ARTIFICIAL INTELLIGENCE

Third Edition

Program Listings

Elaine Rich
Microelectronics and Computer Technology Corporation

Kevin Knight
Carnegie Mellon University

Shivashankar B Nair
IIT Guwahati

Tata McGraw-Hill Publishing Company Limited
NEW DELHI

McGraw-Hill Offices
New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                              A-STAR SEARCH
                              "a-star.lisp"
-----------------------------------------------------------------------------|#

;; ---------------------------------------------------------------------------

(defstruct sn           ; search node
 f
 g
 h
 parent
 children
 state)

(defvar *open*)
(defvar *closed*)

(defvar *nodes-expanded*)


;; ---------------------------------------------------------------------------
;; Functions for manipulating the open and closed lists.  The implementation
;; is straightforward; the open list is kept ordered by the f values of its
;; nodes, smallest first. A more efficient implementation would use priority
;; queues to allow fast insertion and deletion of best nodes.

(defun make-empty-nodelist ()
  nil)

(defun empty-nodelist? (nodelist)
  (null nodelist))

(defun add-to-nodelist (node nodelist)
  (cond ((eq nodelist 'closed)
         (setq *closed* (cons node *closed*)))
        ((eq nodelist 'open)
         (cond ((or (null *open*)
                    (< (sn-f node) (sn-f (car *open*))))
                (setq *open* (cons node *open*)))
               (t
                (do ((x *open*)
                     (y (cdr *open*)))
                    ((or (null y)
                         (< (sn-f node) (sn-f (car y))))
                     (cond ((null y)
                            (setf (cdr x) (list node)))
                           (t
                            (setf (cdr x) (cons node y)))))
                  (setq x y)
                  (setq y (cdr y))))))))

(defun remove-front-of-nodelist (nodelist)
   (cond ((eq nodelist 'open)
          (let ((firstnode (car *open*)))
             (setq *open* (cdr *open*))
             firstnode))
         (t
          (error "Error. Can only remove from open list."))))

(defun find-node-in-nodelist (state nodelist)
  (find-if #'(lambda (n)
               (eq-states state (sn-state n)))
           (if (eq nodelist 'open) *open* *closed*)))

(defun change-priority-of-node (node nodelist)
  (cond ((eq nodelist 'open)
         (setq *open* (delete node *open*))
         (add-to-nodelist node 'open))
        (t
         (error "Error. Can only change priorities of nodes on open list."))))


;; ---------------------------------------------------------------------------
;; Function A-STAR does a heuristic search from the start to the goal state,
;; returning a solution path.  It maintains two lists of nodes, called open
;; and closed, and expands most promising nodes first.

(defun a-star (start &optional verbose)
   (setq *nodes-expanded* 0)
   (setq *open* (make-empty-nodelist))
   (setq *closed* (make-empty-nodelist))
   (let ((start-node (make-sn :state start
                              :g 0
                              :h (heuristic start)
                              :f (heuristic start)
                              :children nil
                              :parent nil)))
     (add-to-nodelist start-node 'open))
   (do ((failure (empty-nodelist? *open*))
        (goal-found nil))
       ((or failure goal-found)
        (if failure "No solution." (extract-a-star-path goal-found)))
      (print (mapcar #'(lambda (s) (list (sn-state s) (sn-f s))) *open*))
      (let ((bestnode (remove-front-of-nodelist 'open)))
        (add-to-nodelist bestnode 'closed)
        (cond ((goal-state?  (sn-state bestnode))
               (setq goal-found bestnode))
              (t (expand-bestnode bestnode verbose))))))


;; ---------------------------------------------------------------------------
;; Function EXPAND-BEST-NODE takes the node with the best heuristic score
;; and expands it.  If the successors are already on the open or closed
;; lists, it updates heuristic scores; otherwise, it adds them to the open
;; list.

(defun expand-bestnode (bestnode verbose)
  (when verbose (format t "Expanding node ~d~%" (sn-state bestnode)))
  (setf *nodes-expanded* (1+ *nodes-expanded*))
  (let ((successors (expand (sn-state bestnode))))
     (do ((s successors (cdr s)))
         ((null s) nil)
       (let* ((succ (car s))
              (old-open (find-node-in-nodelist succ 'open)))
         (cond (old-open                ; step 2c
                (add-child bestnode old-open)
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                        BACKPROPAGATION ALGORITHM
                        (SINGLE, BINARY OUTPUT)
                            "backprop.lisp"
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------

  Backpropagation algorithm (single, binary output)

  To use this program, you must set up three input files: a ".txt",
  a ".train", and a ".test".  Here is an example:

  File xor.txt:

       Network structure:                  (2 2 1)
       Epochs:                             801
       Test after every N epochs:          100
       Learning rate (eta):                0.35
       Momentum (alpha):                   0.90
       Noise?:                             0
       Training data:                      xor.train
       Testing data:                       xor.test

  File xor.train:

       0 0 0
       0 1 1
       1 0 1
       1 1 0

  File xor.test:

       0 0 0
       1 0 1

  The main file contains settings for various parameters.  There are
  separate files for training and testing data.

  To run, call (backprop "xor.txt") (or whatever the filename is).
  The program will periodically append its results to the end of the
  "xor.txt" file.  This version of backpropagation is geared toward
  networks with a single, binary output.  It provides periodic analyses
  of how well the network is predicting the output bit as learning
  progresses.

-----------------------------------------------------------------------|#

;;;
;;; Variables.
;;;

(defvar NETWORK nil)
(defvar OUTPUT-LAYER nil)
(defvar STRUCTURE nil)

(defvar TOTAL-EPOCHS 0)
(defvar TEST-INTERVAL 0)

(defvar TRAINING-INPUTS nil)
(defvar TRAINING-OUTPUTS nil)
(defvar TESTING-INPUTS nil)
(defvar TESTING-OUTPUTS nil)

(defvar TOTAL-TRAINING nil)
(defvar TOTAL-TESTING nil)

(defvar TOTAL-INPUTS nil)
(defvar TOTAL-OUTPUTS nil)

(defvar ETA 0)
(defvar ALPHA 0)
(defvar NOISE 0)

(defvar TRAIN-FILE nil)
(defvar TEST-FILE nil)

(defvar TOTAL-GUESSED (make-array 8 :element-type 'float))
(defvar TOTAL-RIGHT (make-array 8 :element-type 'float))

;;;
;;; Structures.
;;;

(defstruct (unit)
  (weighted-sum 'float)
  (activation 'float) (delta 'float))
(defstruct (net)
  units connections size next-layer prev-layer)
(defstruct (connection)
  (weight 'float) (delta-weight 'float))

(defun output-layer? (layer)
  (null (net-next-layer layer)))
(defun input-layer? (layer)
  (null (net-prev-layer layer)))
(defun hidden-layer? (layer)
  (and (net-next-layer layer) (net-prev-layer layer)))

;;;
;;; Building the network.
;;;

(defun random-real (lo hi) (+ lo (random (- hi lo))))
(defun random-weight () (random-real -0.8 0.8))
(defun random-noise () (random-real 0.0 0.15))

(defun construct-units ()
  (do ((n STRUCTURE (cdr n)) (last-layer nil) (temp-net nil))
      ((null n) (setf OUTPUT-LAYER last-layer))
    (setf temp-net
          (make-net :units (make-array (1+ (car n)) :element-type 'unit)
                    :connections
                      (if (cdr n)
                          (make-array (list (1+ (car n)) (1+ (cadr n)))
                                      :element-type 'connection)
                        nil)
```

```
                (possibly-change-parent old-open bestnode 'open))
               (t
                (let ((old-closed (find-node-in-nodelist succ 'closed)))
                  (cond (old-closed    ;  step 2d
                         (add-child bestnode old-closed)
                         (possibly-change-parent old-closed bestnode
                                                 'closed))
                        (t
                         (let ((new-node (make-sn :state succ
                                                  :g (+ (sn-g bestnode)
                                                        (cost-of-move
                                                            (sn-state bestnode)
                                                            succ))
                                                  :h (heuristic succ)
                                                  :parent bestnode
                                                  :children nil)))
                           (setf (sn-f new-node) (+ (sn-g new-node)
                                                    (sn-h new-node)))
                           (add-child bestnode new-node)
                           (add-to-nodelist new-node 'open)))))))))


;; ---------------------------------------------------------------------------
;; Function ADD-CHILD modifies a node to have a new successor.

(defun add-child (parent child)
   (setf (sn-children parent) (cons child (sn-children parent))))


;; ---------------------------------------------------------------------------
;; Function POSSIBLY-CHANGE-PARENT takes a new node that is already on the
;; open or closed lists, and changes the old node's g, f, and parent fields
;; if the new path is shorter than the old one.

(defun possibly-change-parent (old bestnode nodelist)
  (let ((old-cost (sn-g old))
        (cost-through-bestnode (+ (sn-g bestnode)
                                  (cost-of-move (sn-state bestnode)
                                                (sn-state old)))))
    (when (> old-cost cost-through-bestnode)
          (setf (sn-parent old) bestnode)
          (setf (sn-g old) cost-through-bestnode)
          (setf (sn-f old) (+ (sn-g old) (sn-h old)))
          (cond ((eq nodelist 'open)
                 (change-priority-of-node old 'open))
                ((eq nodelist 'closed)
                 (propagate-cost old))))))

;; ---------------------------------------------------------------------------
;; Function PROPAGATE-COST takes a node and ensures that the heuristic
;; estimates of its descendant nodes are accurate.

(defun propagate-cost (old)
    (do ((children (sn-children old) (cdr children)))
        ((null children) nil)
      (let ((child (car children)))
        (when (or (eq old (sn-parent child))
                  (< (+ (sn-g old)
                        (cost-of-move (sn-state old)
                                      (sn-state child)))
                     (sn-g child)))
          (setf (sn-g child) (+ (sn-g old) 1))
          (setf (sn-f child) (+ (sn-g child) (sn-h child)))
          (setf (sn-parent child) old)
          (when (member child *open*)
              (change-priority-of-node child *open*))
          (propagate-cost child))))))


;; ---------------------------------------------------------------------------
;; Function EXTRACT-PATH retrieves a solution path by tracing parent pointers
;; of a node.

