```
;; ======================================
;;  CMPU-365, Fall 2023
;;  Amst. 5 -- Due Thursday, Nov. 9th at Noon
;;  STNs and friends!
;;  Abby, Joram, Sanjae
;; ======================================


;;  STN struct
;; ---------------------------------------

(defstruct (stn (:print-function print-stn))
  ;; NUM-TPS:  The number of timepoints in the network
  (num-tps 0)
  ;; TP-HASH:  A hash-table of (key,value) entries
  ;;   (a) if key is an integer, then value is a symbol naming a timepoint
  ;;   (b) if key is a symbol, then value is the integer index for that timepoint
  (tp-hash (make-hash-table))
  ;; LIST-O-EDGES:  A list of triples, each of the form (FROM WT TO), where
  ;;   FROM and TO are symbols naming timepoints, and WT is a number
  (list-o-edges nil)
  ;; PREDS:  A vector of hash-tables where the ith hash-table contains (key,value)
  ;;         entries where key=j and value=wt specifies an edge from i to j with weight wt
  (preds nil)
  ;; SUCCS:  A vector of hash-tables where the ith hash-table contains (key,value)
  ;;         entries where key=h and value=wt specifies an edge from h to i with weight wt
  (succs nil))

;;  PRINT-STN
;; ------------------------------
;;  INPUTS: S, an STN struct
;;          STR, a stream (usually T)
;;          DEPTH, ignored
;;  OUTPUT: None
;;  Side Effect:  Pretty-prints the STN

(defun print-stn (s str depth)
  (declare (ignore depth))
  (let ((n (stn-num-tps s)))
    (format str "An STN with ~A tps: " n)
    (dotimes (i n) (format str "~A " (gethash i (stn-tp-hash s))))
    (format str "~%")
    (format str "EDGES: ")
    (dolist (trip (stn-list-o-edges s))
      (format str "~A " trip))
    (format str "~%")))

;;  INIT-STN
;; ------------------------------
;;  INPUT:  TP-NAMES, a list of symbols specifying names for the timepoints
;;          LIST-O-TRIPLES, a list of triples of the form (FROM WT TO)
;;             where FROM and TO are symbols representing timepoints
;;             and TO is a number
;;  OUTPUT:  An STN struct suitable initialized

(defun init-stn (tp-names list-o-triples)
  (let* ((s (make-stn))
     (tp-hash (stn-tp-hash s))
     (n (length tp-names))
     (ctr 0))
    (setf (stn-num-tps s) n)
    ;; Initialize the tp-hash
    (dolist (tp-name tp-names)
      ;; First:  key = ctr, value = tp-name
      (setf (gethash ctr tp-hash) tp-name)
      ;; Second: key = tp-name, value = ctr
      (setf (gethash tp-name tp-hash) ctr)
      (incf ctr))
    ;; Create the VECTORS of HASH-TABLES for PREDS and SUCCS
    (setf (stn-preds s) (make-array n))
    (setf (stn-succs s) (make-array n))
    (let ((preds (stn-preds s))
      (succs (stn-succs s)))
      (dotimes (i n)
    (setf (svref preds i) (make-hash-table))
    (setf (svref succs i) (make-hash-table)))
      ;; Walk through the list of edges
      (dolist (trip list-o-triples)
```

```
    (let* ((from (first trip))
           (wt (second trip))
           (to (third trip))
           ;; Get the numerical indices associated with FROM and TO
           (from-indy (gethash from tp-hash))
           (to-indy (gethash to tp-hash)))
