２０１０年7月２２日

インテリジェントデザインレポート
担当：廣田薫 先生　（東京工業大学）

The Water Jar Problem

氏名：モーノ　ドミニク　(Morneau Dominic)
学籍番号：90910141
慶應義塾大学理工学部　特別短期留学生
dmorneau@gmail.com

In order to apply the principles we have seen during the « Intelligent Systems Design » course this semester, I have chosen to implement a solution to the water jug problem. I have separated the implementation in a reusable production rules module, and a module specific to the problem. I will go over the code and describe the algorithm, then at the end discuss the performance characteristics.

I have chosen Haskell as an implementation language. Haskell is a lazy language which seemed rather well suited to implementing a search problem. It would also have been very easy to implement it in Prolog, but since the goal of this report is partly to implement a breadth-first search, I have avoided using Prolog (or Haskell's) built-in search routines.

Let us start with the two basic data types, a production rule and an intermediary state.

```
data Rule a b = Rule { name :: b, apply :: a -> Maybe a }

data IState a b = IState { history :: [Rule a b], value :: a}
```

The Haskell syntax may take some time to get used to. Here, type 'a' represents the type of state that we use. For example, for the jars problem, the state 'a' is a pair of integers such as (4, 3) which represents the amount of water in jar A and B. Type 'b' is the type of the production rules' description, usually a String.

We define a Rule to have a name (such as "Fill bucket A"), and a function "apply" which takes an 'a' and may return an 'a' or return nothing. An IState is defined to be a transformation history (a list of Rules applied so far), and a value.

In order to make the code clearer, let us define the jars problem using these data types.

```
type JarState = (Int, Int)
type JarRule  = Rule JarState String

rules :: Int -> Int -> [JarRule]
rules maxX maxY = map (uncurry Rule) [
 ("R1 Fill jar A.", \(x,y) -> when (x < maxX) (maxX, y) ),
 ("R2 Fill jar B.", \(x,y) -> when (y < maxY) (x, maxY) ),
 ("R3 Empty jar A.", \(x,y) -> when (x > 0) (0, y) ),
 ("R4 Empty jar B.", \(x,y) -> when (y > 0) (x, 0) ),
 ("R5 Pour B into A until A is full.", \(x,y) ->
      when (x + y >= maxX && x < maxX && y > 0) (maxX, y - (maxX - x)) ),
 ("R6 Pour A into B until B is full.", \(x,y) ->
      when (x + y >= maxY && x > 0 && y < maxY) (x - (maxY - y), maxY) ),
 ("R7 Pour B into A until B is empty.", \(x,y) ->
      when (x + y < maxX && y > 0) (x + y, 0) ),
 ("R8 Pour A into B until A is empty.", \(x,y) ->
      when (x + y < maxY && x > 0) (0, x + y) )]
    where when c v = if c then Just v else Nothing
```

The JarState is defined as a pair of Int. For example, (4,3) would be the JarState where both jars are filled with water. A JarRule is a production rule on JarState and String; this is more easily understood by substituting the types in the definition of Rule:

```
data Rule a b = Rule { name :: b, apply :: a -> Maybe a }
data Rule JarState String = Rule { name :: String, apply :: JarState -> Maybe JarState }
```

As for the rules themselves:

```
("R1 Fill jar A.", \(x,y) -> when (x < maxX) (maxX, y) )
…
where when c v = if c then Just v else Nothing
```

The rule's name is "R1 Fill jar A.", and the rule's function is: if the amount of water in the first bucket is inferior to the maximum, fill the bucket (return a new state with X at the maximum and Y unchanged). Otherwise, return Nothing.

Using these blocks we can try to solve the search problems. We first define the application of a rule to an intermediary state:

```
applyRule :: IState a b -> Rule a b -> Maybe (IState a b)
applyRule x r = apply r (value x) >>= (Just . IState (r:history x))
```

applyRule takes an intermediary state x and a production rule r. It tries to apply the rule to the state. If the result is a new state, we return an IState with the new value and append the production rule r to its history. Otherwise, we return nothing. Haskell details such as the (>>=) operator are beyond the scope of this document, but more information can be obtained on the internet and elsewhere by looking for the "bind" function on monads.

