Declarative Programming in Machine Learning

Software Report

Liviu-Ștefan Neacșu-Miclea

ICA 256/2

1. The CSP Problems addressed

This section briefly introduces the Constraint-Satisfaction Problems studied for this project. A natural language description is provided for each puzzle, followed by one possible way to formally express it by identifying the variables, domains and constraints.

Two problems were initially chosen for this report. An additional third one was also considered later in the work in order to cover one experimental behavior which was not seen in the original suite of two problems.

1.1. Picross

Picross (nonogram) is a mathematic puzzle with Japanese origins that consists in filling a NxM grid with either "on" or "off" cells based on instructions provided for each line and column in the form of a list of numbers, a number k bearing the meaning that there are exactly k adjacent "on" squares on that row/column (Fig.1.). If there are multiple numbers, the order matters, for example if a row is annotated with the list of numbers k1, k2, it means that the sequence of k1 adjacent "on" cells is placed before the other sequence of k2 cells, with at least one empty space between them [1].

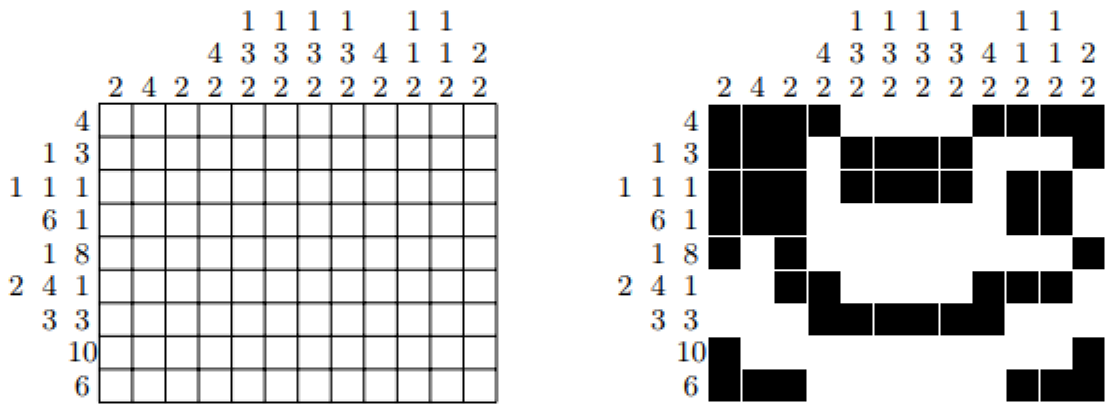


Fig.1. An example of a Picross puzzle and its solution [1]

Formally, the state of each cell can be considered a variable with values in a two-value domain, and the constraints are the rules associated to each row and column. We choose a numbers domain

(1-“on” cell, 0-“off” cell) due to the simplicity of describing the constraints later (we can check if a sequence of adjacent cells has all cells “on” by simply summing all of the cells values).

Let N and M be the number of rows, respectively columns of the grid, and $S = \bigcup_{i=0}^{\infty} \mathbb{N}^i$ the set of ordered sequences of natural numbers. We also define $R_i, C_j \in S$, with $i = \overline{1,N}, j = \overline{1,M}$ the lists of numbers associated with row i and column j. Finally, we can define:

- the CSP problem $P = (Z, D, C)$, with
- the set of variables $Z = \{z_{ij} \mid 1 \leq i \leq N, 1 \leq j \leq M\}$
- the domain mapping $D(z) = \{0,1\}\ \forall\ z \in Z$
- the set of constraints C, defined as follows:
    - $\forall\ (s_1, \dots, s_k) \in R_i, c_{R_i} \in C$, where
        - c_{R_i} constraints the variables $z_{i1}, \dots, z_{iM}$
        - to fulfill $\forall p \in \{1, \dots, k\}\ \exists a_p, b_p \in \{1, \dots M\}, a_p + s_p = b_p, \sum_{j=a_p}^{b_p-1} z_{ij} = s_p$, $(p > 1 \wedge a_p > b_{p-1})$,
    - $\forall\ (s_1, \dots, s_k) \in C_j, c_{C_j} \in C$, where
        - c_{C_j} constraints the variables $z_{1j}, \dots, z_{Nj}$
        - to fulfill $\forall p \in \{1, \dots, k\}\ \exists a_p, b_p \in \{1, \dots N\}, a_p + s_p = b_p, \sum_{i=a_p}^{b_p-1} z_{ij} = s_p$, $(p > 1 \wedge a_p > b_{p-1})$,
    - no other constraints belong to C.

1.2. SEND+MORE=MONEY [2]

This logic arithmetic puzzle requires replacing each of the letters S, E, N, D, M, O, R, Y with a different digit such that the equality SEND+MORE=MONEY holds, and the numbers are not starting with trailing zeros. [2]

The specificity behind this puzzle is the need to introduce additional variables that handle dependencies involving propagating the carry through the addition process (Fig.2.).

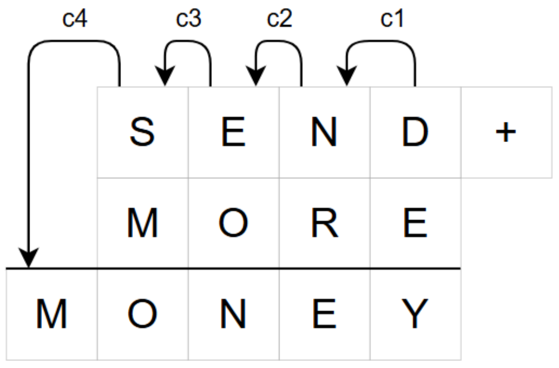


Fig.2. Visual depiction of the arithmetics problem and carries involved

Formally, the problem is described by:

- the set of variables $Z = \{S, E, N, D, M, O, R, Y, c_1, c_2, c_3, c_4\}$
- the domain mapping
    - $D(z) = \{1, \dots, 9\}, z \in \{S, M\}$
    - $D(z) = \{0, \dots, 9\}, z \in \{E, N, D, O, R, Y\}$
    - $D(z) = \{0,1\}, z \in \{c_1, \dots, c_4\}$
- the set of constraints $C = \{C_1, \dots, C_6\}$
    - $C_1: D + E = Y + 10 \cdot c_1$
    - $C_2: N + R + c1 = E + 10 \cdot c_2$
    - $C_3: E + O + c_2 = N + 10 \cdot c_3$
    - $C_4: S + M + c_3 = O + 10 \cdot c_4$
    - $C_5: M = c4$
    - $C_6: \forall x, y \in \{S, E, N, D, M, O, R, Y\}, x \neq y$

1.3. Dinosaurs mystery [3]

We will annotate the sentences of this problem while reproducing it for further reference:

S1: Skeletons of Eucentrosaurus, Hadrosaurus, Herrerasaurus, Megasaurus, and Nuoerosaurus have been found in Argentina, Canada, China, England, and the U.S.
S2: None of these dinosaurs has been found in more than one of these countries.
S3: The dinosaur from China is the largest of the bunch.
S4: Herrerasaurus was the smallest.
S5: Megasaurus was larger than Hadrosaurus or Eucentrosaurus.
S6: The dinosaurs from China and North America were planteaters.
S7: Megasaurus ate meat.
S8: Eucentrosaurus lived in North America.
S9: The dinosaurs from North America and England lived later than Herrerasaurus.
S10: Hadrosaurus lived in Argentina or the U.S. [3]

The features that matter for the choice variables and domains are dinosaurs, countries and sizes, while the temporal and diet-related information can be used later while refining the constraints.

