Corticon - Making Change Possible

Decision Modeling Challenge February 2015

Use Case

How can a given amount of money be made with the least number of coins of given denominations?

Let S be a given sum that we want to achieve with a minimal amount of coins in denominations of x1, x2, …, xn. Here is a simple example: S is 123 cents, n is 4, x1 is 1 cent, x2 is 10 cents, x3 is 25 cents, and xn is 100 cents.

This seemingly simple problem turns out to be quite tricky in its most general form.
We'll present two approaches to this:

- The first makes a few reasonable assumptions about the data since we are trying to make change for real people with real money – and this leads to quite a simple rule model that's easy to understand.
- The second shows a more generalized solution which requires more sophisticated rule modeling techniques and more computing resources and is a little harder to understand but it can solve scenarios that cause the simpler approach to fail.

Ultimately, to get a solution that works for all possible scenarios in a reasonable amount of time, one would probably use a custom Constraint Solver product. However from the examples below we'll show that Business Rules Engine is also able to solve such problems in a simple and elegant way.

Solution 1 – The Simple Approach

This solution employs the "Greedy Algorithm". I.e. successively subtracting the largest coin that does not exceed the amount remaining. http://en.wikipedia.org/wiki/Greedy_algorithm

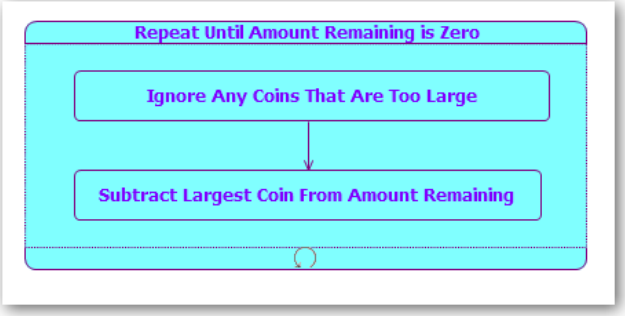
This has the advantage that it's very simple – just two rules - and it will find the optimal solution for normal currencies. However it is possible to choose coin values that will cause the greedy algorithm to fail. For example if the sum of lower value coins exceeds a higher value coin or if there is no 1 coin. This more complex case is discussed later. Normal currencies in use today are specifically designed to avoid this problem as are all number systems (otherwise you'd find there were multiple ways to represent the same number).

The solution presented here is implemented in Corticon using **Extended Decision Tables** (this is a basic decision table supplemented by a SCOPE and FILTER section that defines the context in which the decision table must be applied.

More details regarding Corticon features, functionality and concepts can be found in the online documentation available at http://documentation.progress.com/output/ua/Corticon and in previous challenges, especially the ""Vehicle Insurance – UServ Product Derby" from last December.

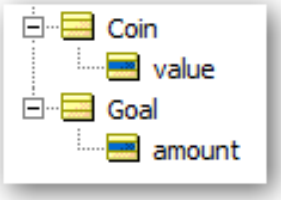
Rule Flow

This pair of rule sheets is executed repeatedly.
This is indicated by the little arrow symbol.
The pair is repeated until no more rules fire.



Vocabulary

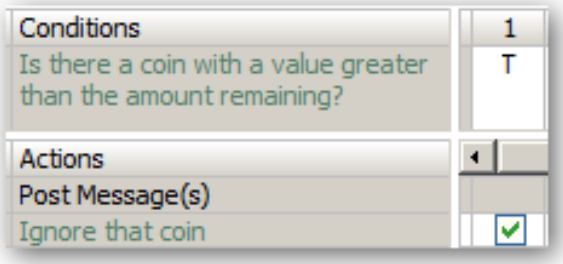
The vocabulary is very simple.
A single object called Goal to hold the amount
One or more objects called Coin to hold the values available for making change.