(defun extract-a-star-path (node)
  (do ((path nil)
       (n node (sn-parent n)))
      ((null n) path)
    (setq path (cons (sn-state n) path))))
```

```
;;;
;;; Multiplication.
;;;

(defun mult (&rest args)
;  (without-floating-underflow-traps (apply #'* args)))
  (apply #'* args))

;;;
;;; Feed Forward phase.
;;;

(defun feed-forward (index set verbose)
  (when verbose (format t "Feeding vector ~d from ~d into network ...~%"
                        index (if (eq set TRAINING-INPUTS) 0 1)))
  (do ((u 1 (1+ u))) ((> u (net-size NETWORK)) nil)
    (let* ((noise-added (if (= NOISE 1) (random-noise) 0.0))
           (this-input (aref set index u))
           (input-plus-noise (if (> this-input 0.5)
                                 (- this-input noise-added)
                                 (+ this-input noise-added))))
      (setf (unit-activation (aref (net-units NETWORK) u)) input-plus-noise)))
  (do ((layer NETWORK (net-next-layer layer)))
      ((output-layer? layer) nil)
    (do ((u1 1 (1+ u1))) ((> u1 (net-size (net-next-layer layer))) nil)
      (let ((this-unit (aref (net-units (net-next-layer layer)) u1)))
        (setf (unit-weighted-sum this-unit) 0.0)
        (do ((u2 0 (1+ u2))) ((> u2 (net-size layer)) nil)
          (setf (unit-weighted-sum this-unit)
                (+ (unit-weighted-sum this-unit)
                   (mult (unit-activation (aref (net-units layer) u2))
                         (connection-weight
                           (aref (net-connections layer) u2 u1))))))
        (setf (unit-activation this-unit)
              (activation-function (unit-weighted-sum this-unit))))))
  (when verbose (format t "Result = ~14,7f ...~%"
                  (unit-activation (aref (net-units OUTPUT-LAYER) 1)))))

;;;
;;; Back Propagate phase.
;;;

(defun back-propagate (index temp-alpha verbose)
  (when verbose (format t "Backpropagating errors ...~%"))
  (do ((u 1 (1+ u))) ((> u (net-size OUTPUT-LAYER)) nil)
    (let ((this-unit (aref (net-units OUTPUT-LAYER) u)))
      (setf (unit-delta this-unit)
            (mult (- (aref TRAINING-OUTPUTS index u)
                     (unit-activation this-unit))
                  (unit-activation this-unit)
                  (- 1.0 (unit-activation this-unit))))))
  (do ((layer (net-prev-layer OUTPUT-LAYER) (net-prev-layer layer)))
      ((input-layer? layer) nil)
    (do ((u 0 (1+ u))) ((> u (net-size layer)))
      (let ((this-unit (aref (net-units layer) u)) (sum 0.0))
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))))
          (setf sum
                (+ sum (mult (unit-delta
                               (aref (net-units (net-next-layer layer)) u2))
                             (connection-weight
                               (aref (net-connections layer) u u2))))))
        (setf (unit-delta this-unit)
              (mult (- 1.0 (unit-activation this-unit))
                    (unit-activation this-unit) sum)))))
  (do ((layer (net-prev-layer OUTPUT-LAYER) (net-prev-layer layer))
       (l 1 (1+ l)))
      ((null layer) nil)
    (do ((u 0 (1+ u))) ((> u (net-size layer)) nil)
      (let ((low-unit (aref (net-units layer) u)))
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))) nil)
          (let* ((hi-unit (aref (net-units (net-next-layer layer)) u2))
                 (the-connection (aref (net-connections layer) u u2))
                 (newchange
                   (+ (mult ETA (unit-delta hi-unit)
                                (unit-activation low-unit))
                      (mult temp-alpha
                            (connection-delta-weight the-connection)))))
            (when verbose
              (format t "Changing weight (~d ~d ~d) from ~14,7f to ~14,7f~%"
                  l u u2 (connection-weight the-connection)
                  (+ (connection-weight the-connection) newchange)))
            (setf (connection-weight the-connection)
                  (+ (connection-weight the-connection) newchange))
            (setf (connection-delta-weight the-connection) newchange)))))))

;;;
;;; Learn.
;;;

(defun learn (infile verbose)
  (dotimes (epoch TOTAL-EPOCHS)
    (when verbose (format t "Starting epoch ~d ...~%" epoch))
    (let ((temp-alpha (if (< epoch 10) 0.0 ALPHA)))
      (when (= 0 (mod epoch TEST-INTERVAL))
        (evaluate-progress epoch infile) )
      (do ((x 1 (1+ x))) ((> x TOTAL-TRAINING))
        (feed-forward x TRAINING-INPUTS verbose)
        (back-propagate x temp-alpha verbose)))))

;;;
;;; Evaluate Progress.
;;;

(defun evaluate-progress (epoch infile)
  (evaluate-training-data epoch infile)
  (evaluate-testing-data epoch infile))

(defun evaluate-training-data (epoch infile)
  (do ((i 0 (1+ i))) ((= i 8)) (setf (aref TOTAL-GUESSED i) 0.0))
  (do ((i 0 (1+ i))) ((= i 8)) (setf (aref TOTAL-RIGHT i) 0.0))
  (do ((i 1 (1+ i))) ((> i TOTAL-TRAINING))
    (let ((right (aref TRAINING-OUTPUTS i 1)))
      (feed-forward i TRAINING-INPUTS nil)
      (let ((output-activation
              (unit-activation (aref (net-units OUTPUT-LAYER) 1))))
        (do ((j 0 (1+ j))) ((= j 8))
          (when (or (> output-activation (+ 0.5 (* 0.05 j)))
                    (< output-activation (- 0.5 (* 0.05 j))))
            (setf (aref TOTAL-GUESSED j) (+ 1.0 (aref TOTAL-GUESSED j)))
            (when (or (and (> right 0.5)
                           (> output-activation (+ 0.5 (* 0.05 j))))
                      (and (< right 0.5)
                           (< output-activation (- 0.5 (* 0.05 j)))))
              (setf (aref TOTAL-RIGHT j) (+ 1.0 (aref TOTAL-RIGHT j))))))))))
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "EPOCH ~d.  Performance on training data:~%~%" epoch)
    (format ifile "Confidence: ")
```

```
                    :prev-layer last-layer))
    (do ((u 0 (1+ u))) ((> u (car n)) nil)
      (setf (aref (net-units temp-net) u) (make-unit)))
    (when (cdr n)
      (do ((u1 0 (1+ u1))) ((> u1 (car n)) nil)
        (do ((u2 1 (1+ u2))) ((> u2 (cadr n)) nil)
          (setf (aref (net-connections temp-net) u1 u2) (make-connection)))))
    (cond ((= (length n) (length STRUCTURE))
           (setf NETWORK temp-net)
           (setf (net-size temp-net) (car n))
           (setf last-layer temp-net))
          (t
           (setf (net-next-layer last-layer) temp-net)
           (setf (net-size temp-net) (car n))
           (setf last-layer temp-net)))))

(defun set-initial-weights ()
  (do ((layer NETWORK (net-next-layer layer))) ((null layer) nil)
    (let ((size (net-size layer))
          (units (net-units layer))
          (connections (net-connections layer)))
      (do ((u 0 (1+ u))) ((> u size) nil)
        (setf (unit-activation (aref units u)) 0)
        (setf (unit-weighted-sum (aref units u)) 0)
        (setf (unit-delta (aref units u)) 0))
      (when (not (output-layer? layer))
        (let ((next-layer-size (net-size (net-next-layer layer))))
          (do ((u1 0 (1+ u1))) ((> u1 size) nil)
            (do ((u2 1 (1+ u2))) ((> u2 next-layer-size) nil)
              (setf (connection-weight (aref connections u1 u2))
                    (random-weight))
              (setf (connection-delta-weight (aref connections u1 u2))
                    0.0)))))))
  (do ((layer NETWORK (net-next-layer layer))) ((null layer) nil)
    (setf (unit-activation (aref (net-units layer) 0)) 1.0)))

(defun build-network ()
  (setf TOTAL-INPUTS (car STRUCTURE))
  (setf TOTAL-OUTPUTS (car (last STRUCTURE)))
  (construct-units)
  (set-initial-weights))

;;;
;;; Building the test and train sets.
;;;

(defun build-test-and-train-sets ()
  (setf TOTAL-TRAINING 0)
  (with-open-file (ifile TRAIN-FILE :direction :input)
    (do ((x (read-line ifile nil 'error) (read-line ifile nil 'error)) (tot 0))
        ((or (string-equal "" x) (eq x 'error)) (setf TOTAL-TRAINING tot))
      (when (not (string-equal "" x)) (setf tot (1+ tot)))))
  (setf TOTAL-TESTING 0)
  (with-open-file (ifile TEST-FILE :direction :input)
    (do ((x (read-line ifile nil 'error) (read-line ifile nil 'error)) (tot 0))
        ((or (string-equal "" x) (eq x 'error)) (setf TOTAL-TESTING tot))
      (when (not (string-equal "" x)) (setf tot (1+ tot)))))
  (setf TRAINING-INPUTS
        (make-array (list (1+ TOTAL-TRAINING) (1+ TOTAL-INPUTS))
          :initial-element 0.0))
  (setf TRAINING-OUTPUTS
        (make-array (list (1+ TOTAL-TRAINING) (1+ TOTAL-OUTPUTS))
          :initial-element 0.0))
  (setf TESTING-INPUTS
        (make-array (list (1+ TOTAL-TESTING) (1+ TOTAL-INPUTS))
          :initial-element 0.0))
  (setf TESTING-OUTPUTS
        (make-array (list (1+ TOTAL-TESTING) (1+ TOTAL-OUTPUTS))
          :initial-element 0.0))
  (with-open-file (ifile TRAIN-FILE :direction :input)
    (do ((x 1 (1+ x))) ((> x TOTAL-TRAINING) nil)
      (do ((y 1 (1+ y))) ((> y TOTAL-INPUTS) nil)
        (setf (aref TRAINING-INPUTS x y) (read ifile)))
      (do ((y 1 (1+ y))) ((> y TOTAL-OUTPUTS) nil)
        (setf (aref TRAINING-OUTPUTS x y) (bound-outputs (read ifile))))))
  (with-open-file (ifile TEST-FILE :direction :input)
    (do ((x 1 (1+ x))) ((> x TOTAL-TESTING) nil)
      (do ((y 1 (1+ y))) ((> y TOTAL-INPUTS) nil)
        (setf (aref TESTING-INPUTS x y) (read ifile)))
      (do ((y 1 (1+ y))) ((> y TOTAL-OUTPUTS) nil)
        (setf (aref TESTING-OUTPUTS x y) (bound-outputs (read ifile)))))))

;; Function BOUND-OUTPUTS takes a target output from a testing or training
;; file, and forces it to be at least 0.1 and at most 0.9.

