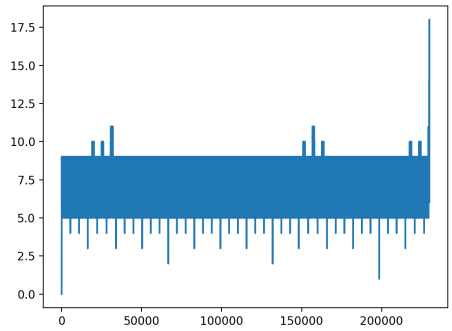
From each constraint we can obtain two arcs which are:

$$arc_1 = \langle w_1, c \rangle$$
$$arc_2 = \langle w_2, c \rangle$$

of which one arc refers to a horizontal word, and the other one to the vertical word.

Domains:
Each variable will initially have the domain of all words of its length. This domain is shared by all the elements of the same length, so as to now word is repeated, and changes throughout the computation.

A. Backtrack
Backtrack has been used as an initial step in the resolution of this problem. As we saw with the sudoku section, the visualisation can be made by plotting the number of variables assigned so far at each iteration. For the 4 puzzles we got the following results:

Puzzle1: time elapsed 3.035552978515625

Puzzle2: No results (Computation time exceeded)
Puzzle3: No results (Computation time exceeded)
Puzzle4: No results (Computation time exceeded)

B. Backtrack with Heuristics

Observing the previous examples, it is easy to notice (...). In order to improve the performance of out implementation, we can include two heuristics to the variables.

Order the variables in decreasing length:

Following the "Choose the most constraining variable first" principle, we can start assigning values to the longest words in the grid. This allows for two advantages:

- The distribution of lengths in most of the fill-in puzzles shows that the number of longer words is relatively small, which is translated into smaller domains for the associated variables.
- Longer words mean higher number of constraints set per assignment. This shows to detect inconsistencies at early phases of the computation and establish the higher number of consistent constraints as possible.

Intercalate horizontal and vertical words:

When we defined the constraints of the problem, we stated that each constraint establishes a relationship of equality between the character of two variables or words:

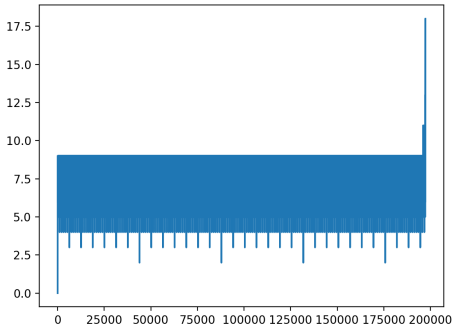
$$c = ((w_1, i), (w_2, j))$$

and those two words were of different type: one is horizontal and another is vertical. For this reason, when a variable is assigned a value, maximum l constraints are set for at most l variables of the opposite type. The words of the same type will not have been constrained and their domain is not reduced. The first inconsistency will occur with the first different-type word that is assigned a value, and it will take in the worst-case scenario to check every combination among the values of the domains of the same-type variables to find a feasible solution.

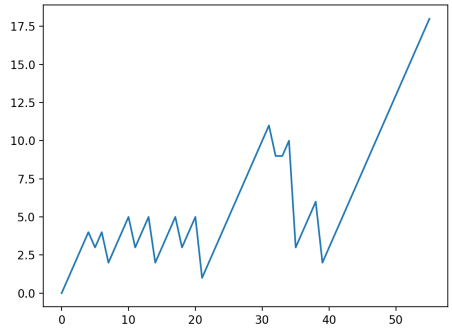
That way, if we intercalate the variables depending on their type, we will get a higher amount of failing rates at early iterations, but almost 0 at later ones.

Combining these two heuristics, we increase the number of effective constraints set at each iteration, which allows us to reduce the domains to a higher extent.

