COMP24011 Lab2: Constraint Satisfaction

Ian Pratt-Hartmann and Francisco Lobo

Academic session: 2024-25

Introduction

This laboratory features the constraint satisfaction solver implemented in the Python `constraint` module, originally written by Gustavo Niemeyer. It consists of two exercises.

- In the first, you will solve the 'logic puzzle' encountered in the introductory lecture to this course (the one about the travellers arriving at an airport). The purpose of this exercise is to practice turning a problem expressed informally in English into a constraint satisfaction problem (CSP). If you are successful, you will see that the Python constraint satisfaction module works very well.
- In the second, we apply the same constraint satisfaction module to a more abstract problem in recreational mathematics. The problem in question is that of generating pandiagonal magic squares. (We explain what these are below.) The purpose of this exercise is to illustrate some of the limitations of CSP solvers — especially when the problem instances they are given grow in size.

As a first step, you will need to install the Python `constraint` module in your environment by issuing the following command

```
$ pip install python-constraint2
```

Notice that the newest version is registered as `python-constraint2` on PyPI because the module has a new maintainer. The source code is available on GitHub, where you'll also find the module's documentation with *relevant examples* for this laboratory. You'll want to broadly understand those examples to be ready to solve the exercises below.

Once you have your Python environment set up, you'll be able to use the sample code we provide in the `lab2` branch of your `COMP24011_2024` GitLab repo. You should test it works by running

```
$ python3 ./constraintsLab.py
Usage: ./constraintsLab.py <FUNCTION> <ARG>...
```

You can then use additional command line arguments to test and debug the functions that you need to implement in this laboratory.

The logic puzzle

Let us recall the logic puzzle encountered in the introductory lecture. Four travellers — Claude, Olga, Pablo and Scott — are returning from four destinations — Peru, Romania, Taiwan and Yemen — departing at four different times — 2:30, 3:30, 4:30 and 5:30 (one hopes: in the afternoon).

No two travellers return from the same destination, and no two depart at the same time. In addition to these constraints, we are given the following *problem axioms*:

1. Olga is leaving 2 hours earlier than the traveller flying from Yemen.
2. Claude is either the person leaving at 2:30 pm or the traveller leaving at 3:30 pm.
3. The person leaving at 2:30 pm is flying from Peru.
4. The person flying from Yemen is leaving earlier than the person flying from Taiwan.
5. The four travellers are Pablo, the traveller flying from Yemen, the person leaving at 2:30 pm and the person leaving at 3:30 pm.

Every solution to this CSP consists of a time and a destination for each of the four travellers. We need to set this up using the Python `constraint` module. One way is to declare a time and a destination variable for each traveller, and register the possible values these may take.

```
import constraint
problem = constraint.Problem()

people = ['claude', 'olga', 'pablo', 'scott']
times = ['2:30', '3:30', '4:30', '5:30']
destinations = ['peru', 'romania', 'taiwan', 'yemen']

t_variables= list(map(( lambda x: 't_'+x ), people))
d_variables= list(map(( lambda x: 'd_'+x ), people))

problem.addVariables(t_variables, times)
problem.addVariables(d_variables, destinations)
```

Thus, time variables get names like `t_olga` and destination variables have names like `d_olga`. You *must* also use this convention in your implementation.

Now let's add the basic premises of the puzzle.

```
# no two travellers depart at the same time
problem.addConstraint(constraint.AllDifferentConstraint(), t_variables)
# no two travellers return from the same destination
problem.addConstraint(constraint.AllDifferentConstraint(), d_variables)
```

Note the use of the shortcut `constraint.AllDifferentConstraint()` to say that certain sets of variables have to have different values. (It would be annoying to have to write this out by hand.) The Python `constraint` module provides a number of these which you'll have to investigate and use in this laboratory. We can get the list of solutions for the CSP that we've define as follows.

```
print( problem.getSolutions() )
```

Without implementing any of the other constraints of the puzzle, you will notice there are rather a lot of solutions (576). Let's add the first special constraint of the puzzle.

```
# Olga is leaving 2 hours before the traveller from Yemen
for person in people:
  problem.addConstraint((
    lambda x,y,z:
         (y != 'yemen')
      or ((x == '4:30') and (z == '2:30'))
      or ((x == '5:30') and (z == '3:30'))
    ), ['t_'+person, 'd_'+person, 't_olga'])

print( problem.getSolutions() )
```

You'll find that there are now just 72 solutions to the CSP. Note the use of a lambda expression here to express a constraint. This is a function of three variables, `x`, `y` and `z`, which returns a Boolean value depending on the values of those variables. (You should be able to read the body of the function.) When, in the for-loop, `person` evaluates to — say — `claude`, the `addConstraint()` method applies the constraint in question to the triple of variables `['t_claude', 'd_claude', 't_olga']`. Once all these constraints are in place, solutions of the CSP must assign variables only values to which the anonymous function returns `True` for lists `[x, y, 't_olga']`, with `x` the time and `y` the destination of a traveller.