(defun bound-outputs (x)
  (cond ((> x 0.9) 0.9)
        ((< x 0.1) 0.1)
        (t x)))

(defun init-network (verbose)
  (when verbose (format t "Building network ...~%"))
  (build-network)
  (when verbose (format t "Building test and train sets ...~%"))
  (build-test-and-train-sets))

;;;
;;; Reading the input file.
;;;

(defun read-data (infile verbose)
  (when verbose (format t "Reading input data ...~%"))
  (with-open-file (ifile infile :direction :input)
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf STRUCTURE (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TOTAL-EPOCHS (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TEST-INTERVAL (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf ETA (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf ALPHA (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf NOISE (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TRAIN-FILE (delete #\Tab (delete #\Space (read-line ifile))))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TEST-FILE (delete #\Tab (delete #\Space (read-line ifile))))))

;;;
;;; Activation function.
;;;

(defvar zero-float (coerce 0.0 'float))

(defun activation-function (sum)
  (/ 1.0 (+ 1.0 (exp (- zero-float sum)))))
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                        BACKPROPAGATION ALGORITHM
                            (MULTIPLE OUTPUTS)
                            "backprop2.lisp"
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------

   Backpropagation algorithm (multiple outputs)

   To use this program, you must set up three input files: a ".txt",
   a ".train", and a ".test".  Here is an example:

   File xor.txt:

        Network structure:                      (2 2 1)
        Epochs:                                 801
        Test after every N epochs:              100
        Learning rate (eta):                    0.35
        Momentum (alpha):                       0.90
        Noise?:                                 0
        Training data:                          xor.train
        Testing data:                           xor.test

   File xor.train:

        0 0 0
        0 1 1
        1 0 1
        1 1 0

   File xor.test:

        0 0 0
        1 0 1

   The main file contains settings for various parameters.  There
   are separate files for training and testing data.

   To run, call (backprop "xor.txt") (or whatever the filename is).
   The program will append periodically its results to the end of
   the "xor.txt" file.

   This version of backprop deals with multiple output values.
   Instead of calculating statistics on how well the net predicts
   the output bit, this program prints out each input vector with
   its actual and target output vectors.

-----------------------------------------------------------------------|#


;;;
;;; Variables.
;;;

(defvar NETWORK nil)
(defvar OUTPUT-LAYER nil)
(defvar STRUCTURE nil)

(defvar TOTAL-EPOCHS 0)
(defvar TEST-INTERVAL 0)

(defvar TRAINING-INPUTS nil)
(defvar TRAINING-OUTPUTS nil)
(defvar TESTING-INPUTS nil)
(defvar TESTING-OUTPUTS nil)

(defvar TOTAL-TRAINING nil)
(defvar TOTAL-TESTING nil)

(defvar TOTAL-INPUTS nil)
(defvar TOTAL-OUTPUTS nil)

(defvar ETA 0)
(defvar ALPHA 0)
(defvar NOISE 0)

(defvar TRAIN-FILE nil)
(defvar TEST-FILE nil)

;;;
;;; Structures.
;;;

(defstruct (unit)
  (weighted-sum 'float)
  (activation 'float) (delta 'float))
(defstruct (net)
  units connections size next-layer prev-layer)
(defstruct (connection)
  (weight 'float) (delta-weight 'float))

(defun output-layer? (layer)
  (null (net-next-layer layer)))
(defun input-layer? (layer)
  (null (net-prev-layer layer)))
(defun hidden-layer? (layer)
  (and (net-next-layer layer) (net-prev-layer layer)))

;;;
;;; Building the network.
;;;

(defun random-real (lo hi) (+ lo (random (- hi lo))))
(defun random-weight () (random-real -0.1 0.1))
(defun random-noise () (random-real 0.0 0.15))

(defun construct-units ()
  (do ((n STRUCTURE (cdr n)) (last-layer nil) (temp-net nil))
      ((null n) (setf OUTPUT-LAYER last-layer))
    (setf temp-net
          (make-net :units (make-array (1+ (car n)) :element-type 'unit)
                    :connections
                      (if (cdr n)
                          (make-array (list (1+ (car n)) (1+ (cadr n)))
                                      :element-type 'connection)
                        nil)
```

```
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6,2f " (* i 0.05)))
    (format ifile "~%")
    (format ifile "Guessed:    ")
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6d "
      (round (aref TOTAL-GUESSED i))))
    (format ifile "~%")
    (format ifile "Correct:    ")
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6d "
      (round (aref TOTAL-RIGHT i))))
    (format ifile "~%")
    (format ifile "Percent:    ")
    (do ((i 0 (1+ i))) ((= i 8))
      (format ifile "~6,2f "
              (if (> (aref TOTAL-GUESSED i) 0.5)
                  (/ (* 100.0 (aref TOTAL-RIGHT i)) (aref TOTAL-GUESSED i))
                  0.0)))
    (format ifile "~%~%")))

(defun evaluate-testing-data (epoch infile)
  (do ((i 0 (1+ i))) ((= i 8)) (setf (aref TOTAL-GUESSED i) 0.0))
  (do ((i 0 (1+ i))) ((= i 8)) (setf (aref TOTAL-RIGHT i) 0.0))
  (do ((i 1 (1+ i))) ((> i TOTAL-TESTING))
    (let ((right (aref TESTING-OUTPUTS i 1)))
      (feed-forward i TESTING-INPUTS nil)
      (let ((output-activation
             (unit-activation (aref (net-units OUTPUT-LAYER) 1))))
        (do ((j 0 (1+ j))) ((= j 8))
          (when (or (> output-activation (+ 0.5 (* 0.05 j)))
                    (< output-activation (- 0.5 (* 0.05 j))))
            (setf (aref TOTAL-GUESSED j) (+ 1.0 (aref TOTAL-GUESSED j)))
            (when (or (and (> right 0.5)
                           (> output-activation (+ 0.5 (* 0.05 j))))
                      (and (< right 0.5)
                           (< output-activation (- 0.5 (* 0.05 j)))))
              (setf (aref TOTAL-RIGHT j) (+ 1.0 (aref TOTAL-RIGHT j)))))))))))
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "EPOCH ~d.  Performance on testing data:~%~%" epoch)
    (format ifile "Confidence: ")
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6,2f " (* i 0.05)))
    (format ifile "~%")
    (format ifile "Guessed:    ")
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6d "
      (round (aref TOTAL-GUESSED i))))
    (format ifile "~%")
    (format ifile "Correct:    ")
    (do ((i 0 (1+ i))) ((= i 8)) (format ifile "~6d "
      (round (aref TOTAL-RIGHT i))))
    (format ifile "~%")
    (format ifile "Percent:    ")
    (do ((i 0 (1+ i))) ((= i 8))
      (format ifile "~6,2f "
              (if (> (aref TOTAL-GUESSED i) 0.5)
                  (/ (* 100.0 (aref TOTAL-RIGHT i)) (aref TOTAL-GUESSED i))
                  0.0)))
    (format ifile "~%~%")))

;;;
;;; Print Weights.
;;;

(defun print-weights (infile)
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "~%~%Weights:~%~%")
    (do ((layer NETWORK (net-next-layer layer)) (k 1 (1+ k)))
        ((output-layer? layer) nil)
      (do ((u 0 (1+ u))) ((> u (net-size layer)) nil)
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))) nil)
          (let ((the-connection (aref (net-connections layer) u u2)))
            (format ifile "(~d ~d ~d ~4.7f)~%"
                    k u u2 (connection-weight the-connection)))))))))

(defun print-weights-to-screen ()
    (format t "~%~%Weights:~%~%")
    (do ((layer NETWORK (net-next-layer layer)) (k 1 (1+ k)))
        ((output-layer? layer) nil)
      (do ((u 0 (1+ u))) ((> u (net-size layer)) nil)
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))) nil)
          (let ((the-connection (aref (net-connections layer) u u2)))
            (format t "~d ~d ~d~%"
                    (list k u u2 (connection-weight the-connection))))))))

;;;
;;; Backprop.
;;;

(defun backprop (infile &optional verbose)
  (read-data infile verbose)
  (init-network verbose)
  (learn infile verbose)
  (print-weights infile))
```

```
;;; Multiplication.
;;;

(defun mult (&rest args)
;  (without-floating-underflow-traps (apply #'* args)))
  (apply #'* args))


;;;
;;; Feed Forward phase.
;;;

(defun feed-forward (index set)
  (do ((u 1 (1+ u))) ((> u (net-size NETWORK)) nil)
    (let* ((noise-added (if (= NOISE 1) (random-noise) 0.0))
           (this-input (aref set index u))
           (input-plus-noise (if (> this-input 0.5)
                                 (- this-input noise-added)
                                 (+ this-input noise-added))))
      (setf (unit-activation (aref (net-units NETWORK) u)) input-plus-noise)))
  (do ((layer NETWORK (net-next-layer layer)))
      ((output-layer? layer) nil)
    (do ((u1 1 (1+ u1))) ((> u1 (net-size (net-next-layer layer))) nil)
      (let ((this-unit (aref (net-units (net-next-layer layer)) u1)))
        (setf (unit-weighted-sum this-unit) 0.0)
        (do ((u2 0 (1+ u2))) ((> u2 (net-size layer)) nil)
          (setf (unit-weighted-sum this-unit)
                (+ (unit-weighted-sum this-unit)
                   (mult (unit-activation (aref (net-units layer) u2))
                         (connection-weight
                           (aref (net-connections layer) u2 u1))))))
        (setf (unit-activation this-unit)
              (activation-function (unit-weighted-sum this-unit)))))))

;;;
;;; Back Propagate phase.
;;;

(defun back-propagate (index temp-alpha)
  (do ((u 1 (1+ u))) ((> u (net-size OUTPUT-LAYER)) nil)
    (let ((this-unit (aref (net-units OUTPUT-LAYER) u)))
      (setf (unit-delta this-unit)
            (mult (- (aref TRAINING-OUTPUTS index u)
                     (unit-activation this-unit))
                  (unit-activation this-unit)
                  (- 1.0 (unit-activation this-unit))))))
  (do ((layer (net-prev-layer OUTPUT-LAYER) (net-prev-layer layer)))
      ((input-layer? layer) nil)
    (do ((u 0 (1+ u))) ((> u (net-size layer)))
      (let ((this-unit (aref (net-units layer) u)) (sum 0.0))
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))))
          (setf sum
                (+ sum (mult (unit-delta
                               (aref (net-units (net-next-layer layer)) u2))
                             (connection-weight
                               (aref (net-connections layer) u u2))))))
        (setf (unit-delta this-unit)
              (mult (- 1.0 (unit-activation this-unit))
                    (unit-activation this-unit) sum)))))
  (do ((layer (net-prev-layer OUTPUT-LAYER) (net-prev-layer layer)))
      ((null layer) nil)
    (do ((u 0 (1+ u))) ((> u (net-size layer)) nil)
      (let ((low-unit (aref (net-units layer) u)))
        (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))) nil)
          (let* ((hi-unit (aref (net-units (net-next-layer layer)) u2))
                 (the-connection (aref (net-connections layer) u u2))
                 (newchange
                   (+ (mult ETA (unit-delta hi-unit)
                            (unit-activation low-unit))
                      (mult temp-alpha
                            (connection-delta-weight the-connection)))))
            (setf (connection-weight the-connection)
                  (+ (connection-weight the-connection) newchange))
            (setf (connection-delta-weight the-connection) newchange)))))))

