1 Python-CPSolver

Is an initiative emerged within this research with the aim of providing a versatile and thoroughly understood tool based on Constraint Programming. We designed this CPSolver to comprehend various algorithms for searching equilibrium, as found in the literature and adapted to constraint technologies, while also to pioneering the development of new methods in the same direction. We don't aim to create a competitive tool; instead, we endeavoured to establish a user-friendly environment to validate our methods.

Keeping our core purpose in mind, Python-CPSolver [See GitHub] comprises four essential modules: Searching, Variables, Branching, and Propagators. Each of these modules was crafted with a minimalist approach, prioritising simplicity over efficiency. For a clearer grasp of our intent, consider our overuse of Python Lists, chosen for their versatility and utility over creating specialised data structures for similar tasks.

Figure 1 provides a comprehensive overview of the inner workings of Python-CPSolver. It is implemented using Object-Oriented Programming, drawing inspiration from the structures of Gecode and Chuffed. Some identifiers of classes, attributes, and methods are borrowed from these two solvers, while other segments of the code are based on the CPFloat-Gecode project [See].

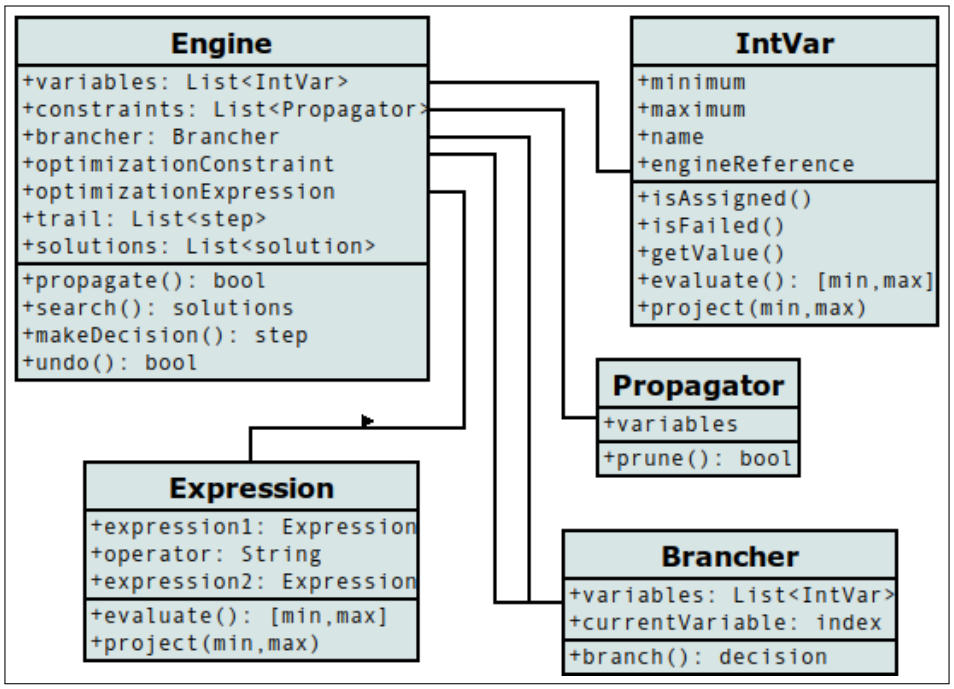

Figure 1: General UML for Python-CPSolver

At the heart of Python-CPSolver lies the Engine class, which orchestrates the search process and serves as the nexus for other classes. Within this class, we find storage for sets of variables and propagators, as well as the optimisation expression and its corresponding propagator. In the latest version (Version-Trail), this class also maintains the trail, tracking the progress of the ongoing search. Its central procedure, "search", iteratively calls upon "propagate", "makeDecision", and "undo" to navigate through the search space.

Once the Solver fulfils its purpose, the Propagators module becomes pivotal, housing our equilibrium search methods. Often, this involves modifying other modules to align with the algorithms under study, resulting in the creation of multiple versions of constraint methods. Figure 2 illustrates the design process behind these developments.