Due to the fact that in multiple places the dinosaurs are referenced by the country where their fossils were found (S3, S6, S9, S10), it would be intuitive to consider dinosaurs as variable and countries as assigned values. In additions, the sizes are referenced by comparison (S4 – "the smallest", S5 – "larger than"), so a possible way of modeling that is assigning each dinosaur size a rank form 1 (smallest) to 5 (largest).

In essence, the formal definition of the problem is:

- The set of variables $Z = \{c_{Euc}, c_{Had}, c_{Her}, c_{Meg}, c_{Nuo}, s_{Euc}, s_{Had}, s_{Her}, s_{Meg}, s_{Nuo}\}$
- The domain mappings D, such as
    - $D(c) = D_c = \{Argentina, Canada, China, England, US\}, \forall c \in Z_c = \{c_{Euc}, \dots c_{Nuo}\}$
    - $D(s) = D_s = \{1,2,3,4,5\} \forall s \in Z_s = \{s_{Euc}, \dots, s_{Nuo}\}$

- The sets of constraints $C = \{C_1, \dots, C_{10}\}$:
    - $C1: \forall c_1, c_2 \in Z_c, c_1 \neq c_2$
      (S2: no two dinosaurs were found in the same country)
    - $C2: \forall s_1, s_2 \in Z_s, s_1 \neq s_2$
      (assume unique sizes)
    - $C3: c_i = China \rightarrow s_i > s_j, \forall j \neq i$
      (S3: China dinosaur is the largest)
    - $C4: s_{Her} = 1$
      (S4: Herrerasaurus was the smallest)
    - $C5: s_{Meg} > s_{Had}$
    - $C6: s_{Meg} > s_{Euc}$
      (S5 splitted in 2 binary constraints)
    - $C7: c_{Meg} \notin \{China, Canada, US\}$
      (S6 & S7: Megasaurus ate meat, therefore was not a plant eater, so it can't be from China and North America = CA+US)
    - $C8: c_{Euc} \in \{Canada, US\}$
      (S8)
    - $C9: c_{Her} \notin \{Canada, US, England\}$
      (S9 "The dinosaurs from North America and England lived later than Herrerasaurus" is not about time at all, it actually suggests that Herrerasaurus didn't live in NA or En)
    - $C10: c_{Had} \in \{Argentina, US\}$
      (S10)

2. CSP Problem Solving Framework

2.1. Modeling the problem

The CSP problem modeling feature closely follows the formal definition for seamless transition between specifications. Its components are (Fig.4.):

- **Value** that can be possibly taken by a variable;
- **Domain** defined as a finite enumeration of Values;
- **Variable** identified by a name, taking values in a Domain;
- **Label**, an association between a Variable and a Value;
- **CompoundLabel** consists of a collection of zero, one or more Labels;
- **Constraint** involving a set of Variables and defined as a logical relation that must be satisfied by their Values. Due to memory concerns, rather than enumerating the compound labels that satisfy the constraints, a logical evaluation approach was preferred instead: a constraint provides a way to dynamically tell whether the values of the variables involved satisfy it or not;
- **Problem** defined as a list of Variables and the Constraints that link them.

The output of the modeling process is the concept of a Problem, which will further be used by the reducing and search algorithms.

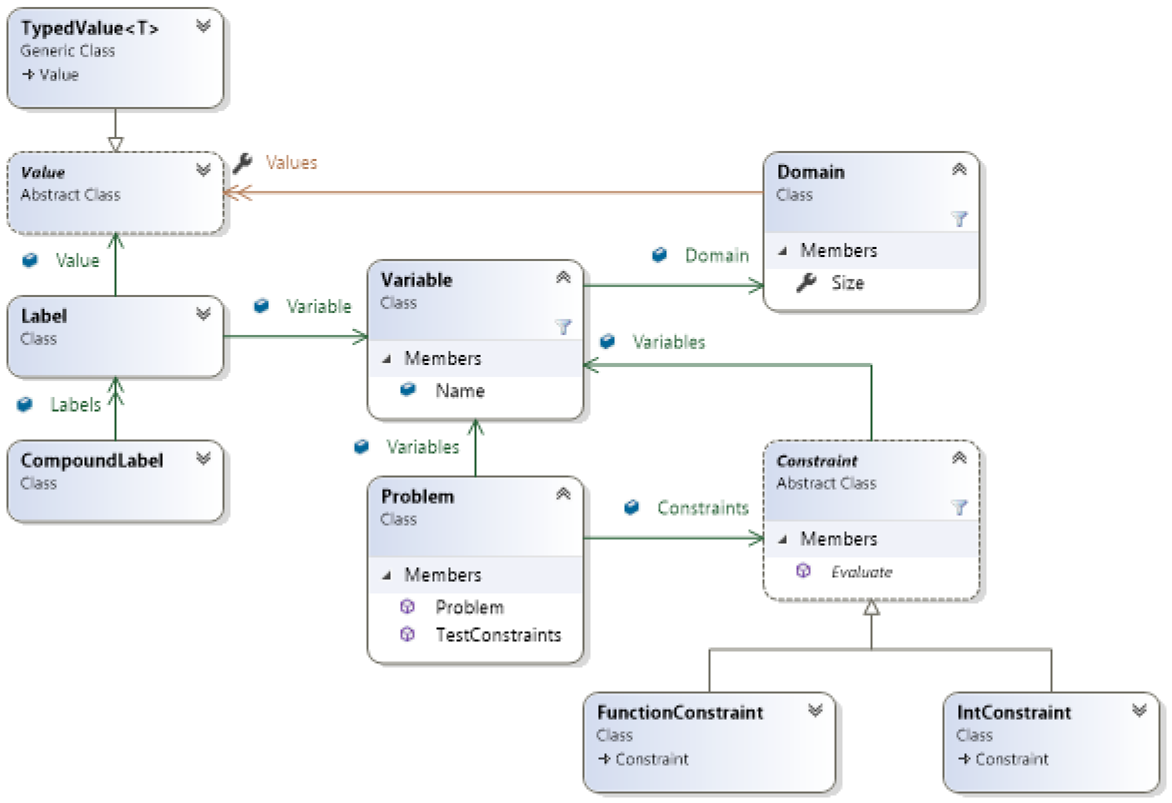


Fig.4. Class diagram showing relations of the entities involved in problem modeling

2.2. Reducing algorithms

Inside the system, a reducing algorithm is viewed as a method that takes in a Problem and outputs an equivalent Problem, obtained after applying any problem reduction techniques. Three reducing algorithms were implemented in this project (Fig.6.).