;;;
;;; Learn.
;;;

(defun learn (infile)
  (dotimes (epoch TOTAL-EPOCHS)
    (let ((temp-alpha (if (< epoch 10) 0.0 ALPHA)))
      (when (= 0 (mod epoch TEST-INTERVAL))
        (evaluate-progress epoch infile))
      (do ((x 1 (1+ x))) ((> x TOTAL-TRAINING))
        (feed-forward x TRAINING-INPUTS)
        (back-propagate x temp-alpha)))))

;;;
;;; Evaluate Progress.
;;;

(defun evaluate-progress (epoch infile)
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "~%Epoch ~d.~%~%" epoch)
    (format ifile "Training.~%"))
  (evaluate-training-data epoch infile)
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "Testing.~%"))
  (evaluate-testing-data epoch infile))

(defun evaluate-training-data (epoch infile)
  (declare (ignore epoch))
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (do ((i 1 (1+ i))) ((> i TOTAL-TRAINING))
      (feed-forward i TRAINING-INPUTS)
      (do ((j 1 (1+ j))) ((> j TOTAL-INPUTS) nil)
        (format ifile "~4,1f" (aref TRAINING-INPUTS i j)))
      (format ifile " --> ")
      (do ((j 1 (1+ j))) ((> j TOTAL-OUTPUTS) nil)
        (format ifile "~3,1f "
          (unit-activation (aref (net-units OUTPUT-LAYER) j))))
      (format ifile "  (target =")
      (do ((j 1 (1+ j))) ((> j TOTAL-OUTPUTS) nil)
        (format ifile "~4,1f" (aref TRAINING-OUTPUTS i j)))
      (format ifile ")~%"))))

(defun evaluate-testing-data (epoch infile)
  (declare (ignore epoch))
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (do ((i 1 (1+ i))) ((> i TOTAL-TESTING))
      (feed-forward i TESTING-INPUTS)
      (do ((j 1 (1+ j))) ((> j TOTAL-INPUTS) nil)
        (format ifile "~4,1f" (aref TESTING-INPUTS i j)))
      (format ifile " --> ")
      (do ((j 1 (1+ j))) ((> j TOTAL-OUTPUTS) nil)
        (format ifile "~3,1f "
          (unit-activation (aref (net-units OUTPUT-LAYER) j))))
```

```
                    :prev-layer last-layer))
    (do ((u 0 (1+ u))) ((> u (car n)) nil)
      (setf (aref (net-units temp-net) u) (make-unit)))
    (when (cdr n)
      (do ((u1 0 (1+ u1))) ((> u1 (car n)) nil)
        (do ((u2 1 (1+ u2))) ((> u2 (cadr n)) nil)
          (setf (aref (net-connections temp-net) u1 u2) (make-connection)))))
    (cond ((= (length n) (length STRUCTURE))
           (setf NETWORK temp-net)
           (setf (net-size temp-net) (car n))
           (setf last-layer temp-net))
          (t
           (setf (net-next-layer last-layer) temp-net)
           (setf (net-size temp-net) (car n))
           (setf last-layer temp-net)))))

(defun set-initial-weights ()
  (do ((layer NETWORK (net-next-layer layer))) ((null layer) nil)
    (let ((size (net-size layer))
          (units (net-units layer))
          (connections (net-connections layer)))
      (do ((u 0 (1+ u))) ((> u size) nil)
        (setf (unit-activation (aref units u)) 0)
        (setf (unit-weighted-sum (aref units u)) 0)
        (setf (unit-delta (aref units u)) 0))
      (when (not (output-layer? layer))
        (let ((next-layer-size (net-size (net-next-layer layer))))
          (do ((u1 0 (1+ u1))) ((> u1 size) nil)
            (do ((u2 1 (1+ u2))) ((> u2 next-layer-size) nil)
              (setf (connection-weight (aref connections u1 u2))
                    (random-weight))
              (setf (connection-delta-weight (aref connections u1 u2))
                    0.0)))))))
  (do ((layer NETWORK (net-next-layer layer))) ((null layer) nil)
    (setf (unit-activation (aref (net-units layer) 0)) 1.0)))

(defun build-network ()
  (setf TOTAL-INPUTS (car STRUCTURE))
  (setf TOTAL-OUTPUTS (car (last STRUCTURE)))
  (construct-units)
  (set-initial-weights))

;;;
;;; Building the test and train sets.
;;;

(defun build-test-and-train-sets ()
  (setf TOTAL-TRAINING 0)
  (with-open-file (ifile TRAIN-FILE :direction :input)
    (do ((x (read-line ifile nil 'error) (read-line ifile nil 'error)) (tot 0))
        ((or (string-equal "" x) (eq x 'error)) (setf TOTAL-TRAINING tot))
      (when (not (string-equal "" x)) (setf tot (1+ tot)))))
  (setf TOTAL-TESTING 0)
  (with-open-file (ifile TEST-FILE :direction :input)
    (do ((x (read-line ifile nil 'error) (read-line ifile nil 'error)) (tot 0))
        ((or (string-equal "" x) (eq x 'error)) (setf TOTAL-TESTING tot))
      (when (not (string-equal "" x)) (setf tot (1+ tot)))))
  (setf TRAINING-INPUTS
        (make-array (list (1+ TOTAL-TRAINING) (1+ TOTAL-INPUTS))
          :initial-element 0.0))
  (setf TRAINING-OUTPUTS
        (make-array (list (1+ TOTAL-TRAINING) (1+ TOTAL-OUTPUTS))
          :initial-element 0.0))
  (setf TESTING-INPUTS
        (make-array (list (1+ TOTAL-TESTING) (1+ TOTAL-INPUTS))
          :initial-element 0.0))
  (setf TESTING-OUTPUTS
        (make-array (list (1+ TOTAL-TESTING) (1+ TOTAL-OUTPUTS))
          :initial-element 0.0))
  (with-open-file (ifile TRAIN-FILE :direction :input)
    (do ((x 1 (1+ x))) ((> x TOTAL-TRAINING) nil)
      (do ((y 1 (1+ y))) ((> y TOTAL-INPUTS) nil)
        (setf (aref TRAINING-INPUTS x y) (read ifile)))
      (do ((y 1 (1+ y))) ((> y TOTAL-OUTPUTS) nil)
        (setf (aref TRAINING-OUTPUTS x y) (bound-outputs (read ifile))))))
  (with-open-file (ifile TEST-FILE :direction :input)
    (do ((x 1 (1+ x))) ((> x TOTAL-TESTING) nil)
      (do ((y 1 (1+ y))) ((> y TOTAL-INPUTS) nil)
        (setf (aref TESTING-INPUTS x y) (read ifile)))
      (do ((y 1 (1+ y))) ((> y TOTAL-OUTPUTS) nil)
        (setf (aref TESTING-OUTPUTS x y) (bound-outputs (read ifile)))))))


;; Function BOUND-OUTPUTS takes a target output from a testing or training
;; file, and forces it to be at least 0.1 and at most 0.9.

(defun bound-outputs (x)
  (cond ((> x 0.9) 0.9)
        ((< x 0.1) 0.1)
        (t x)))

(defun init-network ()
  (build-network)
  (build-test-and-train-sets))

;;;
;;; Reading the input file.
;;;

(defun read-data (infile)
  (with-open-file (ifile infile :direction :input)
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf STRUCTURE (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TOTAL-EPOCHS (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TEST-INTERVAL (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf ETA (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf ALPHA (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf NOISE (read ifile))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TRAIN-FILE (delete #\Tab (delete #\Space (read-line ifile))))
    (do ((x (read-char ifile) (read-char ifile))) ((equal x #\:) nil) nil)
    (setf TEST-FILE (delete #\Tab (delete #\Space (read-line ifile))))))

;;;
;;; Activation function.
;;;

(defvar zero-float (coerce 0.0 'float))

(defun activation-function (sum)
  (/ 1.0 (+ 1.0 (exp (- zero-float sum)))))

;;;
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                        BREADTH-FIRST SEARCH
                            "bfs.lisp"
-----------------------------------------------------------------------|#

;; ---------------------------------------------------------------------
;; Structure BFS-NODE stores nodes of the search tree. Every node in the
;; breadth-first search has two components: the state and the parent of
;; the node.

(defstruct bfs-node state parent)

;; ---------------------------------------------------------------------
;; Function BFS performs a breadth-first search and returns a solution
;; path from start to goal.  Nodes on the frontier of the search space
;; are kept on NODE-LIST.  When a node is expanded, its successors are
;; appended to the back of NODE-LIST.
;;
;; The versions of breadth-first search in this file do not take edge
;; costs into account -- i.e., they expand the search space as though
;; the cost of moving from one state to another were constant.

(defun bfs (start &optional verbose)
  (let ((node-list (list (make-bfs-node :state start :parent nil))))
    (do* ((node (car node-list) (car node-list))
          (solution-found (if (goal-state? start) node nil)))
         ((or (null node-list) solution-found)
          (if (null node-list)
              "No solution."
              (extract-bfs-path solution-found)))
      (when verbose (format t "Expanding node ~d~%" (bfs-node-state node)))
      (let ((succs (mapcar #'(lambda (e)
                               (make-bfs-node :state e
                                              :parent node))
                           (expand (bfs-node-state node)))))
        (setq solution-found (find-if #'(lambda (s)
                                          (goal-state? (bfs-node-state s)))
                                      succs))
        (setq node-list (append (cdr node-list) succs))))))

;; ---------------------------------------------------------------------
;; Function EXTRACT-BFS-PATH takes a goal node and follows its parent
;; pointers back to the initial node, returning the list of all states
;; along the path.

(defun extract-bfs-path (node)
  (do ((path nil)
       (n node (bfs-node-parent n)))
      ((null n) path)
    (setq path (cons (bfs-node-state n) path))))

;; ---------------------------------------------------------------------
;; Function BFS-GRAPH is the same as BFS except duplicate nodes are not
;; added to the search space.  That is, the same state will not be expanded
;; twice.  Nodes already expanded are kept in a list of closed nodes.

