Representation for N Jug Problem

Chris Brown

February 17, 2003

;;; generate successors for n-jug problem

;;; For N jugs, state is N+1 long list giving the current contents
;;; of all the jugs.

;;; The zeroth jug is an infinite source and sink of water and is thus
;;; treated a little differently.

;;; But that means there is only ONE operation, "pour from jug1 into jug2",
;;; whose arguments (jug1, jug2) are the operation’s description (op-desc).

;;; A node is a triple (3-long list) of
;;; (state, op-desc, parent)

;;; disallow
;;; previously generated states (use hashtable)
;;; filling full jug
;;; pouring from empty jug
;;; try to fill jug from self
;; Here’s all you need for a hash function

(setf visited (make-hash-table :test #'equal))

(defun not-visited (state)
  (not (gethash state visited)))

(defun mark-visited (state)
  (setf (gethash state visited) 1))

;;; initialize some obvious variables: capacities is how big the jugs are
;;; (we want jug0 to have nonzero contents for a test later but we don’t care
;;; what its capacity is)

(defconstant n 3)

(setf capacities '(0 3 5))
(setf init-state '(1 0 0))
(setf goal-state '(1 0 4))
(setf start-node (list init-state nil nil))

(setf nodes nil)

;;; clearly initial state is going to be visited...

(mark-visited init-state)

;;; super-simple accessor functions for our node "structure"

(defun state-of (node)
  (car node))
(defun op-of (node)
  (cadr node))
(defun anc-of (node)
  (caddr node))
;;; given a state and two jugs, what is the new state?
(defun new-state ( s j1 j2)
  (let (cont1 0) (cont2 0) (news nil))

; remember contents of two jugs
  (setf cont1 (nth j1 s))
  (setf cont2 (nth j2 s))

; now lots of tests to "do the right thing"
  (cond
    ; don’t fill self from self
    ( (= j1 j2) (setf news nil))

    ; don’t fill from empty jug
    ( (= cont1 0) (setf news nil))

    ; don’t fill full jug
    ( (= cont2 (nth j2 capacities)) (setf news nil))

    ; fill j2 from reservoir
    ( (zerop j1) (setf news (copy-list s))
      (setf (nth j2 news) (nth j2 capacities)))

    ; dump j1 into reservoir
    ( (zerop j2)
      (setf news (copy-list s))
      (setf (nth j1 news) 0 ))

    ; finally the "general" case --
    ; j1 is empty or has cont j1- (cap(j2)- contj2), whichever is >
    ; j2 is full or has conts j2 + j1, whichever is <
    (t
      (setf news (copy-list s))
      (setf (nth j1 news) (max 0 (~ cont1 (~ (nth j2 capacities) cont2))))
      (setf (nth j2 news) (min (nth j2 capacities) (+ cont1 cont2 ))))

; return new state
  news

)
;;;; Generate all successors... examine every (j1, j2)
;;;; pair to see which new states to add to successor list.
;;;; new-state can return nil (ignore) or an already-visited
;;;; state (ignore using hash table).

(defun gen-succs (node)
  (let (succs nil) (newnode nil) (jug1 0) (jug2 0))

    (dotimes (jug1 n)
      (dotimes (jug2 n)

        ; generate a new state...it’s either OK, nil (bad choice of j1,j1)
        ; or visited

        (setf newstate (new-state (state-of node) jug1 jug2))

        (when (and newstate (not-visited newstate))

          ; mark visited and create new node, stick on succ. list.
          (mark-visited newstate)
          (setf newnode (list newstate
                            (list jug1 jug2) node
                          )))

          (setf succs (cons newnode succs))

        )))

    ; return successors to the search routine
    succs)
;;; results

;setf sux (gen-succs start-node ))
;(((1 0 5) (0 2) ((1 0 0) NIL NIL)) ((1 3 0) (0 1) ((1 0 0) NIL NIL)))

;setf sux1 (gen-succs (car sux )))
;(((1 3 2) (2 1) ((1 0 5) (0 2) ((1 0 0) NIL NIL)) ((1 3 5) (0 1) ((1 0 5) (0 2)

;; seems to make sense. Probably OK to start pushing and poping....