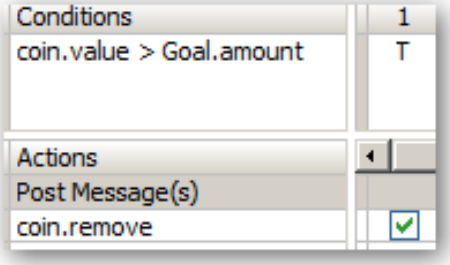
Rule Sheets

Ignore any coins that are too large

Specification

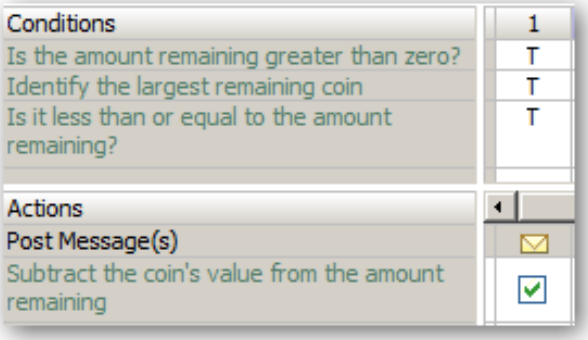


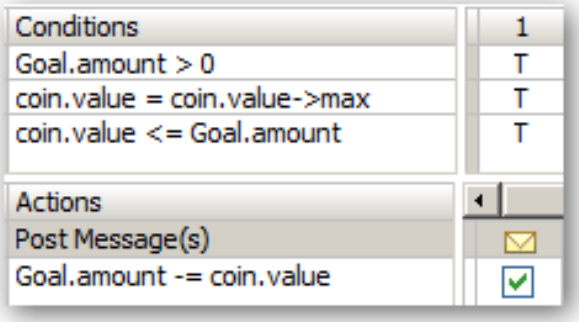
Implementation



Corticon naturally will apply rule 1 to ALL the coins that are in memory. The rule author does not need to define any loops (as they might have to do in java). The **.remove** operator deletes the **coin** from memory.

Subtract largest coin from amount remaining





The key to solving this challenge is the use of this expression:

coin.value = coin.value->max

It does something very powerful:

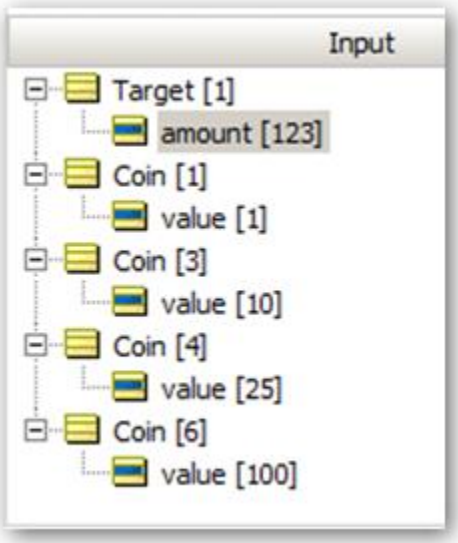
- The reference to **coin** means a specific coin in the collection of possible coins (as defined in the input data).
- The second reference to **coin** means ALL the coins that are currently in memory, the **->max** operator finds the largest of them.

So the net result is that this statement identifies coin as the one with the largest value.
That particular coin is then used in the third condition to see if it's less than or equal to the Goal amount. If it is then it value is subtracted from the amount remaining.

The rule flow diagram allows this process to repeat. It stops when no more rules fire. And as long as we have a 1 coin we are guaranteed a solution.

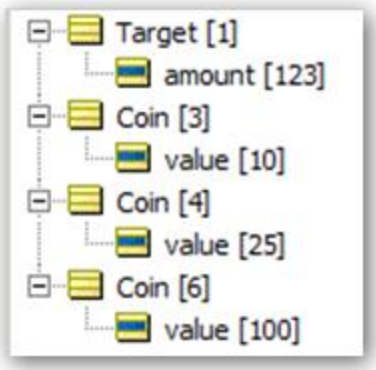
Test Case

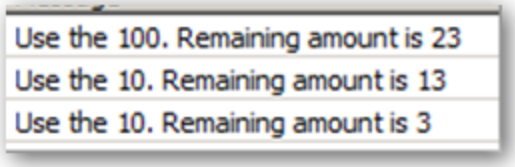
Here is the test case referred to in the spec:



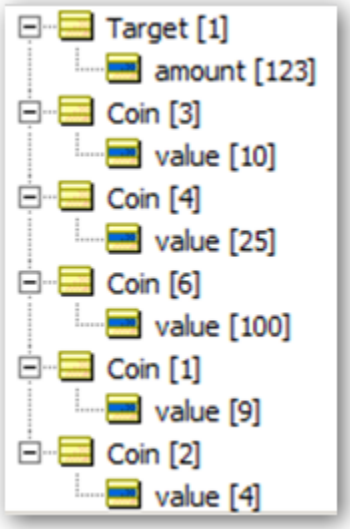
Severity	Message
Info	Use the 100. Remaining amount is 23
Info	Use the 10. Remaining amount is 13
Info	Use the 10. Remaining amount is 3
Info	Use the 1. Remaining amount is 2
Info	Use the 1. Remaining amount is 1
Info	Use the 1. Remaining amount is 0