(defun bfs-graph (start &optional verbose)
  (let ((node-list (list (make-bfs-node :state start :parent nil))))
    (do* ((node (car node-list) (car node-list))
          (closed nil)
          (goal-found (if (goal-state? start) node nil)))
         ((or (null node-list) goal-found)
          (if (null node-list)
              "No solution."
              (extract-bfs-path goal-found)))
      (when verbose (format t "Expanding node ~d~%" (bfs-node-state node)))
      (let* ((all-succs (mapcar #'(lambda (e)
                                    (make-bfs-node :state e
                                                   :parent node))
                                (expand (bfs-node-state node))))
             (succs (nodes-not-visited all-succs node-list closed)))
        (setq goal-found (find-if #'(lambda (s)
                                      (goal-state? (bfs-node-state s)))
                                  succs))
        (setq node-list (append (cdr node-list) succs))
        (setq closed (cons node closed))))))

;; Function NODES-NOT-VISITED returns all nodes on ALL-SUCCS that are not
;; present on NODE-LIST or CLOSED.  That is, all nodes that have not been
;; previously visited.

(defun nodes-not-visited (all-succs node-list closed)
  (set-difference
    (set-difference all-succs
                    node-list
                    :test #'(lambda (n1 n2)
                              (eq-states (bfs-node-state n1)
                                         (bfs-node-state n2))))
    closed
    :test #'(lambda (n1 n2)
              (eq-states (bfs-node-state n1)
                         (bfs-node-state n2)))))
```

```
      (format ifile "  (target =")
      (do ((j 1 (1+ j))) ((> j TOTAL-OUTPUTS) nil)
        (format ifile "~4,1f" (aref TESTING-OUTPUTS i j)))
      (format ifile ")~%"))))

;;;
;;; Print Weights.
;;;

(defun print-weights (infile)
  (with-open-file (ifile infile :direction :output :if-exists :append)
    (format ifile "~%~%Weights:~%~%")
    (do ((layer NETWORK (net-next-layer layer)) (k 1 (1+ k)))
        ((output-layer? layer) nil)
      (do ((u 0 (1+ u))) ((> u (net-size layer)) nil)
          (do ((u2 1 (1+ u2))) ((> u2 (net-size (net-next-layer layer))) nil)
            (let ((the-connection (aref (net-connections layer) u u2)))
              (format ifile "~d~%"
                      (list k u u2 (connection-weight the-connection))))))))

;;;
;;; Backprop.
;;;

(defun backprop (infile)
  (read-data infile)
  (init-network)
  (learn infile)
  (print-weights infile))
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                        DEPTH-FIRST SEARCH
                            "dfs.lisp"
-----------------------------------------------------------------------|#

;; ---------------------------------------------------------------------
;; Function DFS does a depth-first search from the start state. If a goal
;; state is found, it returns a solution path.  DFS can easily get stuck
;; in an infinite loop, since it may enter the same state twice along the
;; same search path.

(defun dfs (start &optional verbose)
  (let ((result (dfs-1 start verbose)))
    (if (null result) "No solution." result)))

(defun dfs-1 (start verbose)
  (when verbose (format t "Expanding node ~d~%" start))
  (cond ((goal-state? start) (list start))
        (t
         (do ((succs (expand start) (cdr succs))
              (solution-found nil))
             ((or solution-found (null succs))
              (if solution-found (cons start solution-found) nil))
           (setq solution-found (dfs-1 (car succs) verbose))))))


;; ---------------------------------------------------------------------
;; Function DFS-AVOID-LOOPS does a depth-first search from the start state,
;; but never enters a state which it has already entered along the current
;; search path. It avoids loops by remembering all of the ancestors of a
;; given state.
;;
;; Function DFS-WITH-CUTOFF takes an additional parameter, DEPTH-CUTOFF,
;; and performs the same search as DFS-AVOID-LOOPS, but never searches
;; to a depth beyond the cutoff.

(defun dfs-avoid-loops (start &optional verbose)
  (let* ((parents nil)
         (depth-cutoff *infinity*)
         (result (dfs-avoid-loops-1 start parents depth-cutoff verbose)))
    (if (null result) "No solution." result)))

(defun dfs-with-cutoff (start depth-cutoff &optional verbose)
  (let* ((parents nil)
         (result (dfs-avoid-loops-1 start parents depth-cutoff verbose)))
    (if (null result) "No solution." result)))


(defun dfs-avoid-loops-1 (start parents depth-cutoff verbose)
  (when verbose (format t "Expanding node ~d~%" start))
  (cond ((goal-state? start) (list start))
        ((= depth-cutoff 0) nil)        ; decrease depth-cutoff until it is 0
        (t  (let ((all-succs (expand start)))
              (do ((succs (remove-dfs-parents all-succs parents)
                          (cdr succs))
                   (solution-found nil))
                  ((or solution-found (null succs))
                   (if solution-found (cons start solution-found) nil))
                (setq solution-found (dfs-avoid-loops-1 (car succs)
                                                        (cons start parents)
                                                        (1- depth-cutoff)
                                                        verbose)))))))


(defun remove-dfs-parents (all-succs parents)
  (mapcan #'(lambda (succ)
              (if (member succ parents :test #'eq-states)
                  nil
                  (list succ)))
          all-succs))
```

```
#|------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------
                        ITERATIVE-DEEPENING SEARCH
                              "dfid.lisp"
------------------------------------------------------------------------|#

;; ---------------------------------------------------------------------
;; Function DFID does a depth-first iterative deepening search.  When it
;; reaches the goal state, it returns a solution path.  DFID calls
;; DFS-WITH-CUTOFF each time with greater depth, which performs a depth
;; limited, depth-first search avoiding loops along a single path.

(defun dfid (start &optional verbose)
   (do ((success nil)
        (depth 1 (1+ depth)))
       (success success)
     (when verbose (format t "Beginning iteration number ~d~%" depth))
     (let ((result (dfs-with-cutoff start depth verbose)))
        (when (not (and (stringp result)
                        (string-equal result "No solution.")))
           (setq success result)))))


(defun dfs-with-cutoff (start depth-cutoff &optional verbose)
  (let* ((parents nil)
         (result (dfs-avoid-loops-1 start parents depth-cutoff verbose)))
    (if (null result) "No solution." result)))


(defun dfs-avoid-loops-1 (start parents depth-cutoff verbose)
  (when verbose (format t "Expanding node ~d~%" start))
  (cond ((goal-state? start) (list start))
        ((= depth-cutoff 0) nil)        ; decrease depth-cutoff until it is 0
        (t  (let ((all-succs (expand start)))
              (do ((succs (remove-ida-parents all-succs parents)
                          (cdr succs))
                   (solution-found nil))
                  ((or solution-found (null succs))
                   (if solution-found (cons start solution-found) nil))
                (setq solution-found (dfs-avoid-loops-1 (car succs)
                                                        (cons start parents)
                                                        (1- depth-cutoff)
                                                        verbose)))))))


;; ---------------------------------------------------------------------
;; Function IDA-STAR performs a heuristic depth-first iterative deepening
;; search.  It explores the search space deeper and deeper on each iteration;
;; during each iteration, it expands all nodes whose g+h values are less
;; than some threshold.
;;
;; The threshold is initialized to *infinity* before we call IDA-STAR-DFS
;; during an iteration, and is augmented by the minumum amount it was exceeded
;; during that iteration.

(defvar *amount-exceeded* 0)

(defun ida-star (start &optional verbose)
   (do ((solution-found nil)
        (iteration 1 (1+ iteration))
        (threshold (heuristic start)))
       (solution-found solution-found)
     (when verbose (format t "Beginning iteration number ~d, threshold = ~d~%"
                           iteration threshold))
     (setq *amount-exceeded* *infinity*)
     (let ((parents nil)
           (depth 0))
       (setq solution-found
             (ida-star-dfs start parents threshold depth verbose)))
     (setq threshold (+ threshold *amount-exceeded*))))


;; Function IDA-STAR-DFS performs heuristic threshold-limited depth-first
;; search avoiding loops along a single path.

(defun ida-star-dfs (start parents threshold cost-so-far verbose)
  (cond ((goal-state? start) (list start))
        ((> (+ cost-so-far (heuristic start)) threshold)
         (setq *amount-exceeded* (min *amount-exceeded*
                                      (- (+ cost-so-far (heuristic start))
                                         threshold)))
         nil)
        (t
         (when verbose (format t "Expanding node ~d~%" start))
         (let ((all-succs (expand start)))
           (do ((succs (remove-ida-parents all-succs parents)
                       (cdr succs))
                (solution-found nil))
               ((or solution-found (null succs))
                (if solution-found (cons start solution-found) nil))
             (setq solution-found (ida-star-dfs (car succs)
                                                (cons start parents)
                                                threshold
                                                (+ cost-so-far
                                                   (cost-of-move start
                                                                 (car succs)))
                                                verbose)))))))

(defun remove-ida-parents (all-succs parents)
  (mapcan #'(lambda (succ)
              (if (member succ parents :test #'eq-states)
                  nil
                  (list succ)))
          all-succs))
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                            GENERIC GAME
                         "generic-game.lisp"
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------

This file can be used as a template to create new games.  To create your own
game, fill in the functions below, adding whatever auxiliary functions are
necessary.  The functions are:

 (deep-enough pos depth)      t if the search has proceeded deep enough.
 (static pos player)          evaluation of position pos from player's
                               point of view.
 (movegen pos player)         generate all successor positions to pos.
 (opposite player)            return the opposite player.
 (print-board pos)            print board position pos.
 (make-move pos player move)  return new position based on old position and
                               player's move.
 (won? pos player)            t if player has won.
 (drawn? pos)                 t if pos is a drawn position.

The important variables are:

 *start*                      the initial board configuration.

These functions and variables are all called from minimax.lisp.

For an example game, see "tictactoe.lisp".
-----------------------------------------------------------------------|#

;; Variable *START* holds the starting position for the game.

(defvar *start*)

(setq *start* nil)

;; Function DEEP-ENOUGH returns t if the search has proceeded deep enough.
;; This can be a function of (either or both) the position and the depth of
;; the search.

(defun deep-enough (pos depth)
  nil)


;; Function STATIC returns a number which is the evaluation of position pos
;; from the viewpoint of the player.  A positive number is favorable for the
;; player; a negative one is unfavorable.

(defun static (pos player)
  nil)


;; Function MOVEGEN takes a position and a player and generates all legal
;; successor positions, i.e., all possible moves a player could make.

(defun movegen (pos player)
  nil)


;; Function OPPOSITE returns player one when given player two, and
;; vice-versa.

(defun opposite (player)
  nil)


;; Function PRINT-BOARD prints the game board.

(defun print-board (pos)
  nil)


;; Function MAKE-MOVE returns a new position based on the old position and
;; a player's move.

(defun make-move (pos player move)
  nil)


;; Function WON? returns t if player has won in position pos.

(defun won? (pos player)
  nil)

;; Function DRAWN? returns t if pos is a drawn (tied) position.

