Constraint Programming as an AI Option

Russ ABBOTT
Department of Computer Science
California State University, Los Angeles
Los Angeles, California 90032
Email: rabbott@calstatela.edu

Jun Soo LIM
Department of Computer Science
California State University, Los Angeles
Los Angeles, California 90032
Email: jlim34@calstatela.edu

ABSTRACT [1]

We examine the history of Artificial Intelligence, from its audacious beginnings to the current day. We argue that constraint programming (a) is the rightful heir and modern-day descendent of that early work and (b) offers a more stable and reliable platform for AI than deep machine learning.

We offer a tutorial on constraint programming solvers that should be accessible to most software developers. We show how constraint programming works, how to implement constraint programming in Python, and how to integrate a Python constraint-programming solver with other Python code.

1. INTRODUCTION

Symbolic artificial intelligence. The birth announcement for Artificial Intelligence took the form of a workshop proposal. The proposal predicted that *every aspect of learning—or any other feature of intelligence—can in principle be so precisely described that a machine can be made to simulate it.*[26]

At the workshop, held in 1956, Newell and Simon claimed that their Logic Theorist not only took a giant step toward that goal but even *solved the mind-body problem*.[33] A year later Simon doubled-down.

> [T]here are now machines that can think, that can learn, and that can create. Moreover, their ability to do these things is going to increase rapidly until—in a visible future—the range of problems they can handle will be coextensive with the range to which the human mind has been applied.[38]

1 Acknowledgement: We would like to express our deeply felt gratefulness to Professor Raj Pamula and Mohammad Pourhomayoun for their comprehensive and detailed peer-reviewing of this document.

Perhaps not unexpectedly, such extreme optimism about what is now known as *symbolic AI* faded into the gloom of a long *AI winter*.

Deep learning. But winter was followed by spring and the green shoots of (a) expert systems, which came and went relatively quickly, and more promisingly (b) (non-symbolic) deep neural networks. Andrew Ng said of that development,

> Just as electricity transformed almost everything 100 years ago, today I have a hard time thinking of an industry that won't be transformed by AI.[28]

Deep learning has achieved extraordinary success in fields such as image captioning and natural language translation.[17] But other than its remarkable achievements in game-playing via reinforcement learning[37], it's triumphs have often been superficial. For example, deep neural nets

> are surprisingly susceptible to what are known as *adversarial* attacks. Small perturbations that are (virtually) imperceptible to humans can cause a neural network to completely change its prediction: a correctly classified image of a school bus is reclassified as an ostrich. Even worse, the classifiers report high confidence in these wrong predictions.[1]

We do not argue that work in deep neural nets is trivial. But we suggest that many deep learning systems learn little more than surface patterns. The patterns may be both subtle and complex, but they are surface patterns nevertheless.

Lacker[24] elicits many examples of such superficial (but sophisticated) patterns from GPT-3[6], a highly acclaimed natural language system. In one, GPT-3 offers to read to its interlocutor his latest email. The problem is that GPT-3 has no access to that person's email—and doesn't "know" that without access it can't read the email.

Both the conversational interaction and the made-up email sound plausible and natural. In reality, each consists of words strung together based simply on co-occurrences that GPT-3 found in the billions upon billions of word sequences it had scanned. Although what GPT-3 produces sounds like coherent English, it's all surface patterns with no underlying semantics.

Recent work[18] (see [8] for a popular discussion) suggests that much of the success of deep learning, at least when applied to image categorization, derives from the tendency of deep learning systems to focus on textures—the ultimate surface feature—rather than shapes. This insight offers an explanation for some of deep learning's brittleness and superficiality—along with possible mitigation strategies.

The Holy Grail: constraint programming. In the meantime, work on symbolic AI continued. Constraint programming was born in the 1980s as an outgrowth of the interest in logic programming triggered by the Japanese Fifth Generation initiative.[36] Logic Programming led to Constraint Logic Programming, which evolved into Constraint Programming. (A familiar example is the n-queens problem: place n queens on an n *x* n chess board so that no queen threatens any other. Constraint programming also has many practical applications.)

In 1997, Eugene Frueder characterized constraint programming as *the Holy Grail of computer science: the user simply states the problem and the computer solves it.*[16] Software that solves constraint programming problems is known as a solver. Solver technology has many desirable properties.

- Solutions found by constraint programming solvers actually solve the given problem. There is no issue of how "confident" the solver is in the solutions it finds.
- One can understand how the solver arrived at the solution. This contrasts with the frustrating feature of neural nets that the solutions they find are generally hidden within a maze of parameters, unintelligible to human beings.
- The structure and limits of constraint programming are well understood: there will be no grand disappointments like those that followed the birth of AI—unless quantum computing turns out to be a bust.
- Constraint programming is closely related to computational complexity, which provides a well-studied theoretical framework.
- There will be no surprises such as adversarial images.
- Solver technology is easy to characterize. It is an exercise in search: find values for uninstantiated variables that satisfy the constraints.
- Improvements are generally incremental and consist primarily of new heuristics and better search strategies. For example, in the n-queens problem one can propagate solution steps by marking as unavailable board squares that are threatened by newly placed pieces. This reduces search times. We will see example heuristics below.

Constraint programming solvers are now available in multiple forms. MiniZinc[41] allows users to express constraints in what is essentially executable predicate calculus.