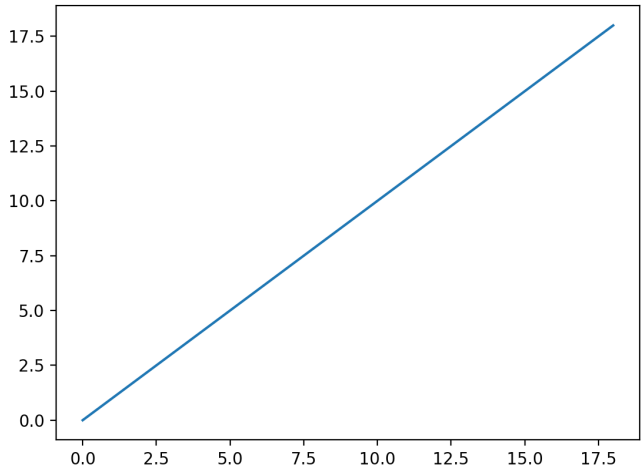
Puzzle1: Only length Ordering
elapsed 1.9399080276489258



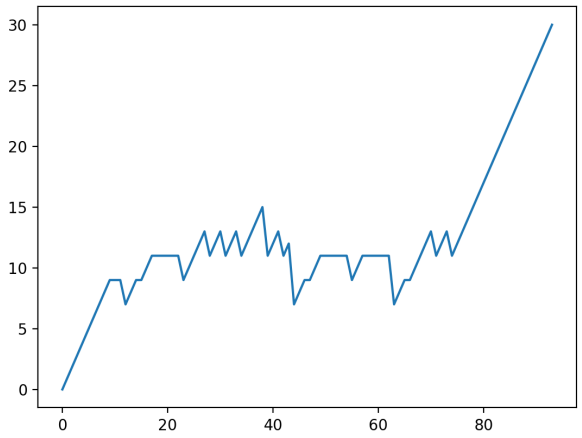
Puzzle1: Only variable intercalation
elapsed 0.0013322830200195312



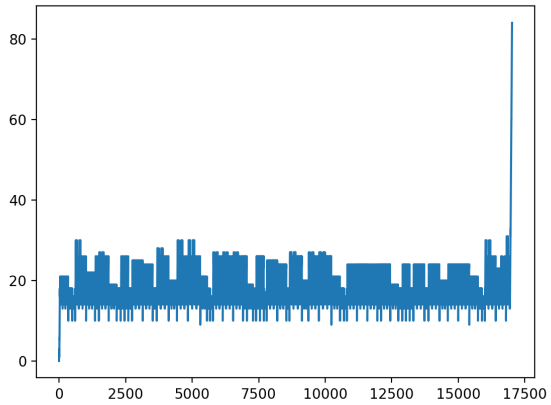
Puzzle1: Length ordering and variable intercalation
elapsed 0.00022220611572265625



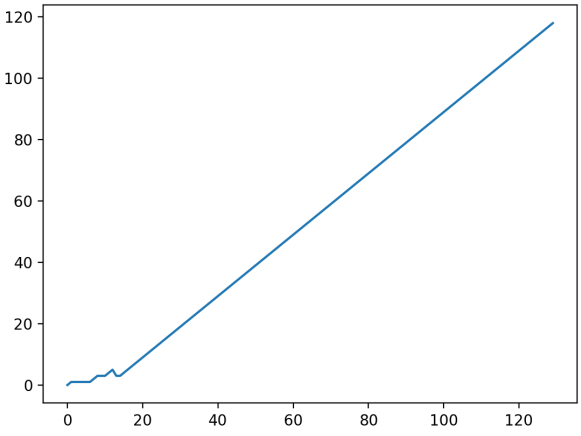
Puzzle2: Length ordering and variable intercalation
elapsed 0.015026092529296875



Puzzle3: Length ordering and variable intercalation
elapsed 1.3710551261901855



Puzzle4: Length ordering and variable intercalation
elapsed 0.007173061370849609



We can easily observe how the heuristics affect the efficiency of the algorithm. For puzzles 2, 3, and 4, no solution was found in a feasible time using the raw backtrack algorithm.

However, applying the heuristics reduced the runtime drastically. We can see that the graphs start showing a linear behaviour in later iterations, showing that inconsistencies have been avoided with an early analysis of the domains.

C. Arc Consistency Algorithm AC-3

One last approach to this problem is yet to be considered. Maintaining our backtrack algorithm, we will reduce the domain of the initial variables in order for it to be arc consistent.