2.2.1. Node Consistency (NC)

Node-consistency enforces every value in a variable's domain to satisfy all unary constraints involving that variable. Values that violate unary constraints are removed. It is intuitive to apply node consistency before other reduction methods (such as arc-consistency) to early get rid of values leading to inconsistency [5].

The NC algorithm follows these steps:

- For each variable, examine only the unary constraints that involve it;
- For every value the domain of that variable, check if it satisfies all of its unary constraints;
- If at least one of the unary constraints are violated, remove that value from the domain;
- Rebuild the CSP problem with the newly pruned domains.

2.2.2. Arc Consistency (AC-3)

Arc-consistency (AC) ensures that for every binary constraint involving variables X and Y, for every value in the domain of X, there is at least one value of Y that satisfies the constraint. Any values of X that cannot be paired with a value of Y without violating the constraint is removed from its domain. The variant of the arc consistency algorithm used in this project is is AC-3, which keeps checking the variables involved in binary constraints until no more pruning is possible [5].

The concrete idea of the algorithm is:

- Find all the binary constraints. If none exist, then return the problem unchanged;
- For every binary constraint CXY, put the arcs $X \overset{C}{\rightarrow} Y$ and $Y \overset{C}{\rightarrow} X$ inside a queue;
- Repeatedly dequeue an arc $X_i \overset{C}{\rightarrow} X_j$ and try to prune the domain of X_i;
- For each value of X_i, remove it if not value in the domain of X_j satisfied the constraint C
- If a domains becomes empty after pruning, then the problem is inconsistent and the algorithm finishes.
- If the domain of X_i was altered by pruning, enqueue all arcs $X_k \rightarrow X_i$ bounded by a constraints, $X_k \neq X_j$
- Run until the queue is consumed, and return the new problem with the reduced domains, keeping the original constraints.

2.2.3. Generalized Arc Consistency (GAC)

Generalized Arc Consistency is an extension of binary AC to constraints between any number of variables. The GAC enforces any value from the domain of a variable X involved in a constraint C to be part of at least one complete set of labels with all variables from C that satisfies the constraint. The process is repeated until values cannot be further removed from any domains [5].

The algorithm works as follows:

- Keep a support table that remembers whether a value of a variable already has a known support for a given constraint;
- Repeatedly loop over all constraints, and for each variable in each constraint, check each value in its domain;
- For each value, try to find a supported tuple, meaning a combination of values for the other variables in the constraint that satisfy it;
- Keep iterating until a full loop over the constraints didn't make any changes;
- Rebuild the CSP with the new pruned variable domains.

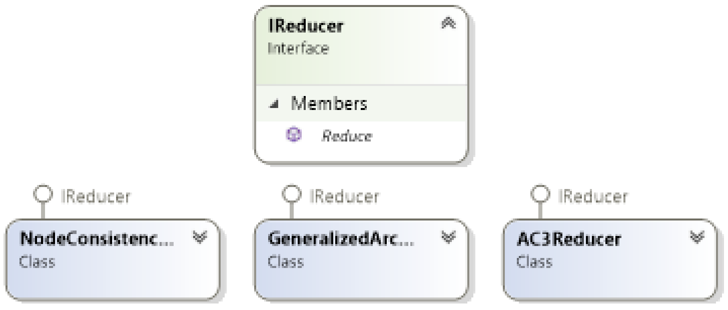


Fig.6. The interface and classes related to problem reduction

2.3. Search algorithms

Reduction is a preparatory step that aims to produce an easier problem that would be further passed to the actual search-based solvers whose task is to find a complete variable-value assignment pairs that satisfies all the constraints. Different solvers have different search strategies, each suitable for certain types of CSP problems and solution expectations – whether we need all solutions, any solution that fits in the constraints, or a fast solution at the risk of violating some of the constraints.

Aside from the naive backtracking algorithm, two more CSP solvers are implemented, each being a representative of their own class of solvers: Forward Checking as a basic search strategy, and Heuristic Repair as a stochastic search method (Fig.7.).

The current implementation does not ensure completeness, the algorithms stop after finding the first solution, or report failure to find any solution.

2.3.1. Backtracking

Backtracking search performs an exhaustive checking of the complete labels by selecting all possible combination of values for all variables, rejecting the ones that violate any constraints, and selecting the one that satisfies all the constraints [4].

The backtracking implementation creates an enumerator over the domain of each variable, and repeatedly generates a compound label taking the current value of each variable’s enumerator. The algorithm then advances one variable’s enumerator (choosing order is not regulated in any way, it is only important to keep it consistent across the cycle) to produce the next solution. If an enumerator is exhausted, it gets reset while the next variable’s enumerator is incremented, and the process continues.

2.3.2. Forward Checking

The forward checking solver is an optimization over simple backtracking by employing one step lookahead: when assigning a variable, it immediately eliminates future domain values of unassigned variables that would violate constraints. When encountering inconsistency (a future domain becomes empty, thus no solution is possible with the current assignment), early backtracking is performed [4].

The implementation follows the FC-1 algorithm [5]:

- Start with an empty assignment and full domains;
- Select the next unassigned variable (order not altered in any way);
- For each value in the domain of that variable, create a new partial assignment;
- Check if all constraints that only include the assigned variables are satisfied;
- Perform the forward check: prune domain values of future variables that are incompatible with this assignment;
- If no domain becomes empty, continue with the extended assignment and updated domains;
- If all variables are assigned and the constraints are satisfied, then return the solution;
- If all values fail, then backtrack;
- If the algorithm finishes without finding a solution, announce the failure.

2.3.3. Heuristic Repair

As opposed to the previous ones, the Heuristic Repair algorithm starts from a complete, possibly inconsistent assignment and iteratively repairs conflicts. At each step, it selects a variable involved in some violated constraints, reassigns it to a value that minimizes the total number of violations, and repeats the process for a finite number of iterations or until a solution is found [5].

The algorithm follows these steps:

- Randomly initialize a complete assignment of all variables;
- Evaluate conflicts and identify which variables are involved in unsatisfied constraints;
- Randomly pick a variable involved in a conflict;
- Select a new value for that variable that minimizes the number of violated constraints (if there are multiple values hitting this number, choose randomly);
- Update the assignment and repeat;
- If solution was found, return it;
- If no solution was found in a given number of steps, announce failure.