      (push trip (stn-list-o-edges s))
      ;; Insert key = from-indy, value = wt into PREDS hash-table for TO-INDY
      (setf (gethash from-indy (svref preds to-indy)) wt)
      ;; Insert key = to-indy, value = wt into SUCCS hash-table for FROM-INDY
      (setf (gethash to-indy (svref succs from-indy)) wt))))
    ;; return the STN
    s))

;;  FW -- Floyd Warshall algorithm
;; ---------------------------------------
;;  INPUT:  S, an STN struct
;;  OUTPUT: DISTY, the distance matrix for S

(defun fw (s)
  (let* ((n (stn-num-tps s))
     (tp-hash (stn-tp-hash s))
     (disty (make-array (list n n)))
     (succs (stn-succs s))
     (preds (stn-preds s)))
    ;; Init disty using the edges in the STN:
    (dotimes (i n)
      (maphash #'(lambda (key value)
           (setf (aref disty i key) value))
           (aref succs i)))
    ;; Triple loop:
    (dotimes (k n)
      (dotimes (i n)
    (when (numberp (aref disty i k))
      (dotimes (j n)
        (let ((val (handle-null-1 (aref disty i k) (aref disty k j) (aref disty i j))))
        (when (numberp val)
          (setf (aref disty i j) val)))
      ))))
    ;; Return DISTY
    disty))

(defun handle-null-1 (ik kj ij)
  (cond
   ;; Case 1: If path is nil
   ((or (null ik) (null kj))
    nil)
   ;; Case 2: If
   ((null ij)
    (+ ik kj))
   ;; Case 3: Both are numbers
   (t
    (if (< (+ ik kj) ij)
    (+ ik kj)
      nil))))

(defun is-soln-for? (soln-veck s)
  (let* ((n (stn-num-tps s))
     (succs (stn-succs s))
     (preds (stn-preds s)))
    (dotimes (i n)
      (let ((val-i (aref soln-veck i)))
    (maphash #'(lambda (k val-k)
             (when (> (- (aref soln-veck k) val-i) val-k)
               (return-from is-soln-for? nil)))
             (aref succs i)))))
    T)

;;  DECK struct
;; ---------------------------
;;  For dealing "cards" at random from a given deck

(defstruct deck
  num-left
  cards)

;;  INIT-DECK
;; ---------------------------
```

```
;;  INPUT:  NUM, the number of cards in the deck
;;  OUTPUT:  A DECK struct that can be used to deal cards from
;;           the set {0,1,...,N-1}

(defun init-deck (num)
  (let ((d (make-deck :num-left num
                      :cards (make-array num))))
    (dotimes (i num)
      (setf (svref (deck-cards d) i) i))
    d))

;;  DEAL!
;; --------------------------
;;  INPUT:  DEK, a DECK struct
;;  OUTPUT:  A card dealt from DEK
;;  Side Effect:  Destructively modifies DEK by putting the dealt
;;    card into a discard pile.

(defun deal! (dek)
  (let* ((num (deck-num-left dek))
     (kardz (deck-cards dek))
     (randy (random num))
     (card (svref kardz randy)))
    ;; Swap chosen card to "discard pile" at end of vector
    (decf (deck-num-left dek))
    (setf (svref kardz randy)
      (svref kardz (deck-num-left dek)))
    (setf (svref kardz (deck-num-left dek))
      card)
    ;; return the "card"
    card
    ))

;;  GET-RANDY-IN-RANGE
;; -----------------------------------
;;  INPUTS:  LOWER, UPPER, either INTEGERS or NIL
;;  OUTPUT:  A number within the interval [LOWER, UPPER], inclusive.
;;  Note:  If either is NIL, limits range to an arbitrary finite interval.