Solvers are also available as package add-ons to many programming languages: Choco[31] and JaCoP[23] (two Java libraries), OscaR/CBLS[29] and Yuck[43] (two Scala libraries), and Google's OR-tools[21] (a collection of C++ libraries, which sport Python, Java, and .NET front ends).

In the systems just mentioned, the solver is a black box. One defines a problem, either directly in predicate calculus or in the host language, and then asks the solver to solve it.

This can be frustrating for those who want more insight into the internal workings of the solvers. Significantly more insight is available when working either (a) in a system like Picat[45], a language that combines features of logic programming and imperative programming, or (b) with Prolog (say either SICStus Prolog[7] or SWI Prolog[39]) to which a Finite Domain package has been added. But these options are accessible only to those with a logic programming background.

Shallow embeddings. Solver capabilities may be implemented directly in a host language and made available to programs in that language.[22, 20] Recent examples include Kanren[32], a Python embedding, and Muli[11], a Java embedding.

Most shallow embeddings have well-defined APIs; but like libraries, their inner workings are not visible. Kanren is open source, but it offers no implementation documentation. The description of the Muli virtual machine[10] is quite technical.

Back to basics. This brings us to our goal for the rest of this paper: to offer an under-the-covers tutorial about how an embeddable solver works.

One can think of Prolog as the skeleton of a constraint satisfaction solver. Consequently, we focus on Prolog as a basic paradigmatic solver. We describe Pylog, a Python shallow embedding of Prolog's core capabilities.

Our primary focus will be on helping readers understand how Prolog's two fundamental features, backtracking and logic variables, can be implemented *simply and cleanly* in Python. We also show how two common heuristics can be added.

Pylog should be accessible to anyone reasonably fluent in Python. In addition, the techniques we use are easily transferred to many other languages.

We stress *simply and cleanly*. An advantage we have over earlier Prolog embeddings is Python generators. Without generators, one is pushed to more complex implementations, such as continuation passing[3] or monads[35]. Generators, which are now widespread[19], eliminate such complexity.

We did not invent the use of generators for implementing backtracking; it has a nearly two-decade history: [4, 5, 12, 15, 27, 40, 34, 9, 25]. We would like especially to thank Ian Piumarta[30]; Pylog began as a fork of his efforts. We build on this record and offer a cleanly coded, well-explained, and fully operational solver.

2. SOLVER BASICS AND HEURISTICS

As an example problem we will use the computation of a transversal. Given a sequence of sets, a transversal is a non-repeating sequence of elements with the property that the n^{th} element of the traversal belongs to the n^{th} set in the sequence. For example, the set sequence {1, 2, 3}, {1, 2, 4}, {1} has three transversals: [2, 4, 1], [3, 2, 1], and [3, 4, 1].

This problem can be solved with a simple depth-first search. Here's a high level description.

- Look for transversal elements from left to right.
- Select an element from the first set and (tentatively) assign that as the first element of the transversal.
- Recursively look for a transversal for the rest of the sets—being sure not to reuse any already selected elements.
- If, at any point, we cannot proceed, say because we have reached a set all of whose elements have already been used, go back to an earlier set, select a different element from that set, and proceed forward.

Following are (a) a utility function (Listing 1) and then (b) *tnvsl_dfs* (Listing 2), the solver. (Please pardon our Python style deficiencies. The column width and page limit compelled compromises.)

```
unassigned = '_'
def uninstantiated_indices(transversal):
  """ Find indices of uninstantiated components. """
  return [indx for indx in range(len(transversal))
                     if transversal[indx] is unassigned]
```

Listing 1. *uninstantiated_indices*

```
1  def tnvsl_dfs(sets, tnvsl):
2    remaining_indices = uninstantiated_indices(tnvsl)
3    if not remaining_indices: return tnvsl
4
5    nxt_indx = min(remaining_indices)
6    for elmt in sets[nxt_indx]:
7      if elmt not in tnvsl:
8        new_tnvsl = tnvsl[:nxt_indx] \
9                    + (elmt, ) \
10                   + tnvsl[nxt_indx+1:]
11       full_tnvsl = tnvsl_dfs(sets, new_tnvsl)
12       if full_tnvsl is not None: return full_tnvsl
```

Listing 2. *tnvsl_dfs*

Here's an explanation of the search engine in some detail.

- The function *tnvsl_dfs* takes two parameters:
  1) *sets*: a list of sets
  2) *tnvsl*: a tuple with as many positions as there are sets, but initialized to undefined.
- *line 2*. *remaining_indices* is a list of the indices of uninstantiated elements of *tnvsl*. Initially this will be all of them. Since *tnvsl_dfs* generates values from left to right, the first element of *remaining_indices* will always be the leftmost undefined index position.
- *line 3*. If *remaining_indices* is null, we have a complete transversal. Return it. Otherwise, go on to *line 5*.
- *line 5*. Set *nxt_indx* to the first undefined index position.
- *line 6*. Begin a loop that looks at the elements of *sets[nxt_indx]*, the set at position *nxt_indx*. We want an element from that set to represent it in the transversal.
- *line 7*. If the currently selected *elmt* of *sets[nxt_indx]* is not already in *tnvsl*:
  1) *lines 8-10*. Put *elmt* at position *nxt_indx*.
  2) *line 11*. Call *tnvsl_dfs* recursively to complete the transversal, passing *new_tnvsl*, the extended *tnvsl*. Assign the returned result to *full_tnvsl*.
  3) *line 12*. If *full_tnvsl* is not **None**, we have found a transversal. Return it to the caller. If *full_tnvsl* is **None**, the *elmt* we selected from *sets[nxt_indx]* did not lead to a complete transversal. Return to *line 6* to select another element from *sets[nxt_indx]*.