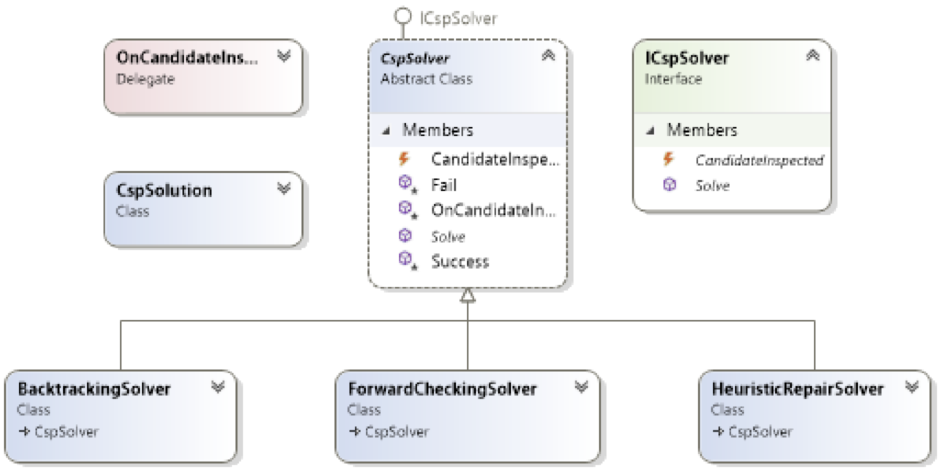


Fig.7. The CSP solver sub-system

2.4. Details on evaluation of CSP solving setups

The complexity of the problem is assessed by computing the size of the constraint graph (number of nodes and edges), where:

- a node corresponds to a variable;
- there is an (unoriented) edge between X and Y if there exists at least one constraint connect X to Y. Constraints of k variable generate a clique between those k variables. Unary constraints are excluded from computation of edges.

Additionally, metrics related to domain size are computed on the problems:

- Average domain size: $\frac{1}{|Z|}\sum_{z\in Z}|D(z)|$ [5];
- Maximum domain size: $\max_{z\in Z}|D(z)|$;
- Search space size: $\prod_{z\in Z}|D(z)|$ [5].

These metrics are used to compare the search effort between problems, or between a problem and one (or more) of its reduced variants.

Multiple reducers setups were applied for each problem and for each solvers. There is at least one instance of a reducer being applied individually (e.g. only NC, AC-3 or GAC), and there are identified per-problem cases where certain combinations of solvers drastically reduce the search space or time. In the choice of which reducers combinations to include in the report, the redundant ones were ruled out (e.g. not apply NC when there aren't any unary constraints, or AC-3 where binary constraints are missing).

Furthermore, the number of inspected candidates is computed on each solver session, and it is used as a way to evaluate and compare the performance of different solvers. This method was preferred to monitoring the execution time, which can be considered a volatile metric, varies between executions, and is machine dependent. Instead, the number of inspected candidates is consistent across executions under the same conditions (except for stochastic methods), and scales well with the magnitude of the expected amount of time required to evaluate that number of candidates.

For the stochastic method, a failure rate was measured: the algorithm is run multiple times, and the number of cases when it exceeds the maximum number of steps without finding a solution is recorded. This provides insight into the flexibility of the problem (multiple solutions allow for more points of convergence, thus lower failure rate) or how trivial it is (was the solution found in the first 10 tries or in 100?).

3. Experiments and results

For each CSP problem mentioned, the constraint graph was computed and displayed visually, along with the metrics described in the previous section.

3.1. Picross

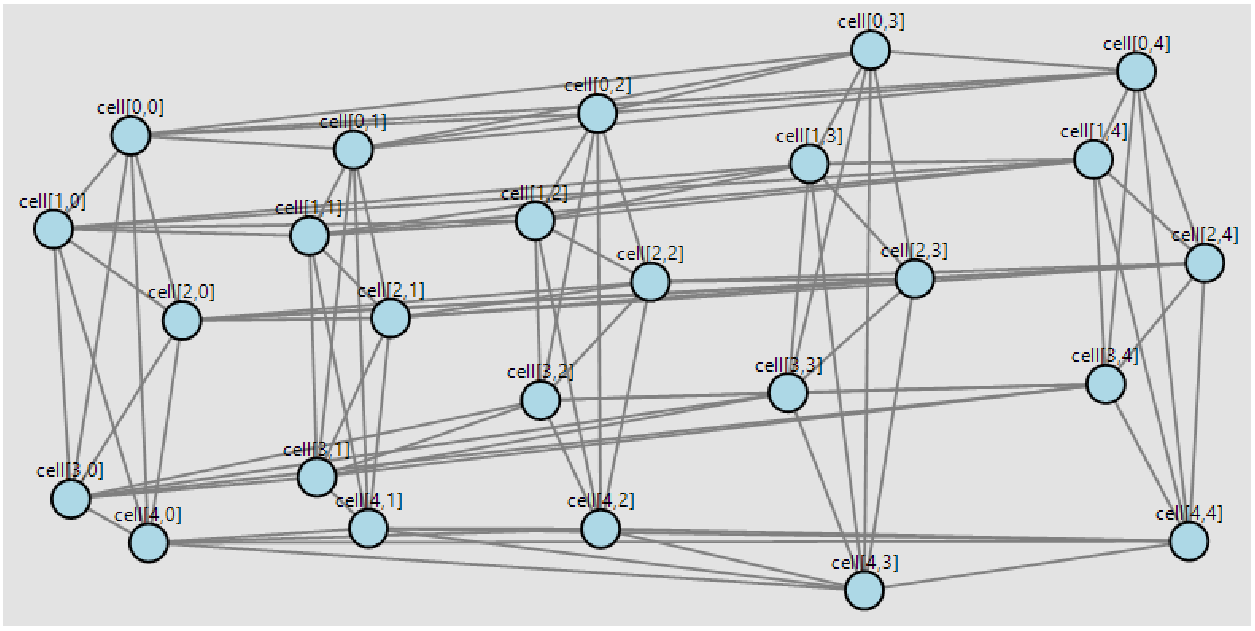


Fig.8. Picross constraint graph

Constraint graph size (Picross 5x5)					
Nodes	25		**Edges**	100	
Algorithms performance					
Reducer	**None**	**NC**	**AC3**	**GAC**	**GAC+GAC**
Domain Size Avg	2	2	2	1.24	1
Domain Size Max	2	2	2	2	1
Search Space	33554432	33554432	33554432	64	1
Backtracking steps	4685803	4685803	4685803	22	1
FC steps	232	232	232	33	25
HR min steps (10 runs)	125	16	67	3	1
HR avg steps	172	117	188	13	1
HR failures (10 runs)	5	5	7	7	0