(defun get-randy-in-range (lower upper)
  (cond
   ;; Case 1:  Both are non-NIL (and hence assumed to be numbers)
   ((and lower upper)
    (+ lower (random (1+ (- upper lower)))))
   ;; Case 2:  LOWER is non-NIL (but UPPER is NIL)
   (lower
    ;; Arbitrarily use upper bound of lower+20
    (+ lower (random 20)))
   ;; Case 3:  UPPER is non-NIL (but LOWER is NIL)
   (upper
    ;; Arbitrarily use lower bound of upper-20
    (- upper (random 20)))
   ;; Case 4:  Both are NIL
   (t
    ;; Pick any number in [0,20)
    (random 20))))

;;  GEN-SOLN
;; -----------------------------------
;;  INPUT:   S, an STN
;;  OUTPUT:  A vector representing a solution for S
;;  Note:    If S has N timepoints, the vector will have N entries
;;           interpreted as a mapping from timepoint index to timepoint value

(defun gen-soln (s)
  (let* (;; Use Floyd-Warshall to compute DISTANCE MATRIX
     (disty (fw s))
     (n (stn-num-tps s))
     ;; DEK will be used to "deal out" the next timepoint to execute
     (dek (init-deck n))
     ;; LOWERS and UPPERS are vectors of the lower and upper bounds
     ;; on the timewindows for the N timepoints.  Initially the bounds
     ;; are all NIL, indicating "no bound".
     (uppers (make-array n))
     (lowers (make-array n))
     ;; SOLN:  a vector to hold the timepoint values
     (soln (make-array n)))
    ;; MAIN LOOP
```

```
    ;;   As long as there are "cards" to deal (i.e., timepoints that are
    ;;   not yet executed) ...
    (while (> (deck-num-left dek) 0)
      (let* ((x (deal! dek))
         (lbx (aref lowers x))
         (ubx (aref uppers x))
         (time nil))
    (when (and ubx lbx (< ubx lbx))
      (return-from gen-soln 'Invalid_Bounds))
    (setf time (get-randy-in-range lbx ubx))
    (setf (aref soln x) time)
    (dotimes (i n)
        (let ((dist-u (aref disty x i))
          (dist-l (aref disty i x))
          (lbi (aref lowers i))
          (ubi (aref uppers i)))
          (when (and (numberp dist-l) (or (null lbi) (< lbi (- time dist-l) )))
        (setf (aref lowers i) (- time dist-l)))
           (when (and (numberp dist-u) (or (null ubi) (> ubi (+ time dist-u))))
         (setf (aref uppers i) (+ time dist-u)))))))
    ;; end of WHILE
    soln))

(defun rte (s &key (verbose? nil))
  (let* ((n (stn-num-tps s))
     (preds (stn-preds s))
     (succs (stn-succs s))
     (uppers (make-array n))
     ;; Arbitrarily set lower bounds to 0
     (lowers (make-array n :initial-element 0))
         (soln (make-array n))
     ;; NEG-EDGE-COUNTS:  a vector whose Ith entry holds the number
     ;;   of negative edges emanating from I and pointing at
     ;;   as-yet-unexecuted timepoints
     (neg-edge-counts (make-array n :initial-element 0))
     ;; UNEXECUTED-TPS:  Will hold a list of the as-yet-unexecuted timepoints
     (unexecuted-tps nil)
     ;; NOW:  current time
     (now 0)
     ;; ENABLEDS:  A list of the "enabled" timepoints (i.e., unexecuted
     ;;   timepoints  all of whose negative edges point at already executed
     ;;   timepoints).
     (enableds nil)
     ;; COUNTER:  Just counts the number of iterations in the RTE algorithm
     (counter 0))
    ;; initialize unexecuted-tps
    ;; Initialize NEG-EDGE-COUNTS
    ;; Initialize enableds
    (dotimes (i (length succs))
      (push i unexecuted-tps)
      (maphash (lambda (h wt)
         (when (< wt 0)
           (incf (aref neg-edge-counts i))))
           (aref succs i))
      (when (= (aref neg-edge-counts i) 0)
    (push i enableds)))

    ;; MAIN LOOP
    (while unexecuted-tps
      (when verbose?