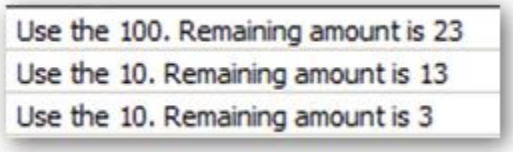
NOTE: It is possible for there to be no solution when there is no 1 coin.
For example to make 123 from 10s, 25s and 100s (no 1's) this is the best we can do:





But what about this test case:





There is a solution 100, 10, 9, 4 but the simple “Greedy” rules as specified do not find it.

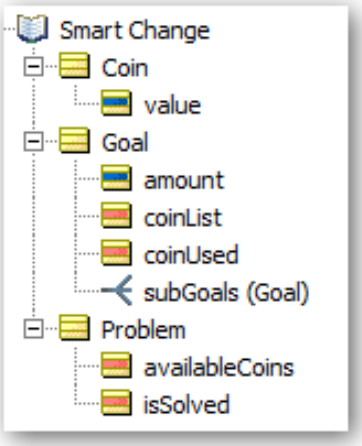
Can you see a way to deal with this problem?

Solution 2 –Improved and Generalized Solution

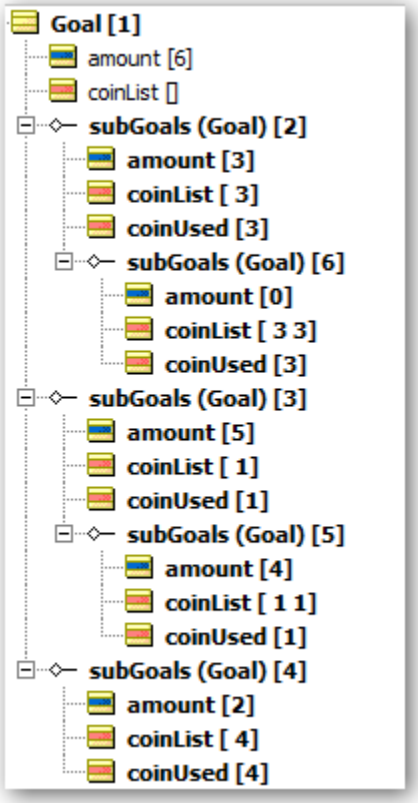
We can't simply subtract the largest coin since that one may not be part of the final solution. We have to try all possible combinations of coins in order to determine the smallest set. This is what makes this scenario much more challenging (and therefore why currencies keep to the simple case)
In order to keep track of many possible combinations of coins at the same time we'll employ a tree structure.

Vocabulary

The approach we'll take in this rule model is to start with the Goal amount and then build a tree structure that represents all the possible ways that total amount can be constructed using the various coin values. You can see we have modeled a recursive structure in the vocabulary to accommodate the possible coin values we could use at each step of the process. It shows that a Goal can have sub goals which themselves are Goals.



Suppose the Goal is 6 and we have the following coins: 1, 3, 4. The Greedy Algorithm discussed earlier will find 4, 1, 1 as a solution, but that's not the best. In fact 3, 3 is the best.



Goal [1] is the starting point with an amount of 6. The **coinList** attribute is empty because we haven't yet chosen any coins.

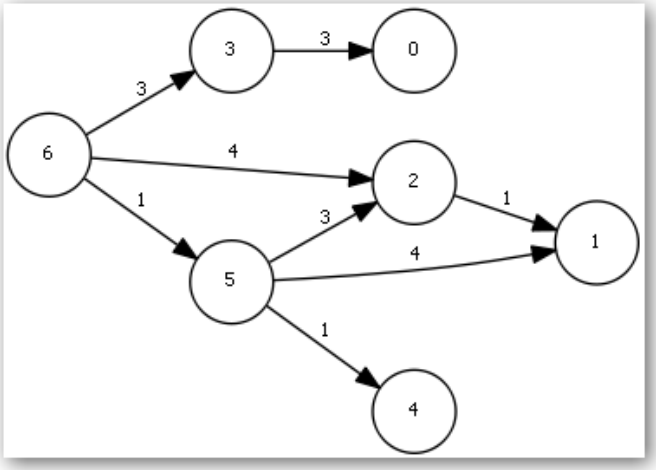
Sub goals [2], [3] and [4] show the three possible choices of a first coin and the new amount that results from using that coin (**coinUsed** attribute value). The **coinList** contains just one coin at this stage.