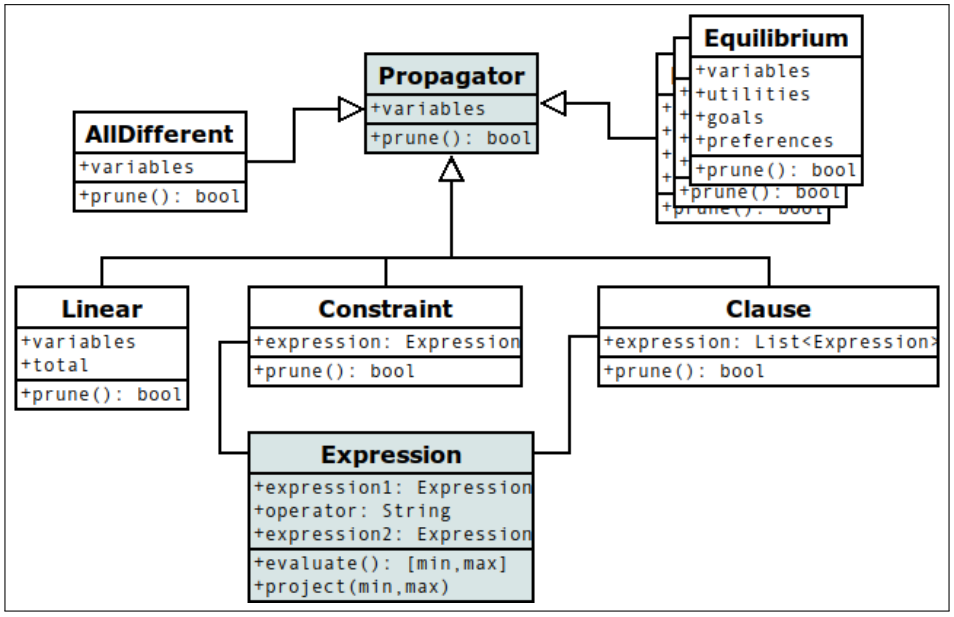

Figure 2: Propagators module for Python-CPSolver

We crafted this module with a commitment to simplicity while preserving the essence of Constraint technologies. Each implementation of a propagator retains a copy of the variables, either directly or through an expression. Additionally, these classes override the "prune" method, responsible for filtering and checking satisfiability.

Several versions of propagators searching Equilibrium will be detailed in the following paragraphs. However, it's crucial to emphasise the significance of the Constraint propagator. We employ an HC4 algorithm [**?**] to evaluate and filter mathematical expressions using Box Consistency. These expressions are stored in a tree structure using the Expression class. The algorithm proceeds in two phases: the first evaluates the expression upward, while the second projects the filtering downward. Figure 3, taken from [**?**], illustrates both phases.

> *Example 1: Let S, E, N, D, M, O, R, and Y represent distinct digits from 0 to 9. Find a digit substitution for each letter such that the equation SEND + MORE = MONEY holds true. No two letters represent the same digit.*

Combined with other straightforward implementations, we have developed a user-friendly tool akin to a laboratory environment, where we can confidently test our algorithms. Python's simplicity played a significant role in achieving this[**?**]. Figure 4 illustrates the implementation of Example 1, tackling the puzzle SEND+MORE=MONEY.

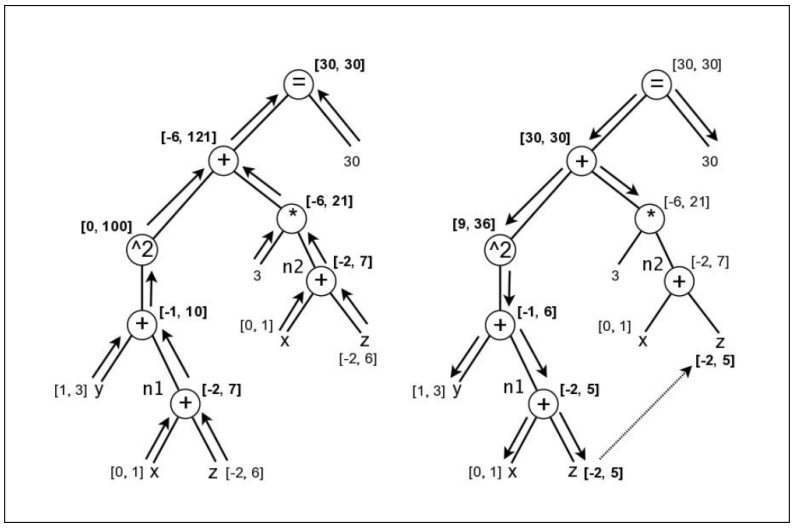

Figure 3: Evaluation and narrowing in the HC4-revise algorithm. The tree represents the constraint: $(x + y + z)2 + 3(x + z) = 30$.

```
V = [s,e,n,d,m,o,r,y] = IntVarArray(8,0,9)

C = [
    AllDifferent( V ),
    Constraint( s > 0 ),
    Constraint( m > 0 ),
    Constraint( s*1000 + e*100 + n*10 + d*1 +
                m*1000 + o*100 + r*10 + e*1 ==
      m*10000 + o*1000 + n*100 + e*10 + y*1 ),
]

e = Engine(V, C)
S = e.search(ALL)

for _ in S :
    print( intVarArrayToStr(_) )
```

Figure 4: Using Python-CPSolver to implement the CSP **??**.