    (format t "Starting round ~A with now=~A and enableds=~A ~%" counter now enableds))
      ;; Check for failure
      (when (null enableds)
    (return-from rte 'FAIL_ENABLEDS_EMPTY))
      ;; Find time range [lb, ub] for next execution event
      (let ((lb nil)
        (ub nil))
    (dolist (i enableds)
      (let ((l (aref lowers i))
        (u (aref uppers i)))
        (setf lb (if (null lb) l (min lb l)))
        (setf ub (cond
              ((and ub u)
               (min ub u))
              ((and u (null ub))
               u)))))
    ;; dealing with null, use the arbitrary value used in get-randy-in-range
    (when (null ub)
```

```
      (setf ub 20))
    (when (< ub lb)
      (return-from rte 'FAIL_UB_LT_LB))
    (when (< lb now)
      (setf lb now))
    ;;Create availables
    (let ((availables nil))
      (dolist (entp enableds)
        (let ((bound (aref lowers entp)))
          (when (<= bound ub)
        (push entp availables))))
      (let* ((x (nth (random (length availables)) availables))
         (lb-x (aref lowers x))
         (ub-x (aref uppers x)))
        (when (< lb-x now)
          (setf lb-x now))
        (when (or (null ub-x) (> ub-x ub))
          (setf ub-x ub))
        (when (< ub-x lb-x)
          (return-from rte 'FAIL_UBX_LBX))
        (let ((rand-time (get-randy-in-range lb-x ub-x)))
          (when verbose?
        (format t "  ---> Executing ~A at time ~A~%" counter rand-time))
          (setf unexecuted-tps (remove x unexecuted-tps))
          (setf enableds (remove x enableds))
          (setf (aref soln x) rand-time)
          (setf now rand-time)
          (maphash (lambda (h wt)
             (when (>= wt 0)
               (let ((update (+ rand-time wt)))
                 (when verbose?
                   (format t "  -- Updating uppers(~A) = ~A~%" h update))
                 (setf (aref uppers h) update))))
               (aref succs x))
          (maphash (lambda (h wt)
             (when (< wt 0)
               (let ((update (- rand-time wt)))
                 (when verbose?
                   (format t "  -- Updating lowers(~A) = ~A~%" h update))
                 (setf (aref lowers h) update))
               (setf (aref neg-edge-counts h) (- (aref neg-edge-counts h) 1))
               (when (= (aref neg-edge-counts h) 0)
                 (push h enableds)
                 (when verbose?
                   (format t "      -- ~A has become enabled!~%" h)))))
               (aref preds x))))))
      (setf counter (+ 1 counter)))
    soln))

;; Tests

;; consistent stns
(defvar tp-names2 '(A B C D E))
(defvar edges2 '((A 2 B)
                 (A 8 E)
                 (B 2 C)
                 (C 4 D)
                 (D 5 E)
                 (D -2 A)
                 (D -1 C)
                 (E 1 A)))
(setf stn2 (init-stn tp-names2 edges2))

(defvar tp-names3 '(A B C D E))
(defvar edges3 '((A 4 D)
                 (B -1 A)
                 (D -2 B)
                 (D -2 C)
                 (D 4 E)))
(setf stn3 (init-stn tp-names3 edges3))

; super simple one?
(defvar tp-names4 '(A B C D E))
(defvar edges4 '((A 2 B)
                 (B 1 C)
                 (C 3 D)
                 (D 2 E)
                 (E -4 A)))
```

```
(setf stn4 (init-stn tp-names4 edges4))

(defvar tp-names5 '(A B C D E))
(defvar edges5 '((A 5 C)
                 (B -3 A)
                 (B 2 E)
                 (C 1 B)
                 (D -1 B)
                 (E 4 D)))
(setf stn5 (init-stn tp-names5 edges5))


;; consistent but not dispatchable
(defvar tp-names1 '(A B C D E))
(defvar edges1 '((A 15 E)
                (B -3 A)
                (B 7 E)
                (C -2 B)
                (D -4 C)
                (D 9 E)
                (E -1 D)))
(setf stn1 (init-stn tp-names1 edges1))

(defvar tp-names6 '(A B C D E))
(defvar edges6 '((A 1 B)
                 (A 3 C)
                 (B 3 D)
                 (C 3 E)
                 (D 2 D)))
(setf stn6 (init-stn tp-names6 edges6))
```