Next we have the most important function in the program, the one which implements the breadth-first search.

```
expandLevel :: (Ord a) => [Rule a b] -> (Set a, [IState a b])
            -> (Set a, [IState a b])
expandLevel rs (s, x) = (set, states)
    where states = filter (flip notMember s . value) (x >>= applyRules)
          set = s `union` fromList (map value states)
          applyRules y = mapMaybe (applyRule y) rs
```

expandLevel takes a list of production rules, a set of states, and a list of intermediary states, returning an updated set and a new list. More concretely: expandLevel takes one level of the tree, in this case variable 'x' (of type [IState a b]). It generates the next level in the tree by applying all the production rules to all states in the current level (x >>= applyRules), then removing all states that have been seen before (members of the set s). After this, the new states are added to the set, and the new level is returned along with the updated set.

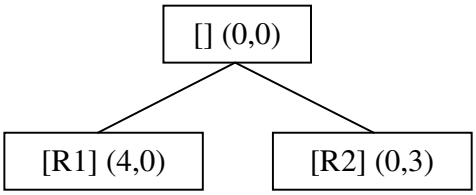
```
expandTree :: (Ord a) => a -> [Rule a b] -> [[IState a b]]
expandTree x rs = takeWhile (not . null) $ map snd $
                  iterate (expandLevel rs) (singleton x, [IState [] x])
```

Then we have a function expandTree, which starts from one initial state with no history and a set containing only this state, and repeatedly calls expandLevel to generate the list of all levels in the tree. This list may be infinite – Haskell being a lazy language, the list will be evaluated on-demand, not all at once, so infinite lists are not a problem.

Finally to solve a search problem we use this function:

```
solve :: (Ord a) => (a -> Bool) -> [Rule a b] -> a -> Maybe (a, [Rule a b])
solve p r i = (value &&& history) <$> (find (p . value) . concat $ expandTree i r)
```

First, we take the tree generated by expandTree. This tree is organized by levels:



This tree would be encoded as [[IState [] (0,0)], [IState [R1] (4,0), IState R2 (0,3)]], that is, a list of levels, each level being a list of intermediary states.
We concatenate to eliminate a level of nesting, yielding [IState [] (0,0), IState [R1] (4,0), IState [R2] (0,3)].
This is, in essence, an in-order traversal of the tree, expressed as a possibly infinite list of states. Then we can use the find function to try to find a state in this list which satisfies the condition we are looking for; if we do, we return a pair containing the value and history (list of rules applied) for the final state found.

In addition to these basic search functions, I have implemented a few useful variations to know the number of states searched or limit the search depth. The final program, jars.hs, is called with a letter indicating the mode: 's' for show solution, 'n' to show the number of steps (tree depth) necessary to reach a final state, 'c' to know the number of states searched before finding a solution, and 't' to print the in-order traversal of the whole tree. This is followed by the maximum amount of water in jar A, in jar B, then the final state for A and B.