This algorithm would be applied to the csp before the backtrack starts, and would follow the schema:

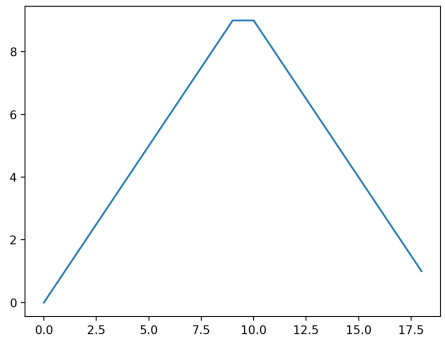
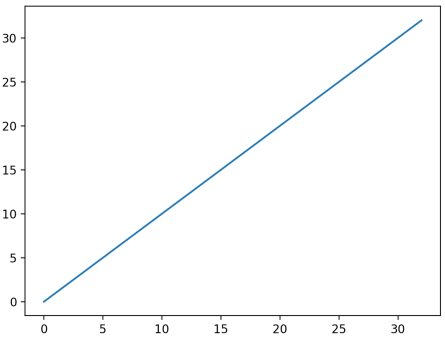
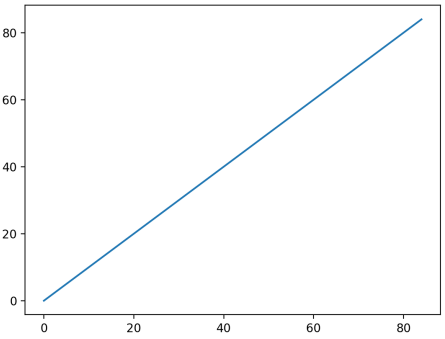
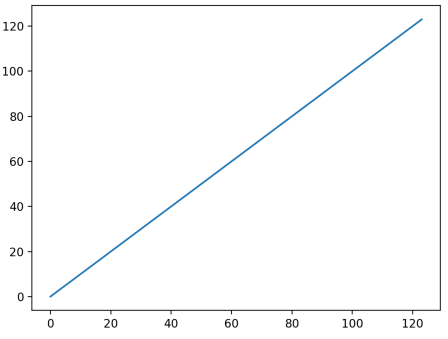
```
1: procedure GAC(V, dom, C)
2:     Inputs
3:         V: a set of variables
4:         dom: a function such that dom(X) is the domain of variable X
5:         C: set of constraints to be satisfied
6:     Output
7:         arc-consistent domains for each variable
8:     Local
9:         D_X is a set of values for each variable X
10:        TDA is a set of arcs
11:    for each variable X do
12:        D_X ← dom(X)
13:        TDA ← {⟨X, c⟩ | c ∈ C and X ∈ scope(c)}
14:    while TDA ≠ {} do
15:        select ⟨X, c⟩ ∈ TDA;
16:        TDA ← TDA \ {⟨X, c⟩};
17:        ND_X ← {x | x ∈ D_X and some {X = x, Y_1 = y_1, ..., Y_k = y_k} ∈ c
   where y_i ∈ D_{Y_i} for all i}
18:        if ND_X ≠ D_X then
19:            TDA ← TDA ∪ {⟨Z, c'⟩ | X ∈ scope(c'), c' is not c, Z ∈ scope(c') \
   {X}}
20:            D_X ← ND_X
21:    return {D_X | X is a variable}
```

Generalized Arc Consistency Algorithm.[1]

The built-in module queue has been used to create the arcs queue. In order to look all the arcs of the problem the function `searchArcs(TDA)` is used, which enqueues into TDA the arcs derived from the constraints of the problem.

The algorithm then dequeues each arc, checks the referenced variable and iterates over its domain and uses `checkConsistency(value, arc)` to create a new domain consistent with the constraint network. If the domain is changed, then the arcs related to the modified variable are enqueued into TDA.

[1] Artificial Intelligence: Foundations of Computational agents

Puzzle 1: AC-3 algorithm Arc consistency time: 0.0017828941345214844 Backtrack time: 0.014760255813598633 Total time: 0.016543149948120117	
Puzzle 2: AC-3 algorithm Arc consistency time: 0.015563011169433594 Backtrack time: 0.042190074920654 3 Total time: 0.05775308609008789	
Puzzle 3: AC-3 algorithm Arc consistency time: 0.035494327545166016 Backtrack time: 0.5327229499816895 Total time: 0.5682172775268555	
Puzzle 4: AC-3 algorithm Arc consistency time: 0.0757749080657959 Backtrack time: 3.173582077026367 Total time: 3.249356985092163	

For the puzzles 2 to 4, the performance is similar. We can see that AC-3 perform a reduction of the domains that transforms the backtrack problem into linear-like. Although the results are quite similar that the ones obtained with the backtrack using the heuristics, we must notice that Puzzle number 3 experiments a reduction in time of about 60%