This is standard depth first search. *tnvsl_dfs* will either find the first transversal, if there are any, or return **None**.

Here's a trace of the recursive calls.

```
1  sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,_)
2    sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (1,_,_)
3      sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (1,2,_)
4      sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (1,4,_)
5    sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (2,_,_)
6      sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (2,1,_)
7      sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (2,4,_)
8        sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (2,4,1)
```

Listing 3. *tnvsl_dfs trace*

- *line 1*. Initially (and on each call) the *sets* are

$$\{1,2,3\},\{1,2,4\},\{1\}$$

Initially *tnvsl* is completely undefined: *(_, _, _)*
- *line 2*. *1* is selected as the first element of *tnvsl*.
- *line 3*. *2* is selected as the second element.
- *line 4*. But now we are stuck. Since *1* is already in *tnvsl*, we can't use it as the third element. Since depth first search is "blind," instead of selecting an alternative for the first set, it backs up to the most recent selection and selects *4* to represent the second set.
- *lines 5*. Of course, that doesn't solve the problem. So we back up again. Since we have now tried all elements of the second set, we back up to the first set and select *2*.
- *lines 6*. Going forward, we select *1* for the second set.
- *lines 7*. Again, we cannot use *1* for the third set. So we back up and select 4 to represent the second set. (We can't use *2* since it is already taken.)
- *lines 8*. Finally, we can select *1* as the third element of *tnvsl*, and we're done.

How recursively nested for-loops implement choice-points and backtracking. This simple depth-first search appears to incorporate backtracking. In fact, there is no backtracking. Recursively nested **for**-loops produce a back-tracking effect.

It is common to use the term *choicepoint* for a place in a program where (a) multiple choices are possible and (b) one wants to try them all, if necessary. Our simple solver implements choicepoints via (recursively) nested **for**-loops.

The **for**-loop on line 6 generates options until either we find one for which the remainder of the program completes the traversal, or, if the options available have been exhausted, the program fails out of that recursive call and "backtracks" to a choicepoint at a higher/earlier level of the recursion.

In this context, backtracking means popping an element from the call stack and restoring the program at the next higher level. As with any function call, the calling function continues at the point after the function call—in this case, line 12.

If the function called on line 11 returns a complete transversal, we return it to the *next* higher level, which continues to return it up the stack until we reach the original caller.

If what was returned on line 11 was not a complete transversal, we go around the **for**-loop again, bind *element* to the next member of *sets[nxt_indx]*, and try again.

The call stack serves as a record of earlier, pending choicepoints. We resume them in reverse order as needed. That's exactly what depth-first search is all about.

We now turn to two heuristics that improve solver efficiency.

Propagate. When we select an element for *trvs* we can *propagate* that selection by removing that element from the remaining sets. We can do that with the following changes. (Of course, a real solver would not hard-code heuristics. This is just to show how it works.)

1) Before *line 11*, insert this line.

```
new_sets = [set - {elmt} for set in sets]
```

Then replace *sets* with *new_sets* in *line 11*. This will remove *elmt* from the remaining sets.

2) Before *line 5*, insert

```
if any(not sets[idx] for idx in remaining_indices):
  return None
```

This tests whether any of our unrepresented sets are now empty. If so, we can't continue. (Recall that Python style recommends treating a set as a boolean when testing for emptiness. An empty set is considered **False**.)

Because the empty sets in lines 2 and 4 of the trace trigger backtracking, the execution takes 6 steps rather than 8.

```
1 sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,_)
2   sets: [{2,3}, {2,4}, set()], tnvsl: (1,_,_)
3   sets: [{1,3}, {1,4}, {1}], tnvsl: (2,_,_)
4     sets: [{3}, {4}, set()], tnvsl: (2,1,_)
5     sets: [{1,3}, {1}, {1}], tnvsl: (2,4,_)
6       sets: [{3}, set(), set()], tnvsl: (2,4,1)
```

Listing 4. *tnvsl_dfs_prop trace*

The *Propagate* heuristic is a partial implementation of the *all-different* constraint. It applies to this problem because we know that the transversal elements must all be distinct.

Smallest first. When selecting which *tnvsl* index to fill next, pick the position associated with the smallest remaining set.

In the original code (Listing 2), replace line 5 with

```
nxt_indx = min(remaining_indices,
               key=lambda indx: len(sets[indx]))
```

The resulting trace (Listing 5) is only 4 lines. (At line 3, the first two sets are the same size. *min* selects the first.)

```
1 sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,_)
2   sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,1)
3     sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (2,_,1)
4       sets: [{1,2,3}, {1,2,4}, {1}, tnvsl: (2,4,1)
```

Listing 5. *tnvsl_dfs_smallest trace*

One could apply both heuristics. Since *smallest first* eliminated backtracking, adding the *propagate* heuristic makes no effective difference. But, one can watch the sets shrink.

```
sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,_)
  sets: [{2,3}, {2,4}, {}], tnvsl: (_,_,1)
    sets: [{3}, {4}, {}], tnvsl: (2,_,1)
      sets: [{3}, {}, {}, tnvsl: (2,4,1)
```

Listing 6. *tnvsl_dfs_both_heuristics trace*

This concludes our discussion of a basic depth-first solver and two useful heuristics. We have yet to mention generators.