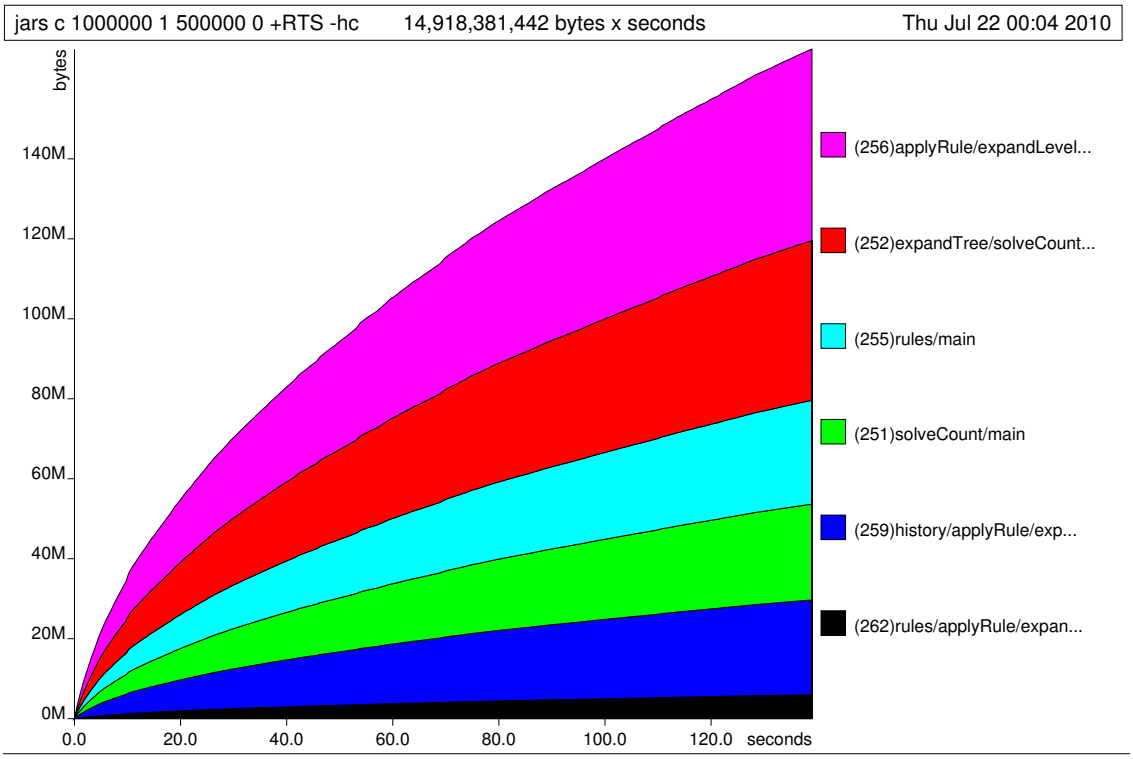
Here are some examples of the program being called with jar A containing 4 liters, B 3 liters, and a final state of (2, 0):

```
>jars c 4 3 2 0
Solution found, 14 states tried
>jars t 4 3 2 0
[(0,0),(4,0),(0,3),(4,3),(1,3),(4,3),(3,0),(1,0),(3,3),(0,1),(4,2),(4,1),(0,2),(2,3),(2,0)]
>jars n 4 3 2 0
Number of steps:
6
>jars s 4 3 2 0
Solution found:
(0,0)R2 Fill jar B.
(0,3)R7 Pour B into A until B is empty.
(3,0)R2 Fill jar B.
(3,3)R5 Pour B into A until A is full.
(4,2)R3 Empty jar A.
(0,2)R7 Pour B into A until B is empty.
```

The program can be called with much bigger trees, for example jars s 1000000 1 500000 0 is solvable within about 15 seconds on a laptop (the solution requires 500000 rule applications). Although it does not affect the correctness of the algorithm, there may be a bug remaining: the line `states = filter (flip notMember s . value) (x >>= applyRules)` in expandLevel removes all states that have been seen in the previous levels, but it does not

consider that states in the current level may be redundant; that is, multiple identical states may appear in one level without being eliminated. For a more correct implementation we could use the function “nub” to eliminate duplicates in the list.

The memory usage for a “jars c 1000000 1 500000 0” call follows this graph:



The runtime is longer than usual due to using profiling functions. Since using the breadth-first traversal requires storing the intermediate levels of the tree, the program requires a lot more memory than if it used a depth-first algorithm; however, it is guaranteed to find the shortest solution possible.