(defun drawn? (pos)
  nil)
```

```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                            GENERIC DOMAIN
                            "generic.lisp"
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------

This file can be used as a template to create new domains.  To create your own
domain, fill in the functions below, adding whatever auxiliary functions are
necessary.  The functions are:

 (goal-state? s)              t if s is a goal state
 (eq-states s1 s2)            are s1 and s2 equal?
 (expand s)                   successor states of s
 (hash-state s)               some has value for s
 (print-state s)              print function
 (destroy-state s)            dispose of state structure
 (heuristic s)                heuristic distance between s and the goal
 (generate-problem)           randomly generated start-state
 (cost-of-move s1 s2)         cost of moving from s1 to s2

The important variables are:

 *sample-initial-state*
 *goal-state*

These functions and variables can all be called from an outside search program.
In fact, these are the functions called by our implementations of depth-first,
breadth-first, hill-climbing, A*, DFID, IDA*, and RTA* search.

For example domains, see "n-puzzle.lisp" and "travel.lisp".

-----------------------------------------------------------------------------|#

;; Variable *INFINITY* will be used as the largest possible number.
;; MOST-POSITIVE-FIXNUM is a Lisp symbol that provides it.

(defvar *infinity* most-positive-fixnum)


;; Variable *SAMPLE-INITIAL-STATE* holds a sample initial state.

(defvar *sample-initial-state*)


;; Variable *GOAL-STATE* holds a goal state.  This can be nil if there is no
;; unique goal state.

(defvar *goal-state*)


(setq *sample-initial-state* nil)

(setq *goal-state* nil)

;; ---------------------------------------------------------------------------
;; Functions required by search programs in any domain.


;; Function GOAL-STATE? returns t iff s is a goal state.

(defun goal-state? (s)
  nil)


;; Function EQ-STATES returns t iff s1 and s2 are the same state.

(defun eq-states (s1 s2)
  nil)


;; Function EXPAND takes a state and returns all possible successor states.

(defun expand (s)
  nil)


;; Function PRINT-STATE prints a state.

(defun print-state (s &rest ignore)
  (declare (ignore ignore))
  nil)


;; Function HEURISTIC provides a numerical estimate of the difficulty of
;; reaching a goal state from s.  The higher the estimate, the more difficulty.

(defun heuristic (s)
  nil)


;; Function GENERATE-PROBLEM returns a randomly generated start state.

(defun generate-problem ()
  nil)
```

```
        ((and (neg-class c1) (mult-class c2))
         *fail*)
        ((and (mult-class c1) (atomic-class c2))
         (cons (car c1) (union (list c2) (cdr c1))))
        ((and (mult-class c1) (disj-class c2))
         (cons (car c1) (cons c2 (cdr c1))))
        ((and (mult-class c1) (neg-class c2))
         *fail*)
        ((and (mult-class c1) (mult-class c2))
         (cons (car c1) (union (cdr c1) (cdr c2))))))

;; Function GRAPH-UNIFY
;;
;; Unifies two graphs.  Congruence closure algorithm, runs in
;; O(n log n) time, where n is the number of nodes in the input graphs.

(defun graph-unify (d1 d2)
 (mf-init d1)
 (mf-init d2)
 (let ((e1 (copy-graph d1)) (e2 (copy-graph d2)))
  (do ((pairs (list (cons e1 e2)))
       (current) (u) (v) (newclass) (w))
      ((null pairs) (create-result-graph-2 e1 e2))
    (setq current (pop pairs))
    (setq u (mf-find (car current)))
    (setq v (mf-find (cdr current)))
    (setq newclass (unify-classes (graph-node-class u) (graph-node-class v)))
    (when (eq newclass *fail*) (return *fail*))
    (when (or (and
                (not (equal (graph-node-class u) *variable*))
                (not (null (graph-node-arcs v))))
              (and
                (not (equal (graph-node-class v) *variable*))
                (not (null (graph-node-arcs u)))))
          (return *fail*))
    (setq w (mf-union u v))
    (setf (graph-node-class w) newclass)
    (if (eq w v)
        (carry-labels u v)
        (carry-labels v u))
    (mapc #'(lambda (l)
              (push (cons (graph-node-subnode u l)
                          (graph-node-subnode v l)) pairs))
          (intersection (graph-node-arc-labels u)
                        (graph-node-arc-labels v))))))


;;-----------------------------------------------------------------------
;;
;; Examples
;;
;;-----------------------------------------------------------------------

(defvar g1)
(defvar g2)
(defvar g3)
(defvar g4)
(defvar g5)
(defvar g6)
(defvar g7)
(defvar g8)
(defvar g9)
(defvar g10)
(defvar g11)
(defvar g12)
(defvar g13)
(defvar g14)

(setq g1 (tree->graph '((a 1) (b 2))))
(setq g2 (tree->graph '((b 2) (c 3))))
(setq g3 (tree->graph '((b 3) (d 4))))

;; (graph-unify g1 g2) --> ((a 1) (b 2) (c 3))
;; (graph-unify g1 g3) --> FAIL

(setq g4 (tree->graph '((a (*OR* 1 2 3)) (b 5))))
(setq g5 (tree->graph '((a (*OR* 2 3 4)) (b 5))))

;; (graph-unify g4 g5) --> ((a (*OR* 2 3)) (b 5))

(setq g6 (tree->graph '((a (*NOT* 1 3)) (b 5))))
(setq g7 (tree->graph '((a (*OR* 1 2 3)) (b 5))))


(setq g8 (tree->graph '((a (*MULT* 1 2 3)) (b 5))))
(setq g9 (tree->graph '((a (*MULT* 2 3 4)) (b 5))))

;; (graph-unify g8 g9) --> ((a (*MULT* 4 1 2 3)) (b 5))

(setq g10 (tree->graph '((a ((b ((c 4) (d 6)))) (e 7)))))
(setq g11 (tree->graph '((a ((b ((d 6)))) (e 7) (f 9)))))

;; (graph-unify g10 g11) --> ((a ((b ((c 4) (d c)))) (e 7) (f 9))

(setq g12 (print->graph '(($0000 VAR ((a $0001) (b $0001)))
                          ($0001 VAR nil))))
(setq g13 (print->graph '(($0000 VAR ((a $0001) (b $0002)))
                          ($0001 4 nil)
                          ($0002 4 nil))))
(setq g14 (print->graph '(($0000 VAR ((a $0001) (b $0002)))
                          ($0001 4 nil)
                          ($0002 5 nil))))

;; (graph-unify g12 g13) --> ((a 4) (b 4))

;; (graph-unify g12 g14) --> FAIL

;; (graph->print (graph-unify g12 g13)) -->
;;                       (($0000 VAR ((a $0001) (b $0001) ($0001 4 nil))

;;-----------------------------------------------------------------------
;;
;; NATURAL LANGUAGE EXAMPLE
;;
;;-----------------------------------------------------------------------

(defvar np-graph)
(defvar vp-graph)
(defvar constituents)
(defvar s-to-np-vp-rule)

;; Graph representing the noun phrase "the man".  The category arc has
;; the value NP, and the head arc contains information about determiner,
;; root, and agreement.

(setq np-graph (print->graph
  '(($0001 VAR ((category $0002) (head $0003)))
    ($0002 NP nil)
```

```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                            GENERIC DOMAIN
                            "generic.lisp"
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------

This file can be used as a template to create new domains.  To create your own
domain, fill in the functions below, adding whatever auxiliary functions are
necessary.  The functions are:

 (goal-state? s)              t if s is a goal state
 (eq-states s1 s2)            are s1 and s2 equal?
 (expand s)                   successor states of s
 (hash-state s)               some has value for s
 (print-state s)              print function
 (destroy-state s)            dispose of state structure
 (heuristic s)                heuristic distance between s and the goal
 (generate-problem)           randomly generated start-state
 (cost-of-move s1 s2)         cost of moving from s1 to s2

The important variables are:

 *sample-initial-state*
 *goal-state*

These functions and variables can all be called from an outside search program.
In fact, these are the functions called by our implementations of depth-first,
breadth-first, hill-climbing, A*, DFID, IDA*, and RTA* search.

For example domains, see "n-puzzle.lisp" and "travel.lisp".

-----------------------------------------------------------------------|#

;; Variable *INFINITY* will be used as the largest possible number.
;; MOST-POSITIVE-FIXNUM is a Lisp symbol that provides it.

(defvar *infinity* most-positive-fixnum)


;; Variable *SAMPLE-INITIAL-STATE* holds a sample initial state.

(defvar *sample-initial-state*)


;; Variable *GOAL-STATE* holds a goal state.  This can be nil if there is no
;; unique goal state.

(defvar *goal-state*)


(setq *sample-initial-state* nil)

(setq *goal-state* nil)

;; -----------------------------------------------------------------------
;; Functions required by search programs in any domain.


;; Function GOAL-STATE? returns t iff s is a goal state.

(defun goal-state? (s)
  nil)


;; Function EQ-STATES returns t iff s1 and s2 are the same state.

(defun eq-states (s1 s2)
  nil)


;; Function EXPAND takes a state and returns all possible successor states.

(defun expand (s)
  nil)


;; Function PRINT-STATE prints a state.

(defun print-state (s &rest ignore)
  (declare (ignore ignore))
  nil)


;; Function HEURISTIC provides a numerical estimate of the difficulty of
;; reaching a goal state from s.  The higher the estimate, the more difficulty.

(defun heuristic (s)
  nil)


;; Function GENERATE-PROBLEM returns a randomly generated start state.

(defun generate-problem ()
  nil)
```

```
                    :atomic-negation :complex-negation))
            (*MULT* (if (atom (cadr t1))
                        :atomic-multiple-value :complex-multiple-value))
            (otherwise :complex)))))

;; Function TREE->GRAPH
;;
;; Takes a tree in list format and returns
;; a dag-structure.  The structure will of course have no reentrancy.

(defun tree->graph (t1)
 (let ((k (graph-node-type t1)))
  (cond ((member k (list :atomic :atomic-disjunction
                         :atomic-negation :atomic-multiple-value))
         (create-graph-node :class t1 :arcs nil))
        ((eq k :complex)
         (let ((n (create-graph-node :class *variable* :arcs nil)))
              (mapc #'(lambda (a)
                        (add-arc-in-order n
                           (create-arc
                             :label (car a)
                             :destination (tree->graph (cadr a)))))
                    t1)
              n))
        ((eq k :complex-disjunction)
         (create-graph-node :class *variable*
                    :arcs (cons '*OR*
                                (mapcar #'(lambda (n) (tree->graph n))
                                        (cdr t1)))))
        ((eq k :complex-negation)
         (create-graph-node :class *variable*
                    :arcs (cons '*NOT*
                                (mapcar #'(lambda (n) (tree->graph n))
                                        (cdr t1)))))
        ((eq k :complex-multiple-value)
         (create-graph-node :class *variable*
                    :arcs (cons '*MULT*
                                (mapcar #'(lambda (n) (tree->graph n))
                                        (cdr t1)))))))))

;; Function GRAPH->PRINT
;;
;; Takes a dag-structure and returns a dag coded in list format.
;; For each node in the dag-structure, there is a corresponding element
;; in the list.  Each element has the form:
;;
;;      (<node-variable> <node-class> <node-subnodes>)
;;
;; The order of elements corresponds to the order of a depth-first
;; traversal of the dag-structure.  If the arcs of the nodes are
;; ordered lexicographically, each dag-structure will have a well-defined
;; canonical list-format code.