3. GENERATORS

In our previous examples, we have been happy to stop once we found a transversal, any transversal. But what if the problem were a bit harder and we were looking for a transversal whose elements added to a given sum. The solvers we have seen so far wouldn't help—unless we added the new constraint to the solver itself. But we don't want to do that. We want to keep the transversal solvers independent of other constraints. (Adding heuristics don't violate this principle. Heuristics only make solvers more efficient.)

One approach would be to modify the solver to find and return all transversals. We could then select the one(s) that satisfied our additional constraints. But what if there were many transversals? Generating them all before looking at any of them would waste a colossal amount of time.

We need a solver than can return results while keeping track of where it is with respect to its choicepoints so that it can continue from there if necessary. That's what a generator does.

Listing 7 shows a generator version of our solver, including both heuristics. When called as on lines 22-23, it produces the trace in Listing 8.

Some comments on Listing 7.

```
1  def tnvsl_dfs_gen(sets, tnvsl):
2    remaining_indices = uninstantiated_indices(tnvsl)
3
4    if not remaining_indices: yield tnvsl
5    else:
6      if any(not sets[i] for i in remaining_indices):
7        return None
8
9      nxt_indx = min(remaining_indices,
10                    key=lambda indx: len(sets[indx]))
11     for elmt in sets[nxt_indx]:
12       if elmt not in tnvsl:
13         new_tnvsl = tnvsl[:nxt_indx] \
14                     + (elmt, ) \
15                     + tnvsl[nxt_indx+1:]
16         new_sets = [set - {elmt} for set in sets]
17         for full_tnvsl in tnvsl_dfs_gen(new_sets,
18                                         new_tnvsl):
19           yield full_tnvsl
20
21
22 for tnvsl in tnvsl_dfs_gen(sets, ('_','_','_')):
23     print('=> ', tnvsl)
```

Listing 7. *tnvsl_dfs_gen*

```
sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_,_,_)
  sets: [{2,3}, {2,4}, {}], tnvsl: (_,_,1)
    sets: [{3}, {4}, {}], tnvsl: (2,_,1)
      sets: [{3}, {}, {}], tnvsl: (2,4,1)
=>  (2, 4, 1)
    sets: [{2}, {2,4}, {}], tnvsl: (3,_,1)
      sets: [{}, {4}, {}], tnvsl: (3,2,1)
=>  (3, 2, 1)
      sets: [{2}, {2}, {}], tnvsl: (3,4,1)
=>  (3, 4, 1)
```

Listing 8. *tnvsl_dfs_gen trace*

- The newly added **else** on line 5 is necessary. Previously, if there were no *remaining_indices*, we returned *tnvsl*. That was the end of execution for this recursive call. But if we **yield** instead of **return**, when *tnvsl_dfs_gen* is asked for more results, *it continues with the line after the **yield***. But if have already found a transversal, we don't want to continue. The **else** divides the code into two mutually exclusive components. **return** had done that implicitly.
- Lines 17-20 call *tnvsl_dfs_gen* recursively and ask for all transversals that can be constructed from the current state. Each one is then **yield**ed. No need to exclude **None**. *tnvsl_dfs_gen* will **yield** only complete transversals.

  Lines 17-20 can be replaced by this single line.

```
    yield from tnvsl_dfs_gen(new_sets, new_tnvsl)
```

Let's use *tnvsl_dfs_gen* (Listing 7) to find a transversal whose elements sum to, say, 6.

```
n = 6
for tnvsl in tnvsl_dfs_gen(sets, ('_','_','_')):
  sum_string = ' + '.join(str(i) for i in tnvsl)
  equals = '==' if sum(tnvsl) == n else '!='
  print(f'{sum_string} {equals} {n}')
  if sum(tnvsl) == n: break
```

Listing 9. *running tnvsl_dfs_gen*

The output (without trace) will be as follows.

```
    2 + 4 + 1 != 6
    3 + 2 + 1 == 6
```

Listing 10. *tnvsl_dfs_gen trace*

We generated transversals until we found one whose elements summed to 6. Then we stopped.

4. LOGIC VARIABLES

This section discusses logic variables and their realization.

A. Instantiation

Logic variables are either instantiated, i.e., have a value, or uninstantiated. The instantiation operation is called *unify*. *unify* is a *generator*, but it *does not* **yield** a value. Consider the code segment in Listing 11.

```
1 A = Var()
2 print(A)  # => _1
3 for _ in unify(A, 'abc'):
4     print(A)  # => abc
5     # This unify fails. Its body never runs.
6     for _ in unify(A 'def'):
7       print(A)  # Never executed
8     print(A)  # => abc
9 print(A)  # => _1
```

Listing 11. *Unification example*

- *line 1*. *A* is a normal Python identifier. We use an initial capital to distinguish logic variables from regular Python variables. *Var* is the constructor for logic variables. After this line, *A* refers to an uninstantiated logic variable object.
- *line 2*. When an uninstantiated logic variable is printed, we see an internal value, which distinguishes it from other logic variables. As the first logic variable in this program, *A*'s internal value is *_1*.
- *lines 3-8*. *unify A* with *abc*. Since *unify* does not **yield** a value, the **for**-loop variable is not used.
- *line 4*. The **for**-loop establishes a context for *unify*. Within the **for**-loop body *A* is instantiated to *abc*.
- *lines 6-7*. Within a *unify* context, logic variables are immutable. Since *A* already has a value, it cannot be unified with *def*. The *unify* on line 6 fails, and the body of that **for**-loop (line 7) does not execute.
- *line 8*, *A* has the same value as on line 4.