Table.1. Picross metrics

3.2. SEND+MORE=MONEY

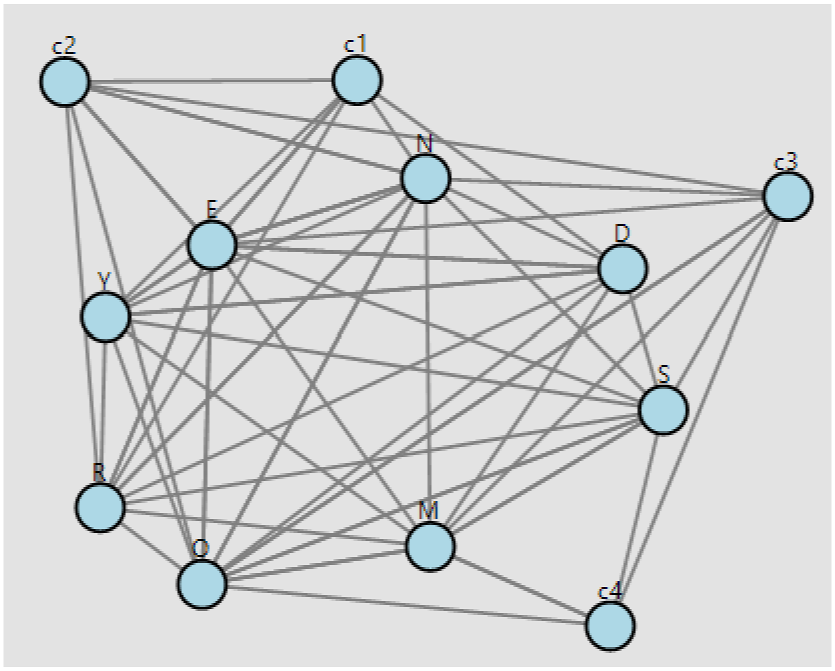


Fig.9. Paleo-arithmetics constraint graph

Constraint graph size (SEND+MORE=MONEY)						
Nodes	12		**Edges**	55		
Algorithms performance						
Reducer	**None**	**NC**	**AC3**	**GAC***	**AC3+ GAC**	**GAC*+ GAC**
Domain Size Avg	7.16	7.16	6.41	5.83	4.25	4.25
Domain Size Max	10	10	10	9	8	8
Search Space	1296000000	1296000000	72000000	34012224	524288	524288
Backtracking steps	913686894	913686894	29526894	13598680	203464	203464
FC steps	1197261	1197261	1128725	672929	12648	12648
HR min steps	1010	-	-	-	11	151
HR avg steps	1010	-	-	-	89	212
HR failures (10 runs)	9	10	10	10	7	8

Table.2. Paleo-arithmetics metrics (*=takes more time to compute)

3.3. Dinosaurs problem

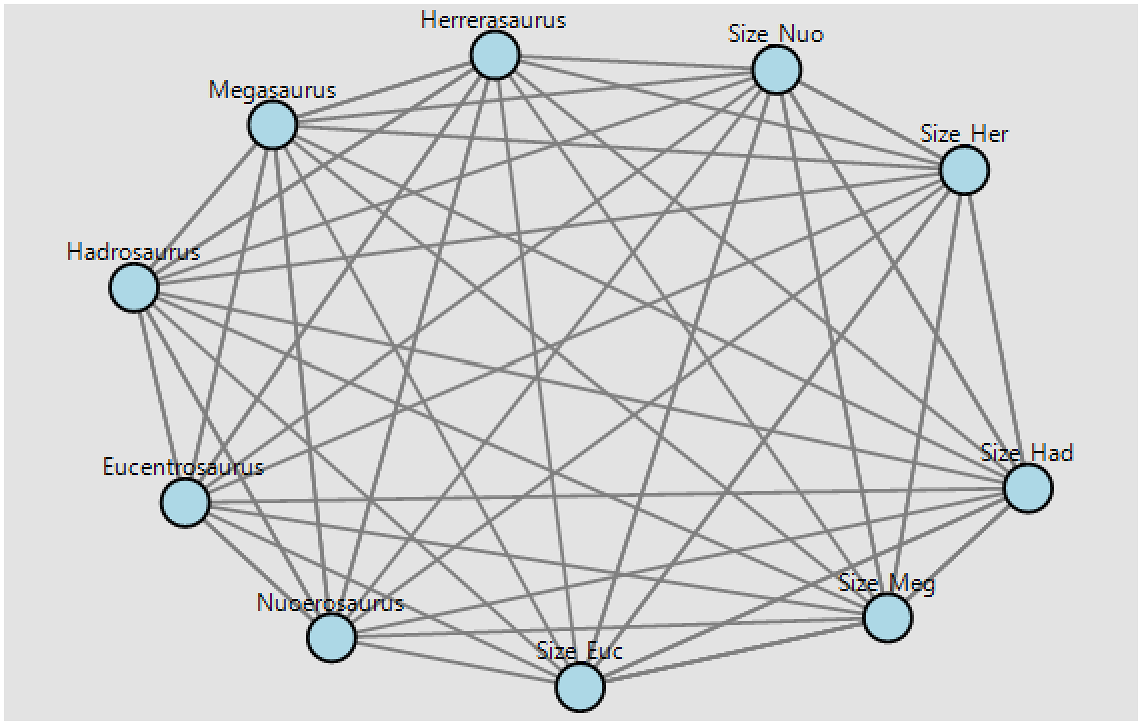


Fig.10. Dinosaurs problem constraint graph

Constraint graph size (Dinosaurs problem)							
Nodes		10		**Edges**		47	
Algorithms performance							
Reducer	**None**	**NC**	**AC3**	**GAC**	**NC+AC3**	**NC+GAC**	**GAC+ GAC+ GAC**
Domain Size Avg	5	3.4	4.7	3.1	3.1	2.2	1.8
Domain Size Max	5	5	5	5	5	4	3
Search Space	9765625	50000	5000000	25600	25600	864	81
Backtracking steps	9007897	46603	4520397	23563	23563	735	65
FC steps	223	120	217	114	114	20	15
HR min steps	16	26	53	38	14	2	2
HR avg steps	62	61	79	44	80	36	27
HR failures (10 runs)	6	6	6	8	7	0	0

Table.3. Dinosaur problem metrics

4. Discussion

The unreduced problems feature huge search spaces, which tend to put the naive backtracking and stochastic solver in difficulty. The impact of not employing any variable ordering strategy (such as the Minimum Remaining Values heuristic) is drastically felt in the backtracking solver finding the solution after analyzing more than 70% percent of the search space (on the arithmetics and dinosaur puzzles).

