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

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]

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

---

1 Acknowledgement: We would like to express our deeply felt gratefulness to Professor Raj Panula and Mohammad Pourhomayoun for their comprehensive and detailed peer-reviewing of this document.
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 Frueeder 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/CBLIS[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) tvmsl_dfs (Listing 2), the solver. (Please pardon our Python style deficiencies. The column width and page limit compelled compromises.)

```python
unassigned = ""
remaining_indices = uninstantiated_indices(transversal)

Listing 1. uninstantiated_indices

```def tvmsl_dfs(set, tvmsl):
  remaining_indices = uninstantiated_indices(tvmsl)
  if not remaining_indices:
    return tvmsl
  nxt_index = min(remaining_indices)
  for elmt in sets[nxt_index]:
    if elmt not in tvmsl:
      new_tvmsl = tvmsl[:nxt_index] \n      + \{(elmt,)\}
      if full_tvmsl is not None: return full_tvmsl
      full_tvmsl = tvmsl_dfs(sets, new_tvmsl)
  return tvmsl

Listing 2. tvmsl_dfs
```

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

- The function tvmsl_dfs takes two parameters:
  1) sets: a list of sets
  2) tvmsl: 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 tvmsl. Initially this will be all of them. Since tvmsl_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_index to the first undefined index position.

- line 6. Begin a loop that looks at the elements of sets[nxt_index], the set at position nxt_index. We want an element from that set to represent it in the transversal.

- line 7. If the currently selected elmt of sets[nxt_index] is not already in tvmsl:
  1) lines 8-10. Put elmt at position nxt_index.
  2) line 11. Call tvmsl_dfs recursively to complete the transversal, passing new_tvmsl, the extended tvmsl. Assign the returned result to full_tvmsl.

- line 12. If full_tvmsl is not None, we have found a transversal. Return it to the caller. If full_tvmsl is None, the elmt we selected from sets[nxt_index] did not lead to a complete transversal. Return to line 6 to select another element from sets[nxt_index].

This is standard depth first search. tvmsl_dfs will either find the first transversal, if there are any, or return None. Here’s a trace of the recursive calls:

```
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (_ _, _) |
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (1, _) |
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (1, 2, _) |
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (1, 2, 4) |
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (1, 2, 4) |
| sets: ([1,2,3], [1,2,4], [1]) | tvmsl: (1, 2, 4) |
```

List 3. tvmsl_dfs trace

- line 1. Initially (and on each call) the sets are $\{1, 2, 3\}, \{1, 2, 4\}, \{1\}$

Initially tvmsl is completely undefined: (_ _, _)

- line 2. 1 is selected as the first element of tvmsl.

- line 3. 2 is selected as the second element.

- line 4. But now we are stuck. Since 1 is already in tvmsl, 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 back 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 tvmsl, 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 backtracking 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.

```python
new_sets = {set - {elm} for set in sets}
```

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

2) Before line 5, insert

```python
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.

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

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

In the original code (Listing 1), replace line 5 with

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

The resulting trace (Listing 3) 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}], tnvl: (1,1,1)
2) sets: [{1,2,3}, {1,2,4}, {1}], tnvl: (1,1,1)
3) sets: [{1,2,3}, {1,2,4}, {1}], tnvl: (2,1,1)
4) sets: [{1,2,3}, {1,2,4}, {1}], tnvl: (2,4,1)

Listing 3. tnvl_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.

1) sets: [{1,2,3}, {1,2,4}, {1}], tnvl: (1,1,1)
2) sets: [{1,2,3}, {2,4}, {}], tnvl: (1,1,1)
3) sets: [{3}, {4}, {}], tnvl: (2,1,1)
4) sets: [{3}, {}], tnvl: (2,4,1)

Listing 6. tnvl_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.
def tvsli_dfs_gen (sets, tvnsli):
    remaining_indices = uninstantiated_indices(tvnsli)
    if not remaining_indices:
        yield tvnsli
    else:
        if any(not set[i] for i in remaining_indices):
            return None
        ntxt_idx = min(remaining_indices,
                        key=lambda idx: len(sets[idx]))
        for elmt in sets[ntxt_idx]:
            if elmt not in tvnsli:
                new_tvnsli = tvnsli[:ntxt_idx] \
                + (elmt,) \
                + tvnsli[ntxt_idx+1:]
                new_sets = [set - {elmt} for set in sets]
                for full_tvnsli in tvsli_dfs_gen(new_sets,
                                                new_tvnsli):
                    yield full_tvnsli
        for tvnsli in tvsli_dfs_gen(sets, ('_-', 'i', 'i', 'i')):
            print('=>', tvnsli)

Listing 7. tvsli_dfs_gen

- The newly added else on line 5 is necessary. Previously, if there were no remaining_indices, we returned tvsli. That was the end of execution for this recursive call. But if we yield instead of return, when tvsli_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 tvsli_dfs_gen recursively and ask for all transversals that can be constructed from the current state. Each one is then yielded. No need to exclude None. tvsli_dfs_gen will yield only complete transversals.