Once the new amount reaches zero we have a solution.

Sub goal [2] (coin used =3, amount = 3) in turn has a sub goal [6] (also coin used 3, but now the amount is zero) and this solves the problem.

Once we have a solution we can stop creating sub goals.

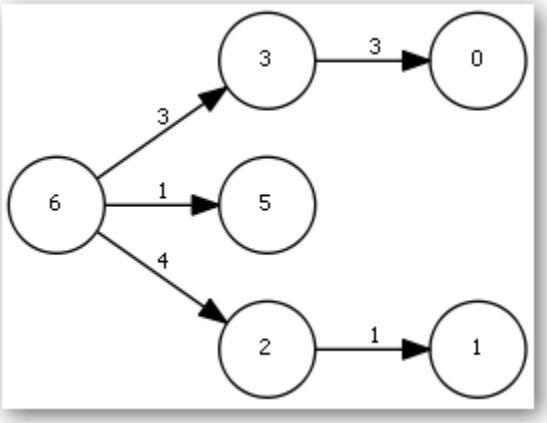
The search tree can also be shown graphically:



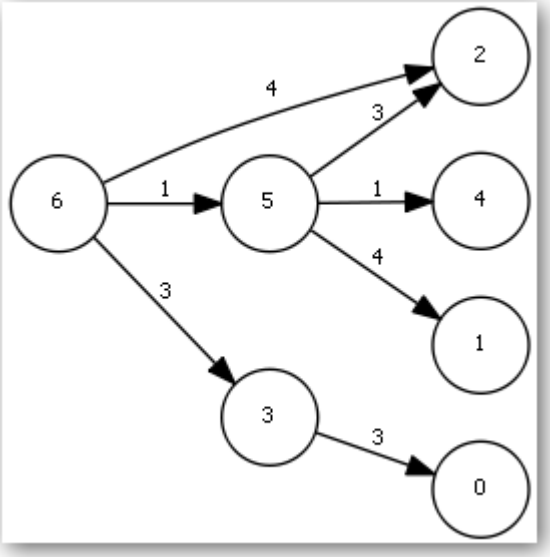
Note Corticon is using a breadth first search which means that at each level it tries every coin (with a value less or equal to the current goal amount) before moving to the next deeper level.
But because Corticon is free to pick the coins in any order you may get different search trees each time depending on the order chosen.
It's not clear whether there is a best strategy for choosing the order of the coins. I would be interested in others' opinions

Some searches will be quicker than others though never deeper than necessary.

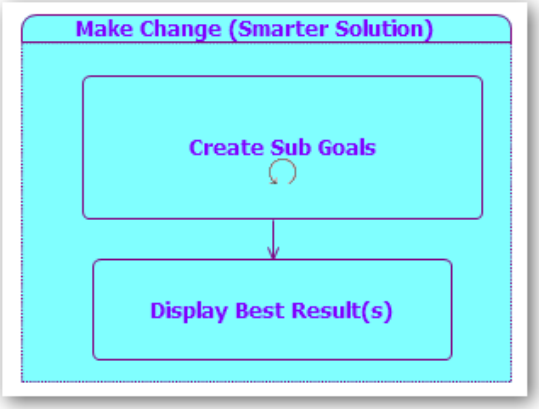
For example, in this case Corticon found a solution rather quickly and so didn't need to expand any more nodes:



In this test, more nodes were expanded before finding a solution:



The Rule Flow



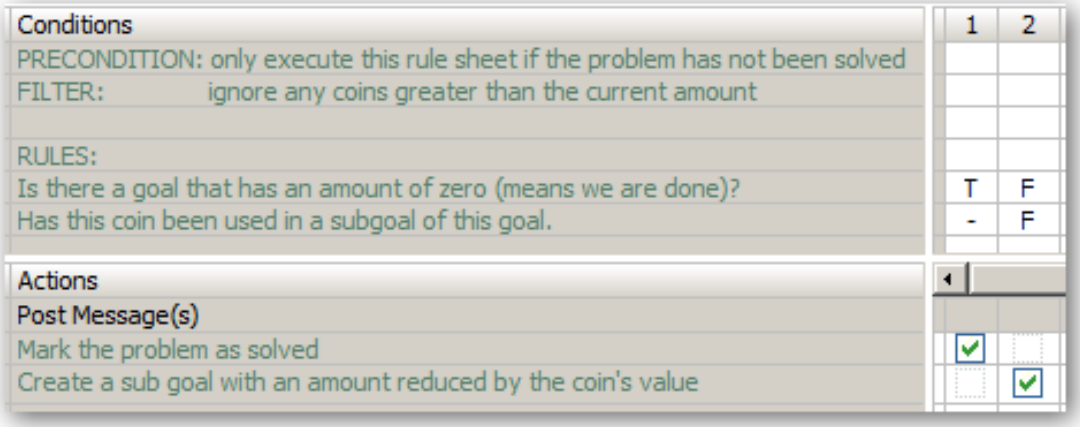
This solution also has two rule sheets.