The other special constraints are handled similarly. Probably the most puzzling is the fourth. As a hint, you might read it as a collection of statements to the effect that certain descriptions do not co-refer: Pablo is not flying from Yemen and is leaving at neither 2:30 nor 3:30; and whoever *is* flying from Yemen is likewise leaving at neither 2:30 nor 3:30.

Once you implement all the special constraints of the puzzle, you will see that there is in fact just one solution:

```
$ ./constraintsLab.py Travellers "[1,2,3,4,5]" "[]"
debug run: Travellers([1,2,3,4,5],[])
ret value: [{'t_olga': '2:30', 'd_olga': 'peru',
  't_claude': '3:30', 'd_claude': 'romania',
  'd_pablo': 'taiwan', 'd_scott': 'yemen',
  't_scott': '4:30', 't_pablo': '5:30'}]
ret count: 1
```

The order of printing is a bit *random*, as Python dictionaries do not print in a particular order. (Do not worry about this.) If you remove the first constraint (the one about Olga leaving two hours before the traveller from Yemen), you will again find that there are other solutions...

Task 1: Write a function `Travellers(axiomList, extraPairs)` which takes a non-empty list of problem axioms and a (possibly empty) list of extra pairs. Your function needs to set up a logic puzzle as above for the problem axioms given in the argument `axiomList`, which are numbered 1 to 5 as defined earlier. It should then add the extra constraints passed as the argument `extraPairs`. Each extra pair consists of either a traveller and a time, e.g. `('olga','2:30')`, or a traveller and a destination, e.g. `('olga','peru')`. These are interpreted as the constraints that Olga travels at 2:30 and Olga travels from Peru, respectively. The function must return the list of solutions that satisfy the resulting CSP.

For instance, the call to `Travellers([1,2,3,4,5], [])` should return a list with the solution to the original puzzle given above. For the call `Travellers([2,3,4,5], [('olga','2:30')])` you should obtain the same result. On the other hand, `Travellers([2,3,4], [('olga','peru')])` will have 6 solutions. There will also be puzzle configurations that have no solutions, for example

```
$ ./constraintsLab.py Travellers "[4,5]" "[('olga','5:30')]"
debug run: Travellers([4,5],[('olga','5:30')])
ret value: []
ret count: 0
```

Pandiagonal magic squares

Let n be a positive integer. A *magic square* of size n is an $n \times n$ list whose entries are the integers 1 to n^2 arranged in such a way that the sum of each row, each column and both diagonals is the same. Figure 1(a) shows an example with $n = 4$, in which the entries in every row, every column and the two diagonals add up to 34.

11	2	5	16
10	8	3	13
7	9	14	4
6	15	12	1

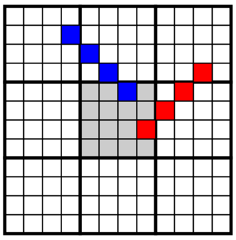
(a) A magic square

13	12	7	2
8	1	14	11
10	15	4	5
3	6	9	16

(b) A pandiagonal magic square

Figure 1: Magic squares of size 4

Figure 2(a) shows the same 4×4 square repeated in a 3×3 super-grid. Consider the central 4×4 square (shaded), and starting at any boundary cell, suppose we proceed diagonally (in any of the four directions) for a distance of 4 squares: two such diagonal paths are shown (in blue and red). Since the central square is repeated, these two diagonal paths are, in effect *broken diagonals* of the original square, as shown in Figure 2(b). Observe that the two main diagonals are just a special case of broken diagonals. A simple check shows that the *broken* diagonals of the square in Figure 1(a) do not all add up to 34.



(a) Two diagonals

11	2	5	16
10	8	3	13
7	9	14	4
6	15	12	1

(b) Broken diagonals

Figure 2: Magic squares and broken diagonals

A *pandiagonal magic square* of size n is an $n \times n$ list whose entries are the integers 1 to n^2 arranged in such a way that the sum of each row, each column and each broken diagonal is the same. Somewhat surprisingly, pandiagonal magic squares exist: Figure 1(b) shows a pandiagonal magic square for $n = 4$.

Task 2: The sum of each row, column and broken diagonal in a pandiagonal magic square of size n must be the same. Write a Python function `CommonSum(`n`)` to compute this common sum. Your code must accept any natural number n as argument, regardless of whether or not a pandiagonal magic square of that size exists.