However, looking at the Puzzle1, we check that the algorithm does not retrieve any valid solution, although it is known that it does exist. We can check the pruned domains for this puzzle after running the AC-3 implementation and we get:

```
{0: ['DAIS', 'EDIT'], 1: ['ARTICLE', 'VESICLE'], 2: ['GI', 'ON'],
3: ['DOLE', 'SILO'], 4: ['DAG', 'EVO'], 5: ['ARID', ‘DENS'],
6: ['OED', 'REF'], 7: ['CLEF', 'CLOD'], 8: ['IO', ‘OR'],
9: ['DAG', 'EVO'], 10: ['ARID', 'DENS'], 11: ['DAIS', ‘EDIT'],
12: ['IO', 'OR'], 13: ['ARTICLE', 'VESICLE'], 14: ['GI', ‘ON'],
15: ['CLEF', 'CLOD'], 16: ['DOLE', 'SILO'], 17: ['OED', 'REF']}
```

All domains have are multi-valued, so we have not encountered any inconsistency in the problem. The reason why we have not found any solution might rely on the implementation of the backtrack algorithm, though it is actually working on the other cases.

Solutions to the tested problems:

A. Sudoku

Sudoku 14

```
7 2 5 | 4 6 8 | 1 9 3
9 3 8 | 7 2 1 | 6 4 5
6 4 1 | 5 9 3 | 2 7 8
---------------------
5 6 2 | 3 7 9 | 8 1 4
1 7 4 | 2 8 6 | 3 5 9
8 9 3 | 1 4 5 | 7 6 2
---------------------
3 8 7 | 9 1 4 | 5 2 6
2 5 9 | 6 3 7 | 4 8 1
4 1 6 | 8 5 2 | 9 3 7
```

Sudoku 24

```
4 3 8 | 2 5 6 | 1 9 7
6 1 2 | 9 4 7 | 8 5 3
7 5 9 | 3 1 8 | 6 2 4
---------------------
2 4 7 | 1 6 9 | 3 8 5
1 8 6 | 4 3 5 | 2 7 9
3 9 5 | 8 7 2 | 4 6 1
---------------------
9 2 1 | 7 8 4 | 5 3 6
5 7 4 | 6 2 3 | 9 1 8
8 6 3 | 5 9 1 | 7 4 2
```

Sudoku 36

```
7 9 3 | 5 8 1 | 2 6 4
6 2 4 | 9 3 7 | 5 8 1
8 1 5 | 4 6 2 | 7 9 3
---------------------
9 8 7 | 3 2 4 | 6 1 5
4 3 6 | 7 1 5 | 9 2 8
1 5 2 | 8 9 6 | 4 3 7
---------------------
5 6 9 | 1 4 8 | 3 7 2
3 4 1 | 2 7 9 | 8 5 6
2 7 8 | 6 5 3 | 1 4 9
```

Sudoku 42

```
8 7 5 | 1 3 2 | 4 6 9
4 6 2 | 9 7 5 | 8 1 3
9 1 3 | 6 4 8 | 5 2 7
---------------------
6 3 1 | 2 9 4 | 7 8 5
5 4 8 | 3 6 7 | 2 9 1
7 2 9 | 8 5 1 | 3 4 6
---------------------
1 5 6 | 4 2 3 | 9 7 8
3 8 4 | 7 1 9 | 6 5 2
2 9 7 | 5 8 6 | 1 3 4
```

Sudoku 45

```
No solution
```

A. Fill-in

Puzzle 1

```
['#', '#', 'E', 'V', 'O', '#', '#']
['#', '#', 'D', 'E', 'N', 'S', '#']
['D', 'A', 'I', 'S', '#', 'I', 'O']
['A', 'R', 'T', 'I', 'C', 'L', 'E']
['G', 'I', '#', 'C', 'L', 'O', 'D']
['#', 'D', 'O', 'L', 'E', '#', '#']
['#', '#', 'R', 'E', 'F', '#', '#']
```

Puzzle 2

```
['Z', 'K', 'O', 'Q', 'S', '#', 'Z', 'G', 'N']
['Q', 'E', 'D', 'T', '#', 'Z', 'O', 'S', 'T']
['O', 'G', '#', '#', '#', 'W', 'G', 'K', 'T']
['#', '#', 'Z', 'G', 'L', 'L', 'T', 'R', '#']
['Q', 'Z', 'Z', 'T', 'F', 'Z', 'O', 'C', 'T']
['#', 'D', 'O', 'F', 'O', 'F', 'U', '#', '#']
['L', 'T', 'T', 'F', '#', '#', '#', 'G', 'A']
['R', 'G', 'U', 'L', '#', 'H', 'O', 'T', 'L']
['Z', 'G', 'H', '#', 'L', 'V', 'O', 'F', 'U']
```