The first rule sheet generates the next set of sub goals (one for each possible coin) for each current goal. It is repeated until it can reduce the goal amount to zero.

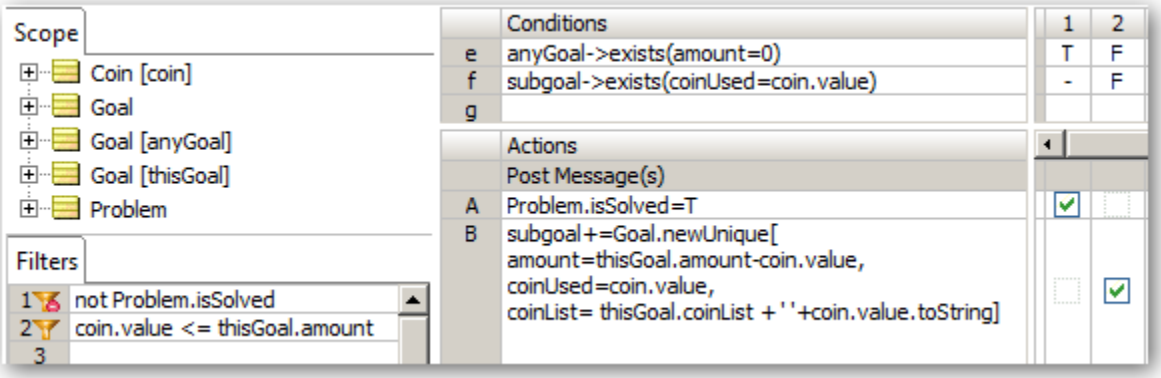
So this time we are repeating a single rule sheet, not the entire rule flow.

The second rule sheet simply displays that result.

Specification for Create Sub Goals rule sheet



Implementation



Note: The first four lines of the natural language view are for documentation purposes (which is why the actual rule conditions are e and f. The precondition is actually implemented in Filter #1 (notice the little red decoration) and the filtering to ignore larger coins is implemented in filter #2

How it Works

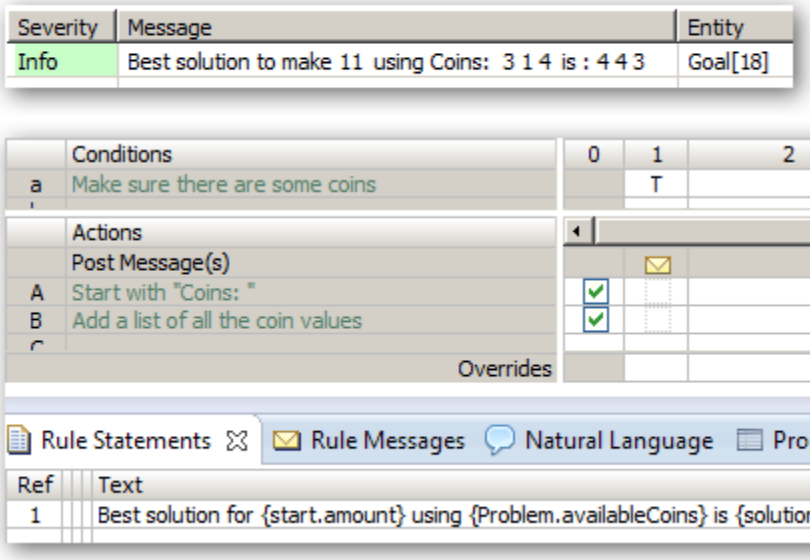
Once rule 1 has fired (i.e. we have a goal with an amount of zero) we set **Problem.isSolved** to True and then Corticon will stop repeating this rule sheet
Filter #1 (a precondition) prevents the rule sheet from executing once the problem is solved.
Filter #2 limits the coins considered by the rule sheet to those whose values are less that the current goal amount. Effectively we create a subset of the coins.