Profiling the code with the new visual-prof library yields the following highlights. As we can see, the slowest component seems to be the set. Adding the new elements to the set (union) and checking for membership takes a lot of execution time. I have not made a special effort to optimize the code so there may be further room for improvement.

```
applyRule :: IState a b -> Rule a b -> Maybe (IState a b)
applyRule x r = apply r (value x) >>= (Just . IState (r : history x))

expandLevel ::
              (Ord a) => [Rule a b] -> (Set a, [IState a b]) -> (Set a, [IState a b])
expandLevel rs (s, x) = (set, states)
  where states = filter (flip notMember s . value) (x >>= applyRules)
        set = s `union` fromList (map value states)
        applyRules y = mapMaybe (applyRule y) rs
```

In the next pages is the full source code for the jars application. The Windows binary (standalone .exe, 215kB) or source files may be provided on demand.

ruleng.hs

```
-- | Rule engine. This module can be used to express a problem as a
--   set of states and transformation rules. Breadth-first search is
--   then used to try all valid transformations, until a state that
--   satisfies all constraints is found.
module Ruleng where
import Data.Maybe (mapMaybe)
import Data.Set (Set, union, notMember, fromList, singleton)
import Data.List (find, findIndex)
import Control.Arrow ((&&&))
import Control.Applicative

-- | Production rule.
data Rule a b = Rule {
      -- | This can be used to hold a name, number or identifier.
      name :: b,
      -- | Applies the rule if possible, transforming the state.
      apply :: a -> Maybe a
    }

-- | Intermediary state. A state, plus its transformation history.
data IState a b = IState {
      -- | Rules applied so far. The head of the list is the rule that
      -- was applied /last/.
      history :: [Rule a b],
      -- | Current state.
      value :: a
    }

-- | Applies a transformation rule to a state.  If the rule was
-- successfully applied, the rule is added to the state's
-- transformation history. Otherwise Nothing is returned.
applyRule :: IState a b -> Rule a b -> Maybe (IState a b)
applyRule x r = apply r (value x) >>= (Just . IState (r:history x))

-- | Takes a tree level (a list of states at a given depth), and a
-- list of unique nodes so far. Generates the next level of depth (a
-- new list of states), eliminating states that have been reached
-- before.
expandLevel :: (Ord a) => [Rule a b] -> (Set a, [IState a b])
            -> (Set a, [IState a b])
expandLevel rs (s, x) = (set, states)
    where states = filter (flip notMember s . value) (x >>= applyRules)
          set = s `union` fromList (map value states)
          applyRules y = mapMaybe (applyRule y) rs

-- | Generates the whole tree of transformations from an initial
-- state. Returns a list of levels; element 0 is the list of elements
-- at depth 0, and so on.
expandTree :: (Ord a) => a -> [Rule a b] -> [[IState a b]]
expandTree x rs = takeWhile (not . null) $ map snd $
                  iterate (expandLevel rs) (singleton x, [IState [] x])

-- | Solves predicate p, under rules r, deriving from initial state i.
-- Generates the tree of states by recursively applying the rules
-- until there is no more possible transformation or a state that
-- satisfies predicate p has been found. The history is in reverse
-- order (the last rule to have been applied is first in the list).
solve :: (Ord a) => (a -> Bool) -> [Rule a b] -> a -> Maybe (a, [Rule a b])
solve p r i = (value &&& history) <$> (find (p . value) . concat $ expandTree i r)

-- | Same as solve, but with a limit on the tree depth.
solveDownTo :: (Ord a) => (a -> Bool) -> [Rule a b] -> a -> Int
            -> Maybe (a, [Rule a b])
solveDownTo p r i l = (value &&& history) <$>
                      (find (p . value) . concat . take l $ expandTree i r)
```

```
-- | Counts the number of states necessary to solve a problem
solveCount :: (Ord a) => (a -> Bool) -> [Rule a b] -> a -> Either Int Int
solveCount p r i = case findIndex (p . value) tree of
                     Nothing -> Left $ length tree
                     Just n  -> Right n
    where tree = concat $ expandTree i r

-- | Takes an initial state, a list of rules, and rebuilds the list of
-- intermediate states.
replay :: a -> [Rule a b] -> [a]
replay x [] = []
replay x (r:rs) = case apply r x of Nothing -> []
                                    Just x' -> x : replay x' rs
```