Puzzle 3

```
['V', 'E', 'R', 'N', 'E', '#', 'E', 'R', 'N', 'E', '#', 'F', 'I', 'G', 'S']
['O', 'R', 'I', 'O', 'N', '#', 'R', 'O', 'A', 'D', '#', 'A', 'L', 'E', 'C']
['L', 'I', 'P', 'I', 'D', '#', 'S', 'O', 'N', 'G', '#', 'R', 'E', 'N', 'O']
['T', 'E', 'E', 'S', '#', 'P', 'E', 'T', '#', 'I', 'N', 'T', 'U', 'I', 'T']
['#', '#', '#', 'E', 'R', 'A', 'S', '#', 'O', 'N', 'E', '#', 'M', 'E', 'S']
['T', 'E', 'U', 'T', 'O', 'N', '#', 'I', 'R', 'E', 'S', '#', '#', '#', '#']
['A', 'U', 'N', 'T', 'Y', '#', 'A', 'V', 'A', 'S', 'T', '#', 'B', 'O', 'X']
['B', 'R', 'I', 'E', '#', 'K', 'N', 'O', 'T', 'S', '#', 'C', 'O', 'I', 'R']
['S', 'O', 'T', '#', 'S', 'N', 'O', 'R', 'E', '#', 'C', 'R', 'E', 'N', 'A']
['#', '#', '#', '#', 'T', 'I', 'D', 'Y', '#', 'Q', 'U', 'I', 'R', 'K', 'Y']
['D', 'O', 'G', '#', 'A', 'T', 'E', '#', 'V', 'E', 'R', 'T', '#', '#', '#']
['E', 'X', 'H', 'O', 'R', 'T', '#', 'K', 'I', 'D', '#', 'I', 'N', 'S', 'T']
['L', 'I', 'A', 'R', '#', 'I', 'R', 'A', 'N', '#', 'S', 'C', 'O', 'N', 'E']
['E', 'D', 'N', 'A', '#', 'N', 'I', 'L', 'E', '#', 'O', 'A', 'S', 'I', 'S']
['S', 'E', 'A', 'L', '#', 'G', 'O', 'E', 'S', '#', 'S', 'L', 'E', 'P', 'T']
```