Condition "**e**" checks if **any** goal exists with an amount of zero (which means we have a solution).
Condition "**f**" makes sure we don't use the same coin more than once at each sub goal level.
Though the same coin can be used at a *different* level.

Action "**A**" sets the **isSolved** attribute that stops the rule sheet from repeating forever.
Action "**B**" creates a new sub goal and sets its specific attributes: the resulting amount, the coin used and the list of coins used down to that sub goal.

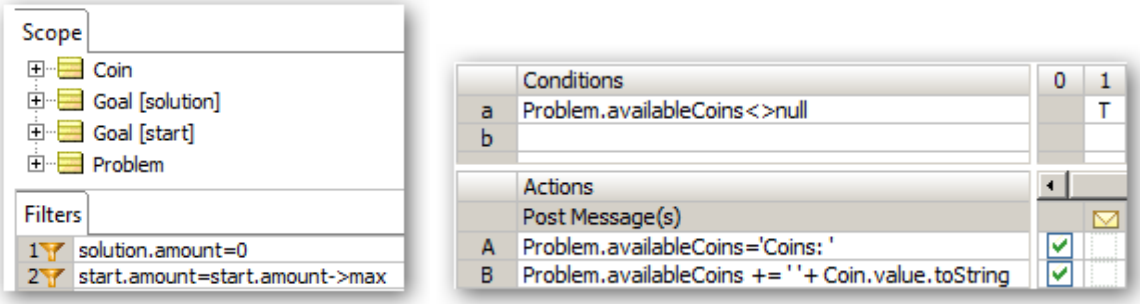
Specification for Display Best Result(s) rulesheet

The result is displayed as a simple message:



The message is generated using the rule statement which has embedded variables shown in {}

Implementation



Corticon will automatically use *all* the coin values in action "B**"** (you don't have to write an explicit loop)
Action "A" initializes the **availableCoins** so we can concatenate individual coin values in action "B".

Filter #1 selects the Goal (alias **solution**) that solved the problem (so we can display **coinList**)
Filter #2 selects the Goal (alias **start**) that we started with (so we can display the starting **amount**)

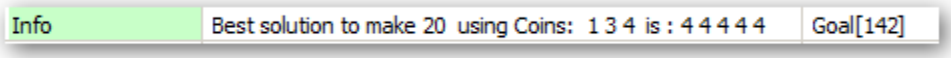
The two aliases (**solution** and **start**) that refer to the same business object (**Goal**) needs some explaining. This is part of the power of Corticon. It allows us to refer to two different items (or collections) within the set of Goals.

The filter expressions define what goals belong to each alias. In this case there are two types of Goal that are of interest:

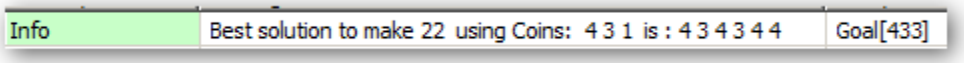
- The first is the starting goal – and this can be identified because it's the one with the largest value for amount. (**start.amount=start.amount->max**)
- The second is the solution goal which will have an amount of zero (**solution.amount=0**) and will also contain in its **coinList** attribute a list of all the coins that were used.

Test Case

In this case with target amount of 20 and coins 1, 3, 4 a solution is found after 142 combinations are tried.



Increasing the target amount by just two causes the search to consider 433 possibilities before finding a solution



What happens when there are two shortest solutions?

Let's consider this scenario: obtain 6 using 1,2,3,4

Depending on the order Corticon happens to choose the coins at each level so it may find the 4-2 or the 3-3 solution. But since they are both 2 coins it doesn't really matter.
If we wanted to find ALL solutions for the minimal number of coins then that would require a bit more work. Instead of stopping immediately upon finding a solution we'd need to let Corticon try all the coins at that level before stopping and then the Display rule sheets will list all solutions that resulted in zero. Maybe we want to find a solution that has as many different coins as possible? i.e. avoiding repeats.

Conclusion

Most business rules engines can probably solve this problem, but as the gap between the goal amount and the highest value coin increases, the number of combinations to be considered increases dramatically – eventually exceeding the memory capacity of the computer.
A more sophisticated (and probably more complex) algorithm is probably going to be required to solve the general case. At that point one should probably consider using a Constraint Engine. Maybe it's possible to use a combination of the Greedy and the Smart algorithms?