  Since there is only one way to *unify A* with *abc*, the **for**-loop body runs only once.
- *lines 9*. Leaving the *unify* context undoes the instantiation.

B. The power of unify

unify can also identify logic variables with each other. After two uninstantiated logic variables are unified, whenever either gets a value, the other gets that same value.

Unification is surprisingly straightforward. Each *Var* includes a *next* field, which is initially **None**. When two *Var*s are unified, the result depends on their states of instantiation.

- If both are uninstantiated the *next* field of one points to the other. It makes no difference which points to which. A chain of linked *Var*s unifies all the *Var*s in the chain.
- If only one is uninstantiated, the uninstantiated one points to the other.
- If both are instantiated to the same value, they are effectively unified. *unify succeeds* but nothing changes.
- If both are instantiated but to different values, *unify fails*.

A note on terminology. When called (as part of a **for**-loop) a generator will either **yield** or **return**. When a generator **yield**s, it is said to *succeed*; the **for**-loop body runs. When a generator **return**s, it is said to *fail*; the **for**-loop body does not run. Instead we exit the **for**-loop.

We can trace the unifications in Listing 12.

```
1  (A, B, C, D) = (Var(), Var(), Var(), Var())
2  print(A, B, C, D) # => _1 _2 _3 _4
3  for _ in unify(A, B):
4    for _ in unify(D, C):
5      print(A, B, C, D) # => _2 _2 _3 _3
6      for _ in unify(A, 'abc'):
7        print(A, B, C, D) # => abc abc _3 _3
8        for _ in unify(A, D):
9          print(A, B, C, D) # => abc abc abc abc
10       print(A, B, C, D) # => abc abc _3 _3
11     print(A, B, C, D) # => _2 _2 _3 _3
12   print(A, B, C, D) # => _2 _2 _3 _4
13 print(A, B, C, D) # => _1 _2 _3 _4
```

Listing 12. *Unification example*

The first unifications, lines 3 and 4, produce the following internal structures.

$$\begin{array}{ccc} A & \rightarrow & B \\ D & \rightarrow & C \end{array} \tag{1}$$

Line 6 unifies *A* and *'abc'*. The first step is to go to the ends of the relevant unification chains. In this case, *B* (the end of *A*'s unification chain) is pointed to *'abc'*. Since *'abc'* is instantiated, the arrow can only go from *B* to *'abc'*.

$$\begin{array}{ccccc} A & \rightarrow & B & \rightarrow & 'abc' \\ & & D & \rightarrow & C \end{array} \tag{2}$$

Finally, line 8 unifies *A* with *D*. *C* (the end of *D*'s unification chain) is set to point to *'abc'* (the end of *A*'s unification chain).

$$\begin{array}{ccccc} A & \rightarrow & B & \rightarrow & 'abc' \\ & & & & \uparrow \\ & & D & \rightarrow & C \end{array} \tag{3}$$

C. A logic-variable version of tnvsl_dfs_gen

Listing 13 adapts Listing 7 for logic variables. The strategy is for *trnsvl* to start as a tuple of uninstantiated *Var*s, which become instantiated as the program runs.

First, an adapted *uninstan_indices_lv* returns the indices of the uninstantiated *Var*s in *trnsvl*.

```
def uninstan_indices_lv(tnvsl):
  return [indx for indx in range(len(tnvsl))
                if not tnvsl[indx].is_instantiated()]
```

Note that *tnvsl[indx]* retrieves the *indxth tnvsl* element. If it is instantiated, it represents the value associated with the *indxth* set. If not, we don't yet have a value for the *indxth* set.

```
1  def tnvsl_dfs_gn_lv(sets, tnvsl):
2    var_indxs = uninstan_indices_lv(tnvsl)
3
4    if not var_indxs: yield tnvsl
5    else:
6      empty_sets = [sets[indx].is_empty()
7                    for indx in var_indxs]
8      if any(empty_sets): return None
9
10     nxt_indx = min(var_indxs,
11                    key=lambda indx: len(sets[indx]))
12     used_values = PyList([tnvsl[i]
13                           for i in range(len(tnvsl))
14                           if i not in var_indxs])
15     T_Var = tnvsl[nxt_indx]
16       for _ in member(T_Var, sets[nxt_indx]):
17         for _ in fails(member)(T_Var, used_values):
18           new_sets = [set.discard(T_Var)
19                       for set in sets]
20           yield from tnvsl_dfs_gn_lv(new_sets,tnvsl)
```

Listing 13. *dfs-with-gen-and-logic-variables*

Some comments on Listing 13. (We reformatted some of the lines and changed some of the names from *tnvsl_-dfs_gen* (Listing 7) so that the program will fit the width of a column.)