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

yield from tvsli_dfs_gen(new_sets, new_tvnsli)

Let’s use tvsli_dfs_gen (Listing 7) to find a transversal whose elements sum to, say, 6.

p = 6
for tvnsli in tvsli_dfs_gen(sets, ('_-', 'i', 'i', 'i')):
    sum_string = '+' + join(str(i) for i in tvnsli)
    equals = '==' if sum(tvnsli) == p else '!='
    out.println(f'{sum_string} {equals} {n} is
    if sum(tvnsli) == p:
        break

Listing 9. running tvsli_dfs_gen

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.

A = Var()
for  in unify(A, 'abc'):
    print(A) # =>
    if A == 'abc'
        print(A) # => abc
        A = 'def'
        print(A) # =>
        print(A) # =>

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 Vars 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 Vars unifies all the Vars 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 yields, it is said to succeed; the for-loop body runs. When a generator returns, 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  \{print(A, B, C, D) => 2_2_3_3\}
5 for in unify(A, C):
6  \{print(A, B, C, D) => abc abc abc 3_3\}
7 for in unify(A, D):
8  \{print(A, B, C, D) => 2_2_3_3\}
9 print(A, B, C, D) => 2_2_3_3
10 print(A, B, C, D) => 2_2_3_3
11 print(A, B, C, D) => 2_2_3_3
12 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.

\[
A \rightarrow B \\
D \rightarrow C
\]

(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'.

\[
A \rightarrow B \rightarrow 'abc' \\
D \rightarrow C
\]

(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).

\[
A \rightarrow B \rightarrow 'abc' \\
D \rightarrow C
\]

(3)

C. A logic-variable version of `tmvl_dfs_gen`

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

First, an adapted `uninstant_indices` returns the indices of the uninstantiated Vars in `tmvl`.

```
1 def uninstant_indices (tmvl):
2     return [index for index in range(len(tmvl))
3             if not tmvl[index].is_instantiated()]
```

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 `tmvl_dfs_gen` (Listing 7) so that the program will fit the width of a column.)

• line 6. The parameter `sets` is a list of `PySets`. 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 `tmvl` elements.
• line 15. `T_Var` is the element at the `nxt_indx` position of `tmvl`. Since `nxt_indx` was selected from the uninstantiated variables, `T_Var` is an uninstantiated `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. `PySets` 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.

```
1 sets: [[1, 2, 3], [2, 4]], tmvl|: (1, 2, 3)
2 sets: [[2, 3], [2, 4]], tmvl: (1, 2, 3)
3 sets: [[3], [4]], tmvl: (2, 4, 1)
4 => (2, 4, 1)
5 sets: [[1], [2, 4]], tmvl: (3, 2, 1)
6 sets: [[], [1]], tmvl: (3, 2, 1)
7 => (3, 2, 1)
8 sets: [[], [2]], tmvl: (3, 4, 1)
9 => (3, 4, 1)
```

The following logic variable version of Listing 9 will run `tmvl_dfs_gen_lv` and produce the same result.
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 Stndt. Each instance has two fields: name and major. (We do not keep track of the students’ scholarships!) For example, a Stndt object that represents Ada studying Phys is constructed like this 

```python
Stndt(name='Ada', major='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: Stndt(major='Bio').

Our world consists of a list of Stndt 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 */Physics*. 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]


REFERENCES

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. """
    yield from is_subseq([Stdnt(name='Emmy'), Stdnt(major=Maj)], Stdnts)
    # Create Major as a local logic variable.
    Major = Var()
    for i in member(Stdnt(name='Emmy'), major=Maj, Stdnts):
        yield from member(Maj, 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 clue 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
Clue 4: /Phys, Emmy, Math, Lynn, CS
Clue 5: /Lynn, Phys
Clue 6: /Lynn, Phys, CS
Clue 7: /Lynn, Phys, CS, Emmy
Clue 8: /Lynn, Phys, CS, Emmy, Math
Clue 9: /Lynn, Phys, CS, Emmy, Math, Lynn
Clue 10: /Lynn, Phys, CS, Emmy, Math, Lynn, CS
Clue 11: /Lynn, Phys, CS, Emmy, Math, Lynn

Listing 16. Trace of the scholarship problem


ISSN: 1690-4524
SYSTEMICS, CYBERNETICS AND INFORMATICS
VOLUME 19 - NUMBER 3 - YEAR 2021
21
[29] Oscar/CBLS. URL: https://oscarlib.bitbucket.io/.