Puzzle 4

```
['B', 'H', 'N', 'N', 'C', 'N', '#', 'P', 'P', 'A', 'G', '#', 'A', 'B', 'D', 'O', 'H', 'N', 'J', 'O', 'A', 'M', 'D', '#', 'A', 'B', 'C', 'I', '#', 'L', 'J', 'I']
['H', 'M', 'L', 'K', 'L', 'M', 'L', 'D', 'I', 'K', 'E', '#', 'P', 'O', 'M', 'H', 'M', 'K', 'J', 'C', 'O', 'F', 'E', 'G', 'C', 'A', 'A', 'A', '#', 'F', 'J', 'L']
['F', 'B', 'J', 'G', 'D', 'C', 'H', 'L', 'O', 'F', 'L', 'G', 'O', 'H', 'P', 'E', 'N', 'D', '#', 'F', 'B', 'D', 'A', 'D', 'H', 'O', 'D', 'P', 'L', 'H', 'I', 'B']
['E', 'L', '#', 'M', 'I', 'L', 'P', 'N', 'B', 'O', 'L', 'O', 'N', 'F', 'F', 'M', 'D', 'I', 'O', 'G', 'C', 'P', 'K', '#', 'O', 'B', 'D', 'C', 'J', 'A', 'I', 'E']
['H', 'O', 'P', 'H', 'P', 'I', 'F', '#', 'A', 'P', 'L', 'O', 'F', 'D', 'P', 'D', 'I', 'E', 'E', 'J', 'E', 'L', 'K', 'B', '#', 'E', 'I', 'H', 'C', 'L', 'N', 'O']
['L', 'I', 'A', 'H', 'L', 'J', 'M', 'K', 'N', 'N', 'N', 'N', 'C', 'J', '#', 'P', 'F', 'G', 'M', 'I', 'H', 'N', 'G', 'A', 'A', 'A', 'D', 'L', 'B', 'E', 'F', 'N']
['J', 'G', '#', 'H', 'K', 'B', 'G', 'D', 'D', 'B', 'M', 'O', 'C', 'I', 'A', 'L', 'O', 'M', 'A', '#', 'G', 'B', 'K', 'I', 'D', 'P', 'M', 'N', 'C', 'J', 'K', 'D']
['K', 'N', 'M', 'J', 'B', 'I', 'J', 'A', 'M', '#', 'I', 'N', 'G', 'B', 'N', 'O', 'D', 'N', 'J', 'P', 'I', 'C', 'N', 'M', 'I', 'L', '#', 'G', 'G', 'M', 'M', 'I']
['M', 'K', 'B', 'C', 'C', 'N', '#', 'F', 'L', 'L', 'M', 'D', 'H', 'M', '#', 'D', 'P', 'C', 'J', 'E', 'N', 'M', 'D', 'B', 'M', 'H', 'M', 'B', 'N', 'F', '#', '#']
['F', 'L', 'A', 'G', 'O', 'N', 'P', 'N', 'H', 'L', 'K', 'L', 'B', 'P', 'B', '#', 'K', 'L', 'E', 'I', 'F', '#', 'M', 'E', 'E', 'D', 'N', 'O', 'H', 'E', 'L', 'O']
['N', 'P', 'H', 'A', 'P', '#', 'P', 'C', 'N', 'D', 'J', 'D', 'L', 'L', 'N', 'E', 'O', 'P', 'P', 'L', 'C', 'E', 'D', 'L', 'L', 'C', '#', 'E', 'L', 'E', 'P', 'I']
['D', 'P', 'E', 'G', 'B', 'P', 'K', 'E', 'F', 'E', '#', 'M', 'L', 'J', 'F', 'L', 'L', 'O', 'N', 'L', 'L', 'C', 'O', 'N', '#', 'G', 'E', 'P', 'E', 'N', 'O', 'K']
['#', '#', 'E', 'P', 'D', 'B', 'P', 'A', 'P', 'K', 'B', 'F', 'F', 'I', 'H', 'F', 'P', '#', 'B', 'G', 'M', 'D', 'F', 'D', 'A', 'F', 'E', 'L', 'M', 'M', 'K', 'B']
['E', 'L', 'M', 'I', 'D', '#', 'J', 'F', 'O', 'J', 'M', 'L', 'I', 'L', 'D', 'M', 'M', 'M', 'L', 'H', 'F', 'O', '#', 'O', 'H', 'N', 'G', 'O', 'G', 'L', 'O', 'E']
['M', 'A', 'A', 'M', 'G', 'P', 'O', 'D', 'B', 'K', 'K', 'L', '#', 'C', 'C', 'K', 'D', 'J', 'E', 'N', 'M', 'E', 'A', 'L', 'E', 'G', 'G', 'E', 'N', '#', 'E', 'G']
['N', 'N', 'H', 'B', 'K', 'K', 'M', 'H', 'F', 'A', 'G', 'K', 'I', 'H', 'J', 'H', 'P', '#', 'D', 'P', 'M', 'H', 'A', 'M', 'G', 'I', 'P', 'K', 'A', 'F', 'F', 'L']
['A', 'N', 'A', 'D', 'M', 'O', 'F', 'N', 'C', 'G', 'C', 'M', 'H', '#', 'F', 'H', 'B', 'G', 'I', 'P', 'C', 'F', 'L', 'B', '#', 'B', 'B', 'A', 'C', 'K', 'A', 'F']
['F', 'C', 'O', 'M', 'E', 'B', 'C', 'I', '#', 'B', 'N', 'K', 'A', '#', 'C', 'I', 'M', 'O', 'P', 'L', 'B', 'A', 'M', 'M', 'H', 'F', 'C', 'N', 'K', '#', 'B', 'L']
['H', 'O', 'L', '#', 'P', 'N', 'O', 'L', 'O', 'C', 'H', 'K', 'M', 'D', '#', 'D', 'O', 'A', 'H', 'D', 'I', 'F', 'D', 'F', 'I', 'N', 'C', 'C', 'I', 'F', 'O', 'A']
['C', 'C', 'H', '#', 'N', 'L', 'G', 'K', 'E', 'H', 'C', 'E', 'K', 'C', '#', 'M', 'M', 'L', 'H', 'K', 'C', 'P', 'M', 'D', 'J', 'H', 'M', 'N', 'K', 'A', 'M', 'M']
```