- *line 6*. The parameter *sets* is a list of *PySet*s. These are logic variable versions of sets. An *is_empty* method is defined for them.
- *lines 12-14*. *used_values* are the values of the instantiated *tnvsl* elements.
- *line 15*. *T_Var* is the element at the *nxt_indxth* position of *tnvsl*. Since *nxt_indx* was selected from the uninstantiated variables, *T_Var* is an uninstantited *Var*.
- *line 16*. *member* successively unifies its first argument with the elements of its second argument. It's equivalent to ***for** T_Var **in** sets[nxt_indx]* but using unification.
- *line 17*. *fails* takes a predicate as its argument. It converts the predicate to its negation. So *fails(member)* succeeds if and only if *member* fails.
- *line 18*. *PySet*s have a *discard* method that returns a copy of the *PySet* without the argument.

When run, we get the same result as before—except that the uninstantiated transversal variables appear as we saw above.

```
sets: [{1,2,3}, {1,2,4}, {1}], tnvsl: (_1, _2, _3)
  sets: [{2,3}, {2,4}, {}], tnvsl: (_1, _2, 1)
    sets: [{3}, {4}, {}], tnvsl: (2, _2, 1)
      sets: [{3}, {}, {}], tnvsl: (2, 4, 1)
=> (2, 4, 1)
    sets: [{2}, {2,4}, {}], tnvsl: (3, _2, 1)
      sets: [{}, {4}, {}], tnvsl: (3, 2, 1)
=> (3, 2, 1)
      sets: [{2}, {2}, {}], tnvsl: (3, 4, 1)
=> (3, 4, 1)
```

The following logic variable version of Listing 9 will run *tnvsl_dfs_gen_lv* and produce the same result.

```
1 (A, B, C) = (Var(), Var(), Var())
2 Py_Sets = [PySet(set) for set in sets]
3 # PyValue creates a logic variable constant.
4 N = PyValue(6)
5 for _ in tnvsl_dfs_gn_lv(Py_Sets, (A, B, C)):
6   sum_string = ' + '.join(str(i) for i in (A, B, C))
7   equals = '==' if A + B + C == N else '!='
8   print(f'{sum_string} {equals} {N}')
9   if A + B + C == N: break
```

Line 1 created three logic variables, *A*, *B*, and *C*. Line 5 passed them to *tnvsl_dfs_gn_lv*. Each time a transversal is found, the body of the **for**-loop is executed with the values to which *A*, *B*, and *C* have been instantiated.

The preceding offers some sense of what one can do with logic variables. The next section really puts them to work.

5. A LOGIC PUZZLE

At this point, one might expect a complex logic puzzle like the Zebra Puzzle[44]. Instead we present a similar but much simpler puzzle. The techniques are the same, but the following puzzle[14] fits the available space better.

- There are four students: Ada, Emmy, Lynn, and Marie. Each has a scholarship and a major. No two students have the same scholarship or the same major.
- The scholarships and majors are $25,000, $30,000, $35,000 and $40,000 and Bio, CS, Math, and Phys.

From the clues listed below, determine which student studies which major and the amount of each student's scholarship.

We create a **class** *Stdnt*. Each instances has two fields: *name* and *major*. (We do *not* keep track of the students' scholarships!) For example, a *Stdnt* object that represents *Ada* studying *Phys* is constructed like this *Stdnt(name='Ada', major='Phys')* and printed as *Ada/-Phys*.

Objects are not always fully instantiated. Missing information is represented by an underscore (_). An object that represents some person studying *Bio* would look like this *_/Bio*. It would be constructed as: *Stdnt(major='Bio')*.

Our *world* consists of a list of *Stdnt* objects with scholarships of increasing size. (Although we don't record scholarship amounts, we know their relative sizes!) This list is passed to the clues and will become fully instantiated as the answer.

A number of utility methods are defined.

- *is_contiguous_in(list1, list2)* unifies the elements of *list1* with those of *list2* if the elements of *list1* appear together in *list2* in the same order as in *list1*. On backtracking, yields all possible matches.

  Unification fails between objects with instantiated fields having different values. For example *Marie/-Physics* would not unify with *_/Math*.

  But *Marie/_* would unify with *_/Phys*. After unification, the two objects would each have both fields identically instantiated: *Mia/Physics*.
- *is_subseq(list1, list2)* is the same as *is_contiguous_in*, but the elements of *list1* may appear in *list2* with gaps between them.
- *member(student, list)* unifies *student*, successively, with eligible elements of *list*, as in the transversal problem.

Listing 14 contains the clues. Listing 15 contains a list of the clues followed by the search engine on lines 3-7. *run_clue* (line 6) runs the clues. Although not shown, it also applies the *all-different* heuristic to prevent the same field value from being used more than once.

Listing 16 shows the sequence of clue executions, including backtracking. Each line shows the then-current list of partially instantiated students. At line 42 we asked the search engine to look for additional solutions. (There weren't any.) The total compute time on a 3-year-old laptop was 0.01 sec.

6. CONCLUSION

We explained how a simple solver for constraint problems works and how solvers can be integrated into Python programs.

It's difficult to imagine a neural net (of any depth!) solving the problems discussed here—although preliminary work toward that end has been reported. [42, 2, 13]

Pylog code at: github.com/RussAbbott/pylog/tree/master/pylog.

REFERENCES

[1] Naveed Akhtar and Ajmal Mian. "Threat of adversarial attacks on deep learning in computer vision: A survey". In: *IEEE Access* 6 (2018), pp. 14410–14430.

[2] Kay R Amel. "From shallow to deep interactions between knowledge representation, reasoning and machine learning". In: *Proceedings 13th International Conference Scala Uncertainity Mgmt (SUM 2019), Compiègne, LNCS*. 2019, pp. 16–18.