Obviously, some consistency-based reducing algorithms have no effect on incompatible problems (NC on problems without unary constraints or AC when there are no binary constraints are unable to prune any variable domains). However, some reducers are powerful tools when applied on certain problems. For example, NC was able to eliminate more than 99.5% of the original searching space of the dinosaurs problem. AC3 provides good pruning in some domains (reduces down to 5% (arithmetics) or 51% (dinosaurs)), but it heavily depends on the constraint structure. An interesting example is the all-different constraint between a cluster of variables, which can be defined either as a singular multivariate constraint, or as an enumeration of binary constraints enforcing pairwise difference between any two variables X and Y. In the latter case, supposing, where technically there exist constraint which can be analyzed by an AC algorithm, if $|D(X)|, |D(Y)| > 1$, then no domain pruning is performed, because for every assignment of X there is a guaranteed value of Y different from X which ensures the arc consistency. GAC results in the most powerful pruning among the individual reducers, significantly reducing the search space: 0.0001% (Picross), 2.62% (arithmetics), 0.26% (dinosaurs). However, it comes at the cost of potential increased computation time, due to the fact that support computation of a value ends up in backtracking over the search space of the other variables involved in the constraint. In the case of constraints that involve all the problem's variables (such is the constraint „The largest dinosaur is from China" from the dinosaurs problem which requires all the country variables in order to find out which is the dinosaur from China, and then all the size variables to check if it is indeed the largest), applying GAC is actually equivalent to backtracking the entire search space of the problem!

Combining multiple reducers on the same problem has been observed to produce beneficial effects. It is intuitive that NC should be applied first to preemptively remove inconsistent values from individual domains, then AC follows to further improve consistency over pairs of constrained variables, then GAC might be used as a divide-et-impera approach of ensuring consistency over each constraint first before solving the problem as a whole. Indeed, applying AC-3 before GAC got rid of the large computational time issue observed when using GAC alone on the arithmetics puzzle, reducing the search space size with evens of magnitude more than each of the reducing algorithm individually (0.04% of the original size). On the same problem, AC-3+GAC had the same effect on the search space as GAC+GAC, but with a considerable time advantage. Moreover, applying GAC three times on the dinosaurs problem was able to get the search process down to only 3 candidates. This confirms that application of reducing algorithms is problem-dependent and results may vary considerably, but the principle of optimizing over the constraints in order of their arity proves effective in most practical cases.

The FC-1 solver is considerably more advantageous than the baseline backtracking solely due to its lookahead feature. The number of FC-1 steps can get down to 0.0006% (picross), 0.13% (arithmetics), and 0.0024% (dinosaurs) of the backtracking steps. The heuristic repair algorithm, while capped up to a maximum number of steps by design, is also prone to failure to provide any solution in the designated iterations. Due to its stochastic nature, it merely relies on the flexibility of the problem, described either by the large number of solution points that might increase the chances of one of them becoming the convergence point, or by the nature of the constraints. The selected problems all have unique solutions, and are expected to have high failure rates (out of 10 runs, 7 to 9 or even all 10 ended in an unconsistent solution, with reducers having sometines atenuating the effect – such is in picross and dinosaurs). However, there are instances of practical problems where stochastic algorithms might suit the best, such as the academical schedule with preferences, or video memory allocation in old text/tile background based graphics systems. This is because sometimes a „good enough" solution (Fig.11) reached fast is often preferable to a global optimum which may be hard to compute [5].

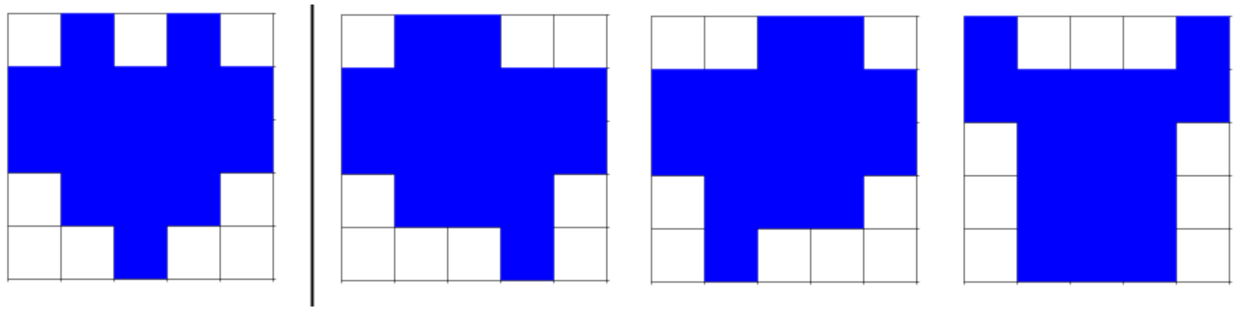


Fig.11. Picross puzzle: the correct solution vs. possible local optimum failure points reached by HR

5. Conclusion and future work

Finally, the experiments prove that reduction based on domain consistency is a great factor in bringing the performance of a CSP problem to a desirable level, with GAC offering the strongest pruning and dramatically reducing the searching costs. Combining reducers makes for a good synergy, often turning huge search spaces into small, manageable sized ones, especially when applied in order of the constraints arity. Heuristic repair can be useful due to its speed if its caveat is backed by flexibility or highly redundant problems, but forward cheking is also fast and crucial when reliability is a concern. Overall, a CSP solver requires care in choosing the right search algorithm and a customized reduction strategy can considerably increase the performance.

Future improvements of this work may include extending the framework to support path-consistency reducing algorithms (which did not fit well in an architecture where constraints were designed as logical predicate validators instead of enumeration of consistent compound labels – a hybrid approach would allow PC reducers to be implemented while keeping the system memory-friendly). Additional care to order of variables would have possibly boosted both the backtracking and other algorithms performance. Finally, more specific evaluation metrics could be employed, like number of supports testing for GAC, memory usage, search-tree related measures like depth and branching factor, number of violated constraints for stochastic methods, or featuring solver completeness in addition to soundness.

6. References

[1] Roucairol, M., & Cazenave, T. (2024, November). Generating Difficult and Fun Nonograms. In *International Conference on Computers and Games* (pp. 119-129). Cham: Springer Nature Switzerland.

[2] Pop, H.F., (2025). 02b problems-1 [Lecture notes]. Declarative programming in Machine Learning, Babeș-Bolyai University of Cluj-Napoca

[3] Pop, H.F., (2025). 02b problems-2 [Lecture notes]. Declarative programming in Machine Learning, Babeș-Bolyai University of Cluj-Napoca

[4] Kondrak, G., & Van Beek, P. (1997). A theoretical evaluation of selected backtracking algorithms. Artificial Intelligence, 89(1-2), 365-387.

