Declarative Programming Project

Ward Muylaert

January 2014

1 Meta

This is Ward Muylaert's (#87041) submission for the January 2014 project of the Declarative Programming class at the Vrije Universiteit Brussel.

2 Description

Full description.

Given a list of objects, place them in two containers in (a) optimal way(s).

3 Software needed

SWI-Prolog version 6.4.1

4 Overview

At first an attempt was made to find only ideal solutions. It quickly became clear how unfeasible this was. Without having a good algorithm at our disposal, it just comes down to bruteforcing over a ridiculously high amount of combinations.

Instead was opted for a good enough solution. The idea is to first sort the given objects from biggest volume to smallest volume. Next we go through the list, each time trying to add the object to the container that is emptiest/lightest at that moment. When adding an object it is always placed as low and as left in the grid as possible. This assures, at the very least, that no object will be *completely* floating in the air. If a point is reached that the object does not fit in the container, then this object is set aside and the algorithm continues with the rest of the objects.

In most of the cases, this will give a decent solution to the problem. Decent in the sense that it comes close to a most balanced, least unused spaces as well as most used objects solution. The chance of putting a heavier object on top of a lighter one is also heavily reduced, but not eliminated, by the order we use to go through the objects.

5 Detailed

5.1 Container and Object

The project has two major data structures around which everything resolves. On the one hand there are the objects, as provided in the project description. An object is of this form.

```
object(ID, size(Height, Length, Depth)).
```

Throughout the project it is assumed `Depth = 1`.

The other structure is, obviously, a container. These are of the form

```
container(ID, size(Height, Length, Depth), Matrix).
```

Here too `Depth = 1` at all times. The `Matrix` part is a list of lists in accordance with the `Height` and `Length` mentioned in `size`. For example if `Height` would be 2 and `Width` would be 3, then the `Matrix` would be of the form

```
[
    [0,0,0],
    [0,0,0]
]
```

The number 0 is used as the default value. When an object gets added into a container, the appropriate values in the matrix get changed to the object's `ID`.

5.2 Matrix

Since we essentially keep track of the state of a container by using a matrix, it was deemed appropriate to write some functions that handle matrices in ways that will be useful.

The concepts deemed appropriate were on the one hand variants of `nth0/3` (one for the value in a matrix, one for the value of a range in a list, one for the values of a block in a matrix) and on the other hand functions that take a list or matrix and change the value on a certain spot (or on a range/block).

5.3 Working with the data

Some functions are added to manipulate the containers and objects to suit our needs.

For the objects there is the addition of a function to calculate its volume as well as something to sort a list of objects by volume (lowest volume to highest volume). This sort uses insertion sort.

On the container end there are some functions which are mostly wrappers for the ones we mentioned creating for matrices. Checking for a free spot is done by checking for spots of value 0 in the container's content matrix. Placing an object in a container is changing a block of values in its matrix.

5.4 Execution

The actual algorithm then uses all these things to:

1. Get all the objects and containers
2. Sort the object list from biggest volume to smallest volume
3. Step through the list of objects
    1. If first container is lightest, try to place it there. If container one is not the lightest or if the object doesn't fit, go to the next step.
    2. Try to place the object in the other container. If it doesn't fit, continue to the next step.
    3. Object doesn't fit in either container, put it in skipped list.
    4. Go back to first step with the next object.

6 Manual

This section describes how to use the program. Start `swipl` and consult `main.pro` or load it immediately on startup by issuing the following command in the terminal (while in the project folder).

```
swipl -l main.pro
```

This loads in all the needed predicates except the objects that you want to try placing in the two containers. This is a matter of consulting a file that contains their definitions. Some example sets are given in the `datasets/` folder. For example to load in `data1.txt` from that folder, issue the following in the swipl command line.

```
consult('datasets/data1.txt').
```

Now that there are some `objects` in the program, finding the solution given our strategy is as simple as issuing

```
place_objects.
```

7 The first attempt

For completeness sake, the *failed* attempt to find an ideal solution is explained here, it essentially came down to checking pretty much every possibility. As mentioned before, this became quickly unfeasible due to the cheer amount of possibilities. While a program could be written, it would essentially never finish.

The idea was that, given a set of objects, the set is split in every way without being redundant. By this is meant that the following splits of a list `[1,2,3]` are all equal.

```
[1]   [2,3]
[1]   [3,2]
[2,3] [1]
[3,2] [1]
```

So that list of `[1,2,3]` was split up into the following 4:

```
[1,2,3] []
[1,2]   [3]
[1,3]   [2]
[2,3]   [1]
```

This came down to for a list of length `n`, we receive `2^(n-1)` possibilities. That's just splitting up the input into which object goes in which container though! It doesn't end there.

Next, given a certain split up of the set into `List1` and `List2`, try all arrangements of `List1` into the first container and all arrangements of `List2` in the second container. Here the criteria are that every object is placed as low as possible and as left as possible when it is handled. This more or less (there were some improvements, but they were minor) comes down to trying all permutations of that part of the split. In other words splitting a list of 4 objects into two lists of 2 still requires checking up to `2!` (ok that sounds less impressive in the low amounts) of arrangements in each container. It requires no explanation that this quickly results in *huge* amounts.