[3] Nada Amin, William E Byrd, and Tiark Rompf. "Lightweight Functional Logic Meta-Programming". In: *Asian Symposium on Programming Languages and Systems*. Springer. 2019, pp. 225–243.

[4] S Berger. *Pythologic: Prolog syntax in Python*. http://code.activestate.com/recipes/. 2004.

[5] Carl Friedrich Bolz. *A Prolog Interpreter in Python*. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.121.8625&rep=rep1&type=pdf. 2007.

[6] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL].

[7] Mats Carlsson. *SICStus Prolog User's Manual v4.3*. BoD–Books on Demand, 2014.

[8] Jordana Cepelewicz. "Where We See Shapes, AI Sees Textures". In: *Quanta Magazine* (2020).

[9] BKM Cesar. *Prol: a minimal, inefficient Prolog interpreter in Python*. https://gist.github.com/brunokim/. 2019.

```
def clue_1(self, Stdnts):
  """ The student who studies Phys gets a smaller scholarship than Emmy. """
  yield from is_subseq([Stdnt(major='Phys'),Stdnt(name='Emmy')],Stdnts)

def clue_2(self, Stdnts):
  """ Emmy studies either Math or Bio. """
  # Create Major as a local logic variable.
  Major = Var()
  for _ in member(Stdnt(name='Emmy', major=Major),Stdnts):
    yield from member(Major, PyList(['Math','Bio']))

def clue_3(self, Stdnts):
  """ The Stdnt who studies CS has a $5,000 larger scholarship than Lynn. """
  yield from is_contiguous_in([Stdnt(name='Lynn'), Stdnt(major='CS')], Stdnts)

def clue_4(self, Stdnts):
  """ Marie gets $10,000 more than Lynn. """
  yield from is_contiguous_in([Stdnt(name='Lynn'),Var(),Stdnt(name='Marie')],Stdnts)

def clue_5(self, Stdnts):
  """ Ada has a larger scholarship than the Stdnt who studies Bio. """
  yield from is_subseq([Stdnt(major='Bio'), Stdnt(name='Ada')],Stdnts)
```

Listing 14. *The clues*

```
self.clues = [clue_1,clue_2,clue_3,clue_4,clue_5]

def run_all_clues(self, clue_number):
  if clue_number >= len(self.clues): yield
  else:
    for _ in self.run_clue(clue_number):
      yield from self.run_all_clues(clue_number + 1)
```

Listing 15. *search engine*

```
Initially: _/_, _/_, _/_, _/_
Clue 1: _/Phys, Emmy/_, _/_, _/_
Clue 2: _/Phys, Emmy/Math, _/_, _/_
Clue 3: _/Phys, Emmy/Math, Lynn/_, _/CS
Clue 2: _/Phys, Emmy/Bio, _/_, _/_
Clue 3: _/Phys, Emmy/Bio, Lynn/_, _/CS
Clue 1: _/Phys, _/_, Emmy/_, _/_
Clue 2: _/Phys, _/_, Emmy/Math, _/_
Clue 3: Lynn/Phys, _/CS, Emmy/Math, _/_
Clue 2: _/Phys, _/_, Emmy/Bio, _/_
Clue 3: Lynn/Phys, _/CS, Emmy/Bio, _/_
Clue 1: _/Phys, _/_, _/_, Emmy/_
Clue 2: _/Phys, _/_, _/_, Emmy/Math
Clue 3: Lynn/Phys, _/CS, _/_, Emmy/Math
Clue 4: Lynn/Phys, _/CS, Marie/_, Emmy/Math
Clue 3: _/Phys, Lynn/_, _/CS, Emmy/Math
Clue 2: _/Phys, _/_, _/_, Emmy/Bio
Clue 3: Lynn/Phys, _/CS, _/_, Emmy/Bio
Clue 4: Lynn/Phys, _/CS, Marie/_, Emmy/Bio
Clue 3: _/Phys, Lynn/_, _/CS, Emmy/Bio
Clue 1: _/_, _/Phys, Emmy/_, _/_
Clue 2: _/_, _/Phys, Emmy/Math, _/_
Clue 2: _/_, _/Phys, Emmy/Bio, _/_
Clue 1: _/_, _/Phys, _/_, Emmy/_
Clue 2: _/_, _/Phys, _/_, Emmy/Math
Clue 3: _/_, Lynn/Phys, _/CS, Emmy/Math
Clue 2: _/_, _/Phys, _/_, Emmy/Bio
Clue 3: _/_, Lynn/Phys, _/CS, Emmy/Bio
Clue 1: _/_, _/_, _/Phys, Emmy/_
Clue 2: _/_, _/_, _/Phys, Emmy/Math
Clue 3: Lynn/_, _/CS, _/Phys, Emmy/Math
Clue 4: Lynn/_, _/CS, Marie/Phys, Emmy/Math
Clue 5: Lynn/Bio, Ada/CS, Marie/Phys, Emmy/Math

After 33 rule applications,
Solution:
  1. Lynn/Bio ($25,000 scholarship)
  2. Ada/CS ($30,000 scholarship)
  3. Marie/Phys ($35,000 scholarship)
  4. Emmy/Math ($40,000 scholarship)

More? (y, or n)? > y
Clue 2: _/_, _/_, _/Phys, Emmy/Bio
Clue 3: Lynn/_, _/CS, _/Phys, Emmy/Bio
Clue 4: Lynn/_, _/CS, Marie/Phys, Emmy/Bio
```

Listing 16. *Trace of the scholarship problem*

[10] JC Dageförde and H Kuchen. "A compiler and virtual machine for constraint-logic obj-oriented programming with Muli". In: *J of Comp Lang* 53 (2019), pp. 63–78.