The following Python code is taken from the documentation of the `constraint` module. It finds all 4×4 magic squares.

```
from constraint import *
problem = Problem()
problem.addVariables(range(0, 16), range(1, 16 + 1))
problem.addConstraint(AllDifferentConstraint(), range(0, 16))
problem.addConstraint(ExactSumConstraint(34), [0, 5, 10, 15])
problem.addConstraint(ExactSumConstraint(34), [3, 6, 9, 12])
for row in range(4):
  problem.addConstraint(ExactSumConstraint(34),
                        [row * 4 + i for i in range(4)])
for col in range(4):
  problem.addConstraint(ExactSumConstraint(34),
                        [col + 4 * i for i in range(4)])
solutions = problem.getSolutions()
print(solutions)
```

This calculation will take some time, but should complete in less than 5 minutes.

For this exercise you'll need to generalise the above example to build a framework for finding magic squares of a chosen size n. Note that we require n^2 variables (numbered 0 to $n - 1$), each of which takes a value in the range 1 to n^2 (inclusive). For definiteness we imagine the variables numbered row by row — i.e. $0, \ldots, n - 1$ along the top row, $n, \ldots, 2n - 1$ along the next row down, and so on, with $n^2 - n, \ldots, n^2 - 1$ on the bottom row. We then need to say that the variables all take different values. Finally, we need constraints stating that all rows, all columns and the two diagonals have to add up the same number, the one given by `CommonSum(`n`)`. Indeed, this is what the above example does for $n = 4$.

To then find pandiagonal magic squares, your challenge is to write constraints saying that the broken diagonals have to add up to the same number. This will require some thought. Note that the positions in the magic square of size n are assumed to be numbered from 0 to $n^2 - 1$, row by row, and thus each broken diagonal can be represented by a subset of $\{0, \dots, n^2 - 1\}$ with its n positions.

Task 3: Write a function `BrokenDiags(`n`)` which for a given integer n generates a list of all the broken diagonals in a magic square of size n. Each broken diagonal will be a list of length n with the magic square positions lying along it. Your code must accept any natural number n as argument, regardless of whether or not a pandiagonal magic square of that size exists. Your return value should not contain repeated broken diagonals.

For example, for magic squares of size $n = 4$ you should find

```
$ ./constraintsLab.py BrokenDiags 4
debug run: BrokenDiags(4)
ret value: [ [0, 5, 10, 15], [0, 7, 10, 13], [1, 4, 11, 14], [1, 6, 11, 12],
             [2, 5, 8, 15], [2, 7, 8, 13], [3, 4, 9, 14], [3, 6, 9, 12] ]
ret count: 8
```

The order of printing does not matter, as long as all of the 8 broken diagonals are included. In this example, the list `[0,5,10,15]` is one of main diagonals of the 4×4 square and `[11,4,1,14]` is the red broken diagonal of Figure 2(b).

Completing tasks 2 and 3 provides some of the components necessary to generate pandiagonal magic squares of a certain size. As noted above, finding all magic squares of a certain size is time consuming, so to test your code in reasonable time we need to add some extra constraints...

Task 4: Write a function `MSquares(`n`, axiomList, extraPairs)` which takes an integer n, a non-empty list of magic square axioms, and a (possibly empty) list of extra pairs. The argument `axiomList` will contain values between 1 and 4 which must be interpreted as the following axioms.

1. The sum of each row is the common sum for a magic square of size n.
2. The sum of each column is the common sum for a magic square of size n.
3. The sum of each main diagonal is the common sum for a magic square of size n.
4. The sum of each broken diagonal is the common sum for a magic square of size n.

Your function needs to return a list of number squares with entries 1 to n^2 that satisfy the given subset of magic square axioms. These must also satisfy the extra constraints passed as the argument `extraPairs`. Each extra pair `(`v`, `k`)` asserts that the position v of the square is filled with the integer k. It will always happens that $0 \le v < n^2$ and $1 \le k \le n^2$.

Remember that the positions in the magic square of size n are assumed to be numbered from 0 to $n^2 - 1$, row by row. When setting up task 4 as a CSP these positions become the variables for the problem. You *must* also use this convention in your implementation. For example, you should find

```
$ ./constraintsLab.py MSquares 4 "[1,2,3]" "[(0,13),(1,12),(2,7)]"
debug run: MSquares(4,[1,2,3],[(0,13),(1,12),(2,7)])
ret value: [ {0: 13, 3: 2, 5: 3, 6: 16, 9: 5, 12: 11, 10: 10, 15: 8,
              1: 12, 2: 7, 14: 1, 4: 6, 8: 4, 7: 9, 13: 14, 11: 15},
             {0: 13, 3: 2, 5: 6, 10: 4, 15: 11, 6: 9, 9: 15, 12: 8,
              1: 12, 2: 7, 13: 1, 4: 3, 8: 10, 11: 5, 14: 14, 7: 16},
             {0: 13, 3: 2, 5: 8, 10: 10, 15: 3, 6: 11, 9: 5, 12: 16,
              1: 12, 2: 7, 4: 1, 8: 4, 14: 6, 13: 9, 7: 14, 11: 15},
             {0: 13, 3: 2, 5: 1, 6: 14, 9: 15, 12: 3, 10: 4, 15: 16,
              1: 12, 2: 7, 13: 6, 14: 9, 4: 8, 8: 10, 11: 5, 7: 11} ]
ret count: 4
```