(defun graph->print (d)
  (let ((n (nodes-in-graph d)))
    (mapc #'(lambda (n1 dv) (setf (graph-node-mark n1) (list dv)))
          n *dag-variables*)
    (mapc #'(lambda (p)
              (setf (graph-node-mark p)
                    (append
                      (graph-node-mark p)
                      (list (graph-node-class p))
                      (list
                       (mapcar
                         #'(lambda (a)
                             (list (arc-label a)
                                   (car (graph-node-mark (arc-destination a)))))
                         (graph-node-arcs p))))))
          n)
    (mapcar #'graph-node-mark n)))

;; Function PRINT->GRAPH
;;
;; Takes a dag coded in list format and returns a dag-structure.
;; (The list format code described above is decoded back into a
;; structure).  This function orders the arcs leaving a node
;; lexicographically, so that exactly the same structure will appear
;; no matter how many times it is coded and decoded.

(defun print->graph (dp)
  (let ((n (mapcar #'(lambda (n1) (create-graph-node :mark n1)) dp)))
    (mapc #'(lambda (p)
              (setf (graph-node-mfset p) nil)
              (setf (graph-node-class p) (second (graph-node-mark p)))
              (mapc
                #'(lambda (a)
                    (add-arc-in-order p
                      (create-arc
                        :label (first a)
                        :destination
                          (find (second a) n
                                :key
                                #'(lambda (x) (car (graph-node-mark x)))))))
                (third (graph-node-mark p))))
          n)
    (car n)))

;;;-------------------------------------------------------------------
;;; DAG-MFSET
;;;-------------------------------------------------------------------

;; This section contains functions for performing disjoint set operations
;; on dag nodes.

;; Function MF-ROOT-CLASS?
;;
;; Takes a node and returns T if the node is the root of its equivalence
;; class tree.

(defun mf-root-class? (n) (listp (graph-node-mfset n)))

;; Function MF-FIND
;;
;; Performs the FIND operation for UNION-FIND disjoint sets.  Given
;; a node in a dag-structure, it returns another node, namely the root
;; of the equivalence class tree for the input node.  After the FIND,
;; the tree is made more shallow by adjustment of pointers to the root.

(defun mf-find (x)
  (do ((q1 nil) (t1 x))
      ((mf-root-class? t1)                    ; do path compression
       (progn ()
              (mapc #'(lambda (n) (setf (graph-node-mfset n) t1)) q1)
              t1))
      (push t1 q1)
      (setq t1 (graph-node-mfset t1))))

;; Function MF-UNION
```

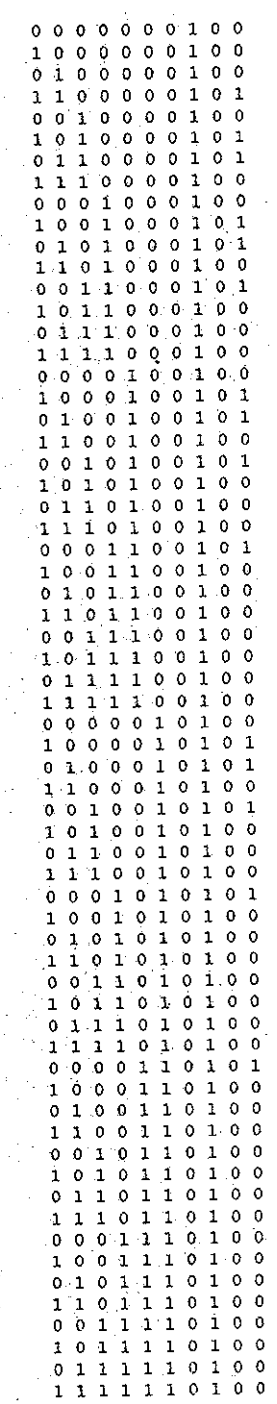
```
                (if (mf-root-class? n)
                    (mapcar #'graph-node-mark (graph-node-mfset n))
                    (graph-node-mark (graph-node-mfset n))))
            (setf (graph-node-arcs (graph-node-mark n))
                  (mapcar #'(lambda (a)
                              (create-arc
                                :label (arc-label a)
                                :destination
                                  (graph-node-mark (arc-destination a))))
                          (graph-node-arcs n))))
          n1)
    (graph-node-mark node)))

;; ACCESSORS

(defun graph-node-arc (node label)
  (find label (graph-node-arcs node)
        :key #'(lambda (a) (arc-label a))))

(defun graph-node-subnode (node label)
  (arc-destination (graph-node-arc node label)))

(defun graph-node-arc-labels (node)
  (mapcar #'arc-label (graph-node-arcs node)))

; MODIFIERS

(defun add-arc (node arc)
  (unless (graph-node-arc node (arc-label arc))
    (push arc (graph-node-arcs node))))

(defun add-arc-in-order (node arc)
  (unless (graph-node-arc node (arc-label arc))
    (setf (graph-node-arcs node)
          (merge 'list (list arc) (graph-node-arcs node)
            #'(lambda (x y) (string< (string (arc-label x))
                                     (string (arc-label y))))))))

;;-----------------------------------------------------------------------
;; DAG-FNS
;;-----------------------------------------------------------------------

; This section contains various functions over dag structures.

;
; Function MARK-GRAPH
;
; Takes a root node of a graph and a marker.  Sets the mark field of
; every node in the graph equal to the marker.  Uses a gensym'd
; temporary marker name.

(defun mark-graph (node mark)
  (let ((marker (gensym "MARKER-")))
    (mark-graph-1 node marker)
    (mark-graph-1 node mark)))

(defun mark-graph-1 (node sym)
  (when (not (eq (graph-node-mark node) sym))
        (setf (graph-node-mark node) sym)
        (mapc #'(lambda (a) (mark-graph-1 (arc-destination a) sym))
              (graph-node-arcs node))))

;
; Function DEPTH-FIRST-TRAVERSAL
;
; Takes a root node of a graph and returns a list of nodes in the
; graph.  Assumes that all nodes begin with MARK = NIL.

(defun depth-first-traversal (node)
  (setf (graph-node-mark node) t)
  (cond ((null (graph-node-arcs node)) (list node))
        (t (cons node
                 (mapcan #'(lambda (n)
                             (if (null (graph-node-mark n))
                                 (depth-first-traversal n)
                                 nil))
                         (mapcar #'arc-destination
                                 (graph-node-arcs node)))))))

;
; Function NODES-IN-GRAPH
;
; Takes a root node of a graph and returns a list of all nodes in
; the graph.  Uses a standard marking procedure to avoid traversing the
; same portion of the graph more than once.

(defun nodes-in-graph (node)
  (mark-graph node nil)
  (depth-first-traversal node))

;;;---------------------------------------------------------------------
;;; DAG-PRINT
;;;---------------------------------------------------------------------

;; This section contains functions to read and write arbitrary graphs.
;; Graphs are coded as lists with variables to mark reentrancy.  Thus,
;; graph structures can effectively (1) be printed on the the screen,
;; and (2) be written to files and read back in.

(defvar *dag-variables*
  '($0000 $0001 $0002 $0003 $0004 $0005 $0006 $0007 $0008 $0009
    $0010 $0011 $0012 $0013 $0014 $0015 $0016 $0017 $0018 $0019
    $0020 $0021 $0022 $0023 $0024 $0025 $0026 $0027 $0028 $0029
    $0030 $0031 $0032 $0033 $0034 $0035 $0036 $0037 $0038 $0039
    $0040 $0041 $0042 $0043 $0044 $0045 $0046 $0047 $0048 $0049
    $0050 $0051 $0052 $0053 $0054 $0055 $0056 $0057 $0058 $0059))

;; Function GRAPH->TREE
;;
;; Takes a dag-structure and returns a tree in list format.  Loses
;; reentrancy of dag (copies are made).

(defun graph->tree (d)
  (cond ((null (graph-node-arcs d)) (graph-node-class d))
        (t (mapcar #'(lambda (a) (list (arc-label a)
                                       (graph->tree (arc-destination a))))
                   (graph-node-arcs d)))))

;; Function GRAPH-NODE-TYPE
;;
;; Return the type of an graph node stored in tree (s-expression) format.

(defun graph-node-type (t1)
  (cond ((atom t1) :atomic)
        (t (case (car t1)
             (*OR* (if (atom (cadr t1))
                       :atomic-disjunction :complex-disjunction))
             (*NOT* (if (atom (cadr t1))
```

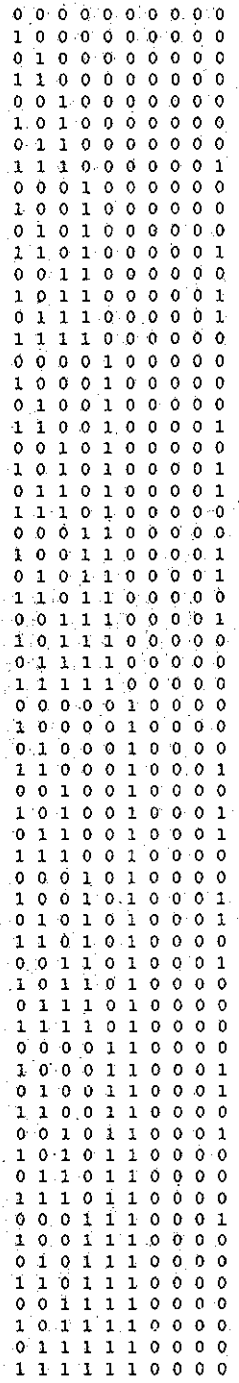
```
#|------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------
                        HILL-CLIMBING SEARCH
                           "hill.lisp"
------------------------------------------------------------------------|#

;; Function HILL-CLIMBING performs steepest-ascent hill climbing search.
;; It either returns a solution path from start to goal, or it gets stuck
;; in a local minimum.  This version of hill-climbing does not do any
;; backtracking to extricate itself from a local minimum -- it simply halts.