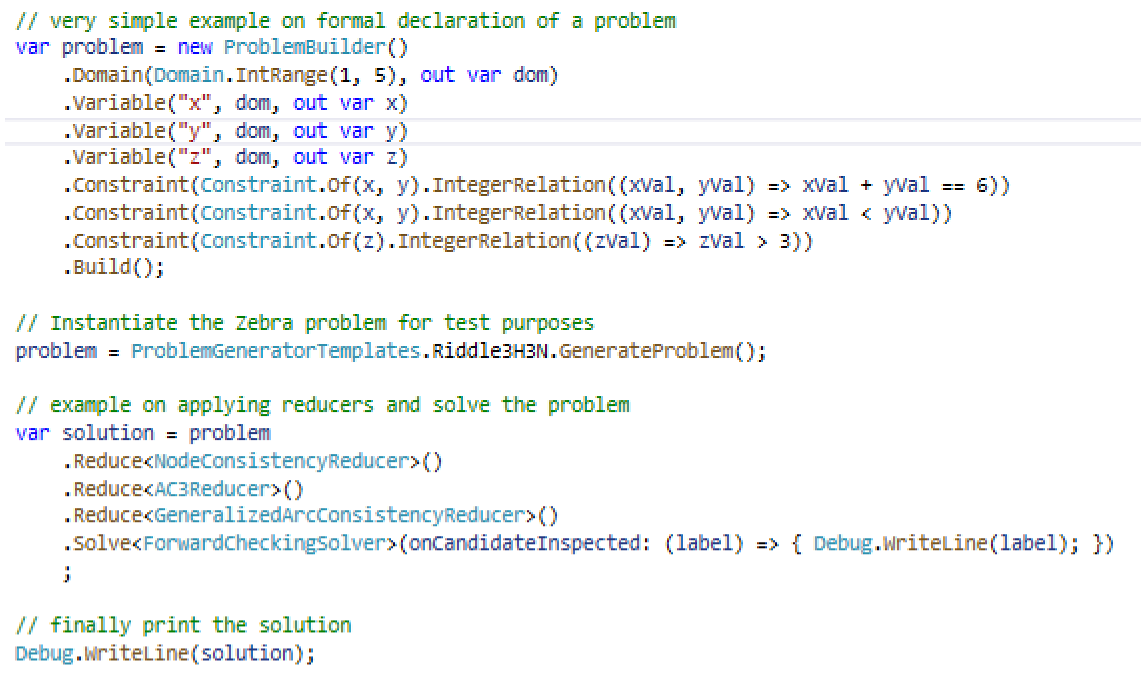
[5] Tsang, E. (1993). Foundations of Constraint Satisfaction. Academic Press.

Acknowledgement:

This work is the result of my own activity, and I confirm I have neither given, nor received unauthorized assistance for this work. I declare that I did not use generative AI or automated tools in the creation of content or drafting of this document.