[11] JC Dageförde and H Kuchen. "A constraint-logic object-oriented language". In: *Proc 33rd Ann ACM Symp on Applied Computing*. 2018, pp. 1185–1194.

[12] Christophe Delord. *PyLog*. http://cdsoft.fr/pylog/index.html. 2009.

[13] Didier Dubois and Henri Prade. "Towards a reconciliation between reasoning and learning-A position paper". In: *International Conference on Scalable Uncertainty Management*. Springer. 2019, pp. 153–168.

[14] Rainhard Findling. URL: https://geekoverdose.wordpress.com/2015/10/31/solving-logic-puzzles-in-prolog-puzzle-1-of-3/.

[15] B Frederiksen. *Pyke*. http://pyke.sourceforge.net/. 2011.

[16] Eugene C Freuder. "In pursuit of the holy grail". In: *Constraints* 2.1 (1997), pp. 57–61.

[17] Marta Garnelo and Murray Shanahan. "Reconciling deep learning with symbolic artificial intelligence: representing objects and relations". In: *Current Opinion in Behavioral Sciences* 29 (2019), pp. 17–23.

[18] Robert Geirhos et al. "ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness". In: *arXiv preprint arXiv:1811.12231* (2018).

[19] *Generator (comp prog)*. https://en.wikipedia.org/wiki/Generator_(computer_programming).

[20] J Gibbons and N Wu. "Folding domain-specific languages: deep and shallow embeddings (functional pearl)". In: *Proc 19th ACM SIGPLAN Intl Conf on Functional programming*. 2014, pp. 339–347.

[21] Google. *OR Tools*. https://developers.google.com/optimization/.

[22] C.A.R. Hoare and H. Jifeng. *Unifying theories of programming*. Vol. 14. Prentice Hall Englewood Cliffs, 1998.

[23] K Kuchcinski and R Szymanek. “Jacop-java constraint programming solver”. In: *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th Intl Conf on Principles and Practice of Constraint Programming*. 2013.

[24] K Lacker. *Conversation with GPT-3*. 2020. URL: https://lacker.io/ai/2020/07/23/conversation-with-gpt3.html.

[25] Nikola M. *Python Prolog Interpreter*. https://github.com/photonlines/Python-Prolog-Interpreter. 2019.

[26] John McCarthy et al. “A Proposal for the Dartmouth Summer Research Project on Artificial Intelligence, Sugust 31, 1955”. In: *reprinted in AI magazine* 27.4 (2006), p. 12. URL: http://www-formal.stanford.edu/jmc/history/dartmouth/dartmouth.html.

[27] Chris Meyers. *Prolog in Python*. http://www.openbookproject.net/py4fun/prolog/prolog1.html. 2015.

[28] Andrew Ng. *AI is the new electricity*. O’Reilly, 2018.

[29] *Oscar/CBLS*. URL: https://oscarlib.bitbucket.io/.

[30] Ian Piumarta. *Lecture notes and slides from weeks 5-7 of a course on programming paradigms*. http://www.ritsumei.ac.jp/~piumarta/pl/index.html. 2017.

[31] Charles Prud’homme and Jean-Guillaume Fages. *Choco Solver*. https://www.cril.univ-artois.fr/CompetitionXCSP17/files/choco.pdf. 2019.

[32] Matthew Rocklin. *Kanren: Logic Programming in Python*. https://github.com/logpy/logpy/. 2019.

[33] Stuart J Russell and Peter Norvig. *Artificial Intelligence-A Modern Approach, Third International Edition*. 2010.

[34] Claudio Santini. *Pampy: The Pattern Matching for Python you always dreamed of*. https://github.com/santinic/pampy. 2018.

[35] Silvija Seres and Michael Spivey. “Embedding Prolog in Haskell”. In: *Proceedings of the 1999 Haskell Workshop*. 1999, pp. 23–38.

[36] Ehud Y Shapiro. “The fifth generation project—a trip report”. In: *Communications of the ACM* 26.9 (1983), pp. 637–641.

[37] David Silver et al. “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419 (2018), pp. 1140–1144.

[38] HA Simon. *Models of man; soc and rat*. Wiley, 1957.

[39] *SWI prolog*. URL: https://www.swi-prolog.org/pldoc/doc_for?object=manual.

[40] Jeff Thompson. *Yield Prolog*. http://yieldprolog.sourceforge.net/. 2017.

[41] Mark Wallace. “Problem Modelling in MiniZinc”. In: *Blding Dec Sup Sys*. Springer, 2020, pp. 37–47.

[42] Hong Xu, Sven Koenig, and TK Satish Kumar. “Towards effective deep learning for constraint satisfaction problems”. In: *Intl Conf on Principles and Practice of Constraint Programming*. Springer. 2018, pp. 588–597.

[43] *Yuck*. URL: https://github.com/informarte/yuck.

[44] *Zebra Puzzle*. https://en.wikipedia.org/wiki/Zebra_Puzzle.

[45] Neng-Fa Zhou, Håkan Kjellerstrand, and J Fruhman. *Constraint solving and planning with Picat*. Springer, 2015.