Open in Colab

Handcrafting Instance Generators in Essence

Original Handcrafting Instance Generators in Essence by Joan Espasa Arxer and Christopher Stone. Adapted by Alex Gallagher.

In modelling it is common to create an abstract model that expects some input parameters (Also known as “instances”) which are required to run and test the model. In this tutorial we demonstrate how to use ESSENCE to handcraft a generator of instances that can be used to produce input parameters for a specific model.

[10]:
```
!source <(curl -s https://raw.githubusercontent.com/conjure-cp/conjure-notebook/v0.0.9/scripts/install-colab.sh)
%load_ext conjure
```

```
Installing Conjure version v2.5.1 and Conjure Notebook version v0.0.9...
Conjure is already installed.
Conjure notebook is already installed.
Conjure: The Automated Constraint Modelling Tool
Release version 2.5.1
Repository version a9cbc2e (2023-11-07 23:44:00 +0000)
The conjure extension is already loaded. To reload it, use:
  %reload_ext conjure
```

Instances for the Knapsack problem

This model from the Knapsack Problem has 4 different “given” statements :

- number_items: an integer for number of items
- weight: a functions that associates an integer(weight) to each item
- gain: a function that associates an integer(gain) to each item

- capacity: an integer that defines the capacity of the knapsack

The first parameter is fairly simple and we can even write this parameter with some value by hand as seen below.

[11]:
```
%%conjure
letting number_items be 20
```

```
{}
```

The remaining 3 parameters are more complex and labourious to be defined (too much work to be done by hand!) so we are going to write an ESSENCE specification that will create them for us. The fundamental starting step is writing find statements for each variable we wish to generate and ensure that the names of the variable (identifiers) are left unchanged. We can do so by writing:

[12]:
```
%%conjure+
letting items be domain int(1..number_items)
find weight: function (total) items --> int(1..1000)
find gain: function (total) items --> int(1..1000)
find capacity: int(1..5000)
```


```
{"capacity": 1, "gain": {"1": 1, "10": 1, "11": 1, "12": 1, "13": 1, "14": 1, "15": 1, "16": 1,
"17": 1, "18": 1, "19": 1, "2": 1, "20": 1, "3": 1, "4": 1, "5": 1, "6": 1, "7": 1, "8": 1, "9":
1}, "weight": {"1": 1, "10": 1, "11": 1, "12": 1, "13": 1, "14": 1, "15": 1, "16": 1, "17": 1,
"18": 1, "19": 1, "2": 1, "20": 1, "3": 1, "4": 1, "5": 1, "6": 1, "7": 1, "8": 1, "9": 1}}
```


Solving the above model (by running the cell above) will create a set of parameters for our knapsack model. However, these instances are not interesting enough yet.

We can make our instances more interesting by adding constraints into our generator's model. The first thing we notice is that all values assigned are identical, a bit TOO symmetrical for our taste. One simple solution to this issue is ensuring that all weights and gains assignments are associated with distinct values. This can be done by imposing injectivity as a property of the function.

[13]:
```
%%conjure
letting number_items be 20
letting items be domain int(1..number_items)
find weight: function (total, injective) items --> int(1..1000)
find gain: function (total, injective) items --> int(1..1000)
find capacity: int(1..5000)
```


```
{"capacity": 1, "gain": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15, "16":
16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7, "8":
8, "9": 9}, "weight": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15, "16":
16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7, "8":
8, "9": 9}}
```


Running this gives us a slighly more interesting parameters set but it is not there yet The specific order that appears in the results is solver dependent. The default solver used by conjure is Minion and we can use an optional flag to have the variables assigned in a random order. This can be done with this command:

```
--solver-options=-randomiseorder
```

Alternatively one can use another solver that uses randomness by default

[14]:
```
%%conjure+ --solver=minion --solver-options='-randomiseorder'
```


```
{"capacity": 763, "gain": {"1": 509, "10": 113, "11": 230, "12": 167, "13": 659, "14": 489, "15":
269, "16": 235, "17": 385, "18": 922, "19": 701, "2": 915, "20": 909, "3": 502, "4": 229, "5":
447, "6": 231, "7": 472, "8": 929, "9": 440}, "weight": {"1": 830, "10": 512, "11": 478, "12":
707, "13": 2, "14": 960, "15": 810, "16": 684, "17": 9, "18": 10, "19": 751, "2": 200, "20": 516,
"3": 114, "4": 548, "5": 323, "6": 981, "7": 469, "8": 364, "9": 150}}
```


Now it is starting to look more like a proper instance. At this point we can add some knowledge about the problem to formulate some constraints that will ensure that the instances are not trivial. ie when the sum of all the weights is smaller than the capacity so we can't put all the objects in the knapsack or when all the objects are heavier than the capacity so that no object can be picked. Thefore we add constraints such as:

[15]:
```
%%conjure --solver=minion
letting number_items be 20
letting items be domain int(1..number_items)
find weight: function (total, injective) items --> int(1..1000)
find gain: function (total, injective) items --> int(1..1000)
find capacity: int(1..5000)

such that (sum([w | (_,w) <- weight]) > (capacity*2))
```


```
{"capacity": 1, "gain": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15, "16":
16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7, "8":
8, "9": 9}, "weight": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15, "16":
16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7, "8":
8, "9": 9}}
```


This means that the sum of all the weights should be greater than twice the capacity of the knapsack. From this we can expect that on average no more than half of the objects will fit in the knapsack. The expression `[w | (_,w) <- weight]` is a list comprehension that extracts all right hand values of the `weight` function. The underscore character means we do not care about the left hand side values. To ensure that the solver does not take it too far we impose an upper bound using a similar constraint. We impose that the sum of the objects weights 5 times the capacity of the knapsack, so we can expect that only between 20% and 50% of the items will fit in the knapsack in each instance.

[16]:
```
%%conjure+
such that (sum([w | (_,w) <- weight]) < capacity*5),
```


```
{"capacity": 43, "gain": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}, "weight": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}}
```


At this point it will be harder to see specific properties of the instances just by eyeballing the parameters but we can be confident that the properties we have imposed are there. We can add some more constraints to refine the values of the instances for practice/exercise by enforcing that no object is heavier than a third of the knapsack capacity

[17]:
```
%%conjure+ --solver=minion

such that forAll (_,w) in weight .  w < capacity / 3,
```


```
{"capacity": 63, "gain": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}, "weight": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}}
```


On top of that we can enfore a constraint on the density of the values in each object by limiting the ratio between the weight and gain of each specific object with:

[18]:
```
%%conjure+ --solver=minion
such that forAll element : items .
        gain(element) <= 3*weight(element)
```


```
{"capacity": 63, "gain": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}, "weight": {"1": 1, "10": 10, "11": 11, "12": 12, "13": 13, "14": 14, "15": 15,
"16": 16, "17": 17, "18": 18, "19": 19, "2": 2, "20": 20, "3": 3, "4": 4, "5": 5, "6": 6, "7": 7,
"8": 8, "9": 9}}
```


After running all cells, we can take the output solution and run the Knapsack Problem solution on it.

Tada! your model is being tested on some instance!

If your computer is powerful enough you can try larger values in “letting number_items be 20” (40-50 items will already produce substantially harder instances) Like for other forms of modelling writing instance generators is in large part an art. If this is not your kind of thing and you would like a fully automated system that can produce instances you may check out this method

(code available here)