7. Code snippets

```
public static ProblemGenerator Picross = new ProblemGeneratorBuilder()
    .Name("Picross")
    .Description($"Picross puzzle (or Nonogram): the player has to fill an NxM white grid {NL}" +
        $"with black cells based on a list of number associated to each row/column that describe {NL}" +
        $"the lengths of continuous black stripes separated by white spaces what exist on that row/column.")
    .Viewer<PicrossSolutionViewer>()
    .Param<int>("N", 5)
    .Param<int>("M", 5)
    .Param<int[][]>("rowCues", new[]
    {
        new[] {1,1}, // first row: 2 squares separated by arbitrary space
        new[] {5},   // second row: 5 adjacent squares
        new[] {5},   // third row: 5 adjacent squares
        new[] {3},   // fourth row: 3 adjacent squares
        new[] {1},   // fourth row: 1 square
    })
    .Param<int[][]>("colCues", new[]
    {
        new[] {2},   // first row: 2 adjacent squares
        new[] {4},   // second row: 4 adjacent squares
        new[] {4},   // third row: 4 adjacent squares
        new[] {4},   // fourth row: 4 adjacent squares
        new[] {2},   // fifth row: 2 adjacent squares
    })
    .Build(pms =>
    {
        var b = new ProblemBuilder();

        var N = pms.Get<int>("N");
        var M = pms.Get<int>("M");

        var rowCues = pms.Get<int[][]>("rowCues");
        var colCues = pms.Get<int[][]>("colCues");

        // black/white or on/off grids => binary domain
        b.Domain(Domain.FromValues(new[] { 0,1 }), out var domain);

        // variable for each cell
        var grid = new Variable[N, M];
        for (int i=0; i<N; i++)
        {
            for (int j=0; j<M; j++)
            {
                b.Variable($"cell[{i},{j}]", domain, out grid[i, j]);
            }
        }

        // predicate to check if a row/column of values matches the given clue
        bool MatchesClue(Value[] vals, int[] clue)
        {
            // build the sequence of lengths of adjacent black squares separated by spaces
            var seq = new List<int>();
            int count = 0;
            foreach (var v in vals)
            {
                if (v.EvaluateAs<int>() == 1)
                {
                    count++;
                }
                else
                {
                    if (count > 0)
                    {
                        seq.Add(count);
                        count = 0;
                    }
                }
            }
            if (count > 0)
                seq.Add(count);

            // compare the obtained sequence to the clue
            if (seq.Count != clue.Length)
                return false;

            for (int k = 0; k < clue.Length; k++)
            {
                if (seq[k] != clue[k])
                    return false;
            }

            return true;
        }

        // Add row constraints
        for (int i=0; i<N; i++)
        {
            var rowVars = Enumerable.Range(0, M).Select(j => grid[i, j]).ToArray();
            var clue = rowCues[i];
            b.Constraint(new FunctionConstraint(rowVars, vals => MatchesClue(vals, clue)));
        }

        // Add column constraints
        for (int j=0; j<M; j++)
        {
            var colVars = Enumerable.Range(0, N).Select(i => grid[i, j]).ToArray();
            var clue = colCues[j];
            b.Constraint(new FunctionConstraint(colVars, vals => MatchesClue(vals, clue)));
        }
        return b.Build();
    });



public static ProblemGenerator PaleoArithmetics = new ProblemGeneratorBuilder()
    .Name("Paleo Arithmetics")
    .Description("Replace letters with digits such that SEND + MORE = MONEY")
    .Build(_ => new ProblemBuilder()
        // The usual digits domain
        .Domain(Domain.IntRange(0,9), out var dom)
        // The first digits domain (must be non-zero)
        .Domain(Domain.IntRange(1,9), out var domNotNull)
        // Carry domain
        .Domain(Domain.IntRange(0,1), out var domCarry)
        // The letter variables
        .Variable("S",domNotNull, out var S)
        .Variable("E",dom, out var E)
        .Variable("N",dom, out var N)
        .Variable("D",dom, out var D)
        .Variable("M",domNotNull, out var M)
        .Variable("O",dom, out var O)
        .Variable("R",dom, out var R)
        .Variable("Y",dom, out var Y)
        // carries for each digits place jump
        .Variable("c1",domCarry, out var C1)
        .Variable("c2",domCarry, out var C2)
        .Variable("c3",domCarry, out var C3)
        .Variable("c4",domCarry, out var C4)
        // unit places constraint
        .Constraint(Constraint.Of(D,E,Y,C1).IntegerRelation((d,e,y,c1) => d + e == y + 10 * c1))
        // tens places constraint
        .Constraint(Constraint.Of(N,R,C1,E,C2).IntegerRelation((n,r,c1,e,c2) => n + r + c1 == e + 10 * c2))
        // hundregs places constraint
        .Constraint(Constraint.Of(E,O,C2,N,C3).IntegerRelation((e,o,c2,n,c3) => e + o + c2 == n + 10 * c3))
        // thousands places constrain
        .Constraint(Constraint.Of(S,M,C3,O,C4).IntegerRelation((s,m,c3,o,c4) => s + m + c3 == o + 10 * c4))
        // Two 4-digit numbers add up to a 5-digit number, therefore, the first digit of the result must be the carry
        // [We can deduce it is actually 1 but we will let the solver find it]
        .Constraint(Constraint.Of(M,C4).IntegerRelation((m, c4) => m == c4))
        // Force all letters are different (drastically reduces solutions domain)
        .Constraint(Constraint.Of(S,E,N,D,M,O,R,Y).AllUnique())
        .Build()
    );

public static ProblemGenerator DinosaurMystery = new ProblemGeneratorBuilder()
    .Name("Dinosaur Mystery")
    .Description(
        "Five dinosaurs (Eucentrosaurus, Hadrosaurus, Herrerasaurus, Megasaurus, Nuoerosaurus)\n" +
        "lived in Argentina, Canada, China, England, or the United States.\n" +
        "Each dinosaur lived in exactly one country, and all countries are distinct."
    )
    .Build(_ => new ProblemBuilder()
        // Countries domain
        .Domain(Domain.StringLabels("Argentina","Canada","China","England","United States"), out var countries)

        // Country variables
        .Variable("Country_Euc",countries, out var cEuc)
        .Variable("Country_Had",countries, out var cHad)
        .Variable("Country_Her",countries, out var cHer)
        .Variable("Country_Meg",countries, out var cMeg)
        .Variable("Country_Nuo",countries, out var cNuo)

        // Sizes domain: 1=smallest, 5=largest
        .Domain(Domain.IntRange(1, 5), out var sizes)

        // Size variables
        .Variable("Size_Euc",sizes, out var sEuc)
        .Variable("Size_Had",sizes, out var sHad)
        .Variable("Size_Her",sizes, out var sHer)
        .Variable("Size_Meg",sizes, out var sMeg)
        .Variable("Size_Nuo",sizes, out var sNuo)

        // All dinosaurs have unique countries
        .Constraint(Constraint.Of(cEuc,cHad,cHer,cMeg,cNuo).AllUnique())

        // All sizes are unique
        .Constraint(Constraint.Of(sEuc,sHad,sHer,sMeg,sNuo).AllUnique())

        // China dinosaur is largest
        .Constraint(Constraint.Of(cEuc, cHad, cHer, cMeg, cNuo,
                                  sEuc, sHad, sHer, sMeg, sNuo)
            .FunctionConstraint(vals =>
            {
             // find the index of the China dinosaur and check if it has the largest size
             var dCountries = vals.Take(5).Select(v => v.EvaluateAs<string>()).ToArray();
                var dSizes = vals.Skip(5).Select(v => v.EvaluateAs<int>()).ToArray();

                int largest = dSizes.Max();
                int chinaIndex = Array.IndexOf(dCountries, "China");
                if (chinaIndex < 0) return false;
                return dSizes[chinaIndex] == largest;
            }))

        // Herrerasaurus was the smallest
        .Constraint(Constraint.Of(sHer).Equal(1))

        // Megasaurus larger than Hadrosaurus or Eucentrosaurus
        .Constraint(Constraint.Of(sMeg, sHad).IntegerRelation((meg, hads) => meg > hads))
        .Constraint(Constraint.Of(sMeg, sEuc).IntegerRelation((meg, eu) => meg > eu))

        // China and North America (Canada or U.S.) dinosaurs were plant eaters
        // Megasaurus ate meat => Megasaurus NOT in {China, U.S., Canada}
        .Constraint(Constraint.Of(cMeg).FunctionConstraint(vals =>
        {
            var c = vals[0].EvaluateAs<string>();
            return c != "China" && c != "Canada" && c != "United States";
        }))

        // Eucentrosaurus lived in North America -> Canada or U.S.
        .Constraint(Constraint.Of(cEuc).FunctionConstraint(vals =>
        {
            var c = vals[0].EvaluateAs<string>();
            return c == "Canada" || c == "United States";
        }))

        // North America (Canada/US) and England lived later than Herrerasaurus
        // This is not about time, but the fact that Herrerasaurus is not from NA (Ca/US) or EN
        .Constraint(Constraint.Of(cHer).FunctionConstraint(vals =>
        {
            var c = vals[0].EvaluateAs<string>();
            return c != "Canada" && c != "United States" && c != "England";
        }))


        // Hadrosaurus lived in Argentina or the U.S.
        .Constraint(Constraint.Of(cHad).FunctionConstraint(vals =>
        {
            var c = vals[0].EvaluateAs<string>();
            return c == "Argentina" || c == "United States";
        }))

        .Build()
    );
```



```
// very simple example on formal declaration of a problem
var problem = new ProblemBuilder()
    .Domain(Domain.IntRange(1, 5), out var dom)
    .Variable("x", dom, out var x)
    .Variable("y", dom, out var y)
    .Variable("z", dom, out var z)
    .Constraint(Constraint.Of(x, y).IntegerRelation((xVal, yVal) => xVal + yVal == 6))
    .Constraint(Constraint.Of(x, y).IntegerRelation((xVal, yVal) => xVal < yVal))
    .Constraint(Constraint.Of(z).IntegerRelation((zVal) => zVal > 3))
    .Build();

// Instantiate the Zebra problem for test purposes
problem = ProblemGeneratorTemplates.Riddle3H3N.GenerateProblem();

// example on applying reducers and solve the problem
var solution = problem
    .Reduce<NodeConsistencyReducer>()
    .Reduce<AC3Reducer>()
    .Reduce<GeneralizedArcConsistencyReducer>()
    .Solve<ForwardCheckingSolver>(onCandidateInspected: (label) => { Debug.WriteLine(label); })
    ;

// finally print the solution
Debug.WriteLine(solution);
```