We've formatted the output so it is easier to read, but essentially these are the four magic squares of size 4 which include `{0: 13, 1: 12, 2: 7}`. Only two of these are pandiagonal magic squares, so you should verify that

```
$ ./constraintsLab.py MSquares 4 "[1,2,4]" "[(0,13),(1,12),(2,7)]"
debug run: MSquares(4,[1,2,4],[(0,13),(1,12),(2,7)])
ret value: [ {0: 13, 1: 12, 2: 7, 3: 2, 4: 8, 8: 10, 12: 3, 11: 5,
              14: 9, 6: 14, 10: 4, 5: 1, 7: 11, 13: 6, 9: 15, 15: 16},
             {0: 13, 1: 12, 2: 7, 3: 2, 4: 3, 5: 6, 10: 4, 15: 11,
              8: 10, 12: 8, 6: 9, 11: 5, 14: 14, 13: 1, 9: 15, 7: 16} ]
ret count: 2
```

The first one of them was already given in Figure 1(b). As explained before, your Python output is likely to show the squares in a different order.

For squares of size 3, you should confirm that the call `MSquares(3,[1,2,4],[])` — i.e. with an empty list of fixed positions — returns an empty list because there are no 3×3 pandiagonal magic squares. The call `MSquares(3,[1,2,3],[])` should obtain the 8 magic squares of size 3. On the other hand, `MSquares(3,[1,2],[])` will have 72 solutions.

When testing your code for task 4 do not worry if the program is very slow for $n > 4$ and a small (or especially empty) list of extra constraints. We will **only** test your implementation on examples that should run in reasonable time.

Submission

Please follow the `README.md` instructions in your `COMP24011_2024` GitLab repo. Refresh the files of your `lab2` branch and develop your solution to the lab exercise. The solution consists of a single file called `constraintsLab.py` which must be committed to your GitLab repo and tagged as `lab2_sol`. The `README.md` instructions that accompany the lab files include the `git` commands necessary to commit, tag, and then push **both** the commit and the tag to your `COMP24011_2024` GitLab repo. Further instructions on coursework submission using GitLab can be found in Appendix L of the CS Handbook, including how to change a `git` tag after pushing it.

The deadline for submission is **09:00 on Monday 4th November**. In addition, no work will be considered for assessment and/or feedback if submitted more that **1 week** after the deadline. (Of course, these rules will be subject to any mitigating procedures that you have in place.)

The lab exercise will be **auto-marked** offline. The automarker program will download your submission from GitLab and test it against our reference implementation. For each task the return value of your function will be checked on a random set of valid arguments. A time limit of 10 seconds will be imposed on every function call, and exceeding this time limit will count as a runtime error.

A total of 20 marks is available in this exercise. The marking scheme is as follows.

Task 1 There are 10 marks for this laboratory task. You obtain the first mark if all tests complete without runtime errors. The proportion of tests with fully correct return values will determine the remaining 9 marks.

Task 2 There are 2 marks for this laboratory task. The proportion of tests with fully correct return values will determine your score (out of 2).

Task 3 There are 4 marks for this laboratory task. The proportion of tests with fully correct return values will determine your score (out of 4).

Task 4 There are 4 marks for this laboratory task. The proportion of tests with fully correct return values will determine your score (out of 4).

Important Clarifications

- It will be very difficult for you to circumvent time limits during testing. If you try to do this, the most likely outcome is that the automarker will fail to receive return values from your implementation, which will have the same effect as not completing the call. In any case, an additional time limit of 240 seconds for all tests of each task will be enforced; this is sufficient to allow your code 10 seconds for each function call.
- This lab exercise is fully auto-marked. If you submit code which the Python interpreter does not accept, you will score 0 marks. The Python setup of the automarker is the same as the one on the department's lab machines, but only a **minimal** set of Python modules are available. If you choose to add `import` statements to the sample code, it is **your responsibility** to ensure these are part of the default Python package available on the lab machines.
- It doesn't matter how you organise your `lab2` branch, but you should avoid having multiple files with the same name. The automarker will sort your directories alphabetically (more specifically, in ASCII ascending order) and find submission files using breadth-first search. It will mark the first `constraintsLab.py` file it finds and ignore all others.
- Every file in your submission should only contain printable ASCII characters. If you include other Unicode characters, for example by copying and then pasting code from the PDF of the lab manuals, then the automarker is likely to reject your files.