jars.hs

```
module Main where
import Ruleng
import Control.Applicative
import System.Environment
import Text.Printf

type JarState = (Int, Int)
type JarRule  = Rule JarState String

rules :: Int -> Int -> [JarRule]
rules maxX maxY = map (uncurry Rule) [
 ("R1 Fill jar A.", \(x,y) -> when (x < maxX) (maxX, y) ),
 ("R2 Fill jar B.", \(x,y) -> when (y < maxY) (x, maxY) ),
 ("R3 Empty jar A.", \(x,y) -> when (x > 0) (0, y) ),
 ("R4 Empty jar B.", \(x,y) -> when (y > 0) (x, 0) ),
 ("R5 Pour B into A until A is full.", \(x,y) ->
      when (x + y >= maxX && x < maxX && y > 0) (maxX, y - (maxX - x)) ),
 ("R6 Pour A into B until B is full.", \(x,y) ->
      when (x + y >= maxY && x > 0 && y < maxY) (x - (maxY - y), maxY) ),
 ("R7 Pour B into A until B is empty.", \(x,y) ->
      when (x + y < maxX && y > 0) (x + y, 0) ),
 ("R8 Pour A into B until A is empty.", \(x,y) ->
      when (x + y < maxY && x > 0) (0, x + y) )]
    where when c v = if c then Just v else Nothing

solveJars maxX maxY initial final = solve (==final) (rules maxX maxY) initial

showSolution mx my i f = case solveJars mx my i f of
                           Nothing -> putStrLn "No solution"
                           Just (v,h) -> do
                             putStrLn $ "Solution found: "
                             mapM_ (\(x,y) -> putStr x >> putStrLn y) desc
                                 where desc = zip (map show $ replay i history)
                                                  (map name history)
                                       history = reverse h

countSteps mx my i f = length . snd <$> solveJars mx my i f

main = do
  args <- getArgs
  let out = head args
  let [mx, my, fx, fy] = map read $ tail args
  case out of
    "s" -> showSolution mx my (0,0) (fx, fy)
    "n" -> do putStrLn $ "Number of steps: "
              case countSteps mx my (0,0) (fx, fy) of
                Nothing -> putStrLn "No solution"
                Just n  -> print n
    "c" -> case solveCount (==(fx,fy)) (rules mx my) (0,0) of
             Left n -> printf "No solution, %d states tried" n
             Right n -> printf "Solution found, %d states tried" n
    "t" -> print . map value . concat $ expandTree (0,0) (rules mx my)
```

Bonus: Travelling salesman

The following is a Haskell solution to the travelling salesman problem as seen in class.

```
import Data.List (minimumBy, permutations)
import Data.Ord (comparing)

type City = Int
type Distance = Int

--            札幌  東京  名古屋  大阪  福岡
distance :: City -> City -> Distance
distance a b = [[  0, 230, 260, 260, 280]
               ,[230,   0, 120, 200, 240]
               ,[260, 120,   0,  80, 230]
               ,[260, 200,  80,   0, 220]
               ,[280, 240, 230, 220,   0]] !! a !! b

total :: [City] -> Distance
total xs = sum $ zipWith distance (init xs) (tail xs)

-- Finds the shortest itinery start from city x and passing through a
-- list of cities.
salesman :: City -> [City] -> [City]
salesman x = minimumBy (comparing (total . (x:))) . permutations

-- Solves the Salesman problem, with starting city Tokyo (1), visiting
-- in any order Sapporo, Nagoya, Osaka and Fukuoka ([0,2,3,4]).
main :: IO ()
main = putStrLn "Shortest path: " >>
       mapM_ (putStrLn . cityName) (salesman 1 [0,2,3,4])

cityName :: City -> String
cityName = (!!) ["Sapporo", "Tokyo", "Nagoya", "Osaka", "Fukuoka"]
```

Example run :

```
>sales
Shortest path:
Nagoya
Osaka
Fukuoka
Sapporo
```