(defun hill-climbing (start &optional verbose)
  (do ((current-state start)
       (solution-path (list start))
       (minimum-reached nil))
      (minimum-reached
         (if (goal-state? current-state)
             (nreverse solution-path)
             "Stuck in local-minimum."))
    (when verbose (format t "Expanding state ~d~%" current-state))
    (let* ((succs (expand current-state))
           (best-succ (choose-best succs)))
       (cond ((< (heuristic best-succ)
                 (heuristic current-state))
              (setq current-state best-succ)
              (setq solution-path (cons current-state solution-path)))
             (t
              (setq minimum-reached t))))))


;; Function CHOOSE-BEST returns the member of SUCCESSORS with the lowest
;; heuristic score, i.e., the state deemed closest to goal.

(defun choose-best (successors)
  (do ((s successors (cdr s))
       (best-succ nil)
       (best-score *infinity*))
      ((null s) best-succ)
    (let ((succ (car s)))
       (when (< (heuristic succ)
                best-score)
          (setq best-succ succ)
          (setq best-score (heuristic succ))))))
```

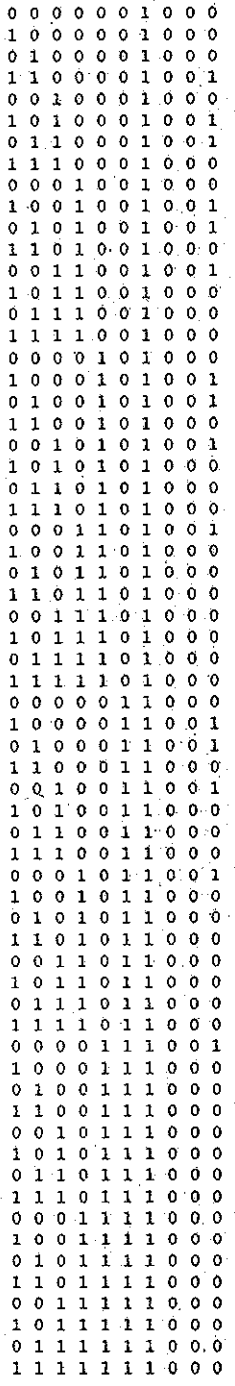
```
    ($0003 VAR ((det $0004) (root $0005) (agreement $0006)))
    ($0004 the nil)
    ($0005 man nil)
    ($0006 singular nil))))

;; Graph representing the verb phrase "kills bugs".

(setq vp-graph (print->graph
  '(($0007 VAR ((category $0008) (head $0009)))
    ($0008 VP nil)
    ($0009 VAR ((root $0010) (tense $0011)
                (agreement $0012) (object $0013)))
    ($0010 kill nil)
    ($0011 present nil)
    ($0012 singular nil)
    ($0013 VAR ((category $0014) (head $0015)))
    ($0014 NP nil)
    ($0015 VAR ((root $0016) (agreement $0017)))
    ($0016 bug nil)
    ($0017 plural nil))))

;; Graph tying NP-GRAPH and VP-GRAPH into a single constituent graph with
;; arcs labeled CONSTIT1 and CONSTIT2.

(setq constituents
  (create-graph-node :arcs (list (create-arc :label 'constit1
                                             :destination np-graph)
                                 (create-arc :label 'constit2
                                             :destination vp-graph))))

;; Graph representing the augmented context-free grammar rule S -> NP VP,
;; enforcing subject-object number agreement, and building the resulting
;; structure for a sentence.

(setq s-to-np-vp-rule (print->graph
  '(($0000 VAR ((constit1 $0001) (constit2 $0002) (build $0003)))
    ($0001 VAR ((category $0004) (head $0007)))
    ($0002 VAR ((category $0005) (head $0008)))
    ($0003 VAR ((category $0006) (head $0008)))
    ($0004 NP nil)
    ($0005 VP nil)
    ($0006 S nil)
    ($0007 VAR ((agreement $0009)))
    ($0008 VAR ((subject $0007) (agreement $0009) (mood $0010)))
    ($0009 VAR nil)
    ($0010 declarative nil))))

;; Unify rule with constituents...
;;
;; (graph-unify s-to-np-vp-rule constituents)

;; Unify rule with constituents, and retrieve result...
;;
;; (graph-node-subnode (graph-unify s-to-np-vp-rule constituents) 'build)


;; (graph-unify s-to-np-vp-rule constituents) ->
;;
;; ((BUILD
;;   ((CATEGORY S)
;;    (HEAD
;;     ((TENSE PRESENT) (ROOT KILL)
;;      (OBJECT
;;       ((CATEGORY NP) (HEAD ((AGREEMENT PLURAL) (ROOT BUG)))))
;;      (AGREEMENT SINGULAR) (MOOD DECLARATIVE)
;;      (SUBJECT ((ROOT MAN) (DET THE) (AGREEMENT SINGULAR)))))))
;;  (CONSTIT1
;;   ((CATEGORY NP) (HEAD ((ROOT MAN) (DET THE) (AGREEMENT SINGULAR)))))
;;  (CONSTIT2
;;   ((CATEGORY VP)
;;    (HEAD
;;     ((TENSE PRESENT) (ROOT KILL)
;;      (OBJECT
;;       ((CATEGORY NP) (HEAD ((AGREEMENT PLURAL) (ROOT BUG)))))
;;      (AGREEMENT SINGULAR) (MOOD DECLARATIVE)
;;      (SUBJECT ((ROOT MAN) (DET THE) (AGREEMENT SINGULAR)))))))))


;; (graph-node-subnode (graph-unify s-to-np-vp-rule constituents) 'build) ->
;;
;; ((CATEGORY S)
;;  (HEAD
;;   ((TENSE PRESENT) (ROOT KILL)
;;    (OBJECT ((CATEGORY NP) (HEAD ((AGREEMENT PLURAL) (ROOT BUG)))))
;;    (AGREEMENT SINGULAR) (MOOD DECLARATIVE)
;;    (SUBJECT ((ROOT MAN) (DET THE) (AGREEMENT SINGULAR))))))
```

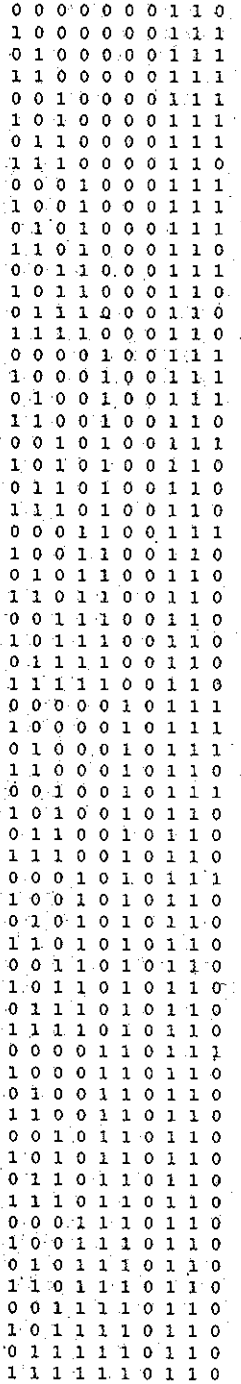
```
1 1 1 0 1 1 0 0 0 0
0 1 0 0 0 1 1 0 0 1
1 0 1 0 1 1 1 1 0 0
0 1 0 1 0 0 0 0 1 1
0 1 1 1 0 0 0 0 1 1
```

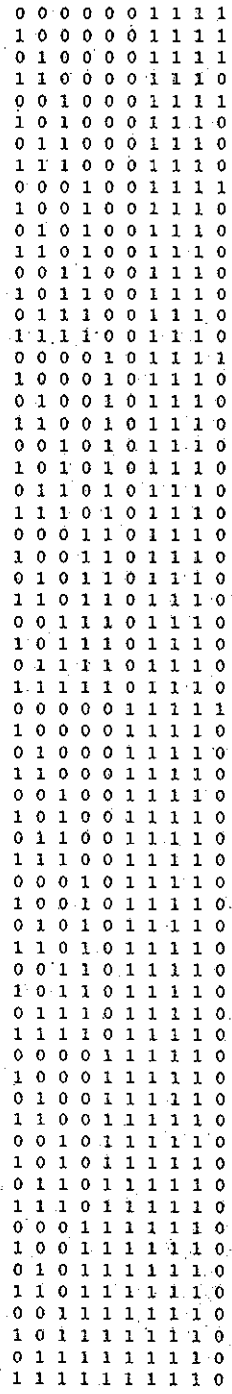
```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                            HOPFIELD NETWORKS
                             "hopfield.lisp"
-----------------------------------------------------------------------------|#

;; ---------------------------------------------------------------------------

;; Structure HOPNET defines a Hopfield network.  It consists of a number of
;; units, a set of activations (a 1-d matrix), and a set of connections
;; (a 2-d matrix).

(defstruct hopnet
  size
  activations
  connections)


;; Variable *SAMPLE-NET* represents the network shown on page 490 of
;; Artificial Intelligence.

(defvar *SAMPLE-NET* nil)

(setq *SAMPLE-NET*
  (make-hopnet
        :size 7
        :activations (make-array 7
                                 :initial-contents '(0 1 0 0 1 0 0))
        :connections (make-array '(7 7)
                                 :initial-contents
                                      '((0 -1 1 -1 0 0 0)
                                        (-1 0 0 3 0 0 0)
                                        (1 0 -1 2 1 0 0)
                                        (-1 3 -1 0 0 -2 3)
                                        (0 0 2 0 0 1 0)
                                        (0 0 1 -2 1 0 -1)
                                        (0 0 0 3 0 -1 0)))))

;; Function PUT-NET-INTO-RANDOM-STATE scrambles the activation levels of the
;; units in a network.

(defun put-net-into-random-state (net)
  (let ((size (hopnet-size net))
        (activations (hopnet-activations net)))
    (dotimes (i size)
      (setf (aref activations i) (random 2)))
    (hopnet-activations net)))


;; Function SETTTLE implements Hopfield's algorithm for letting a network
;; settle into a stable state.

(defun settle (net epochs)
  (let ((size (hopnet-size net))
        (activations (hopnet-activations net))
        (connections (hopnet-connections net)))
    (dotimes (e (1+ epochs))
      (format t "Epoch ~4d:  " e)
      (print-activations net)
      (let ((u (random size))
            (weighted-input 0))
        (dotimes (i size)
          (setq weighted-input
                (+ weighted-input (* (aref activations i)
                                     (aref (hopnet-connections net) u i)))))
        (cond ((> weighted-input 0)
               (setf (aref activations u) 1))
              (t
               (setf (aref activations u) 0)))))))


;; Function PRINT-ACTIVATIONS prints the activation levels of the units.

(defun print-activations (net)
  (dotimes (i (hopnet-size net))
    (format t "~2d" (aref (hopnet-activations net) i)))
  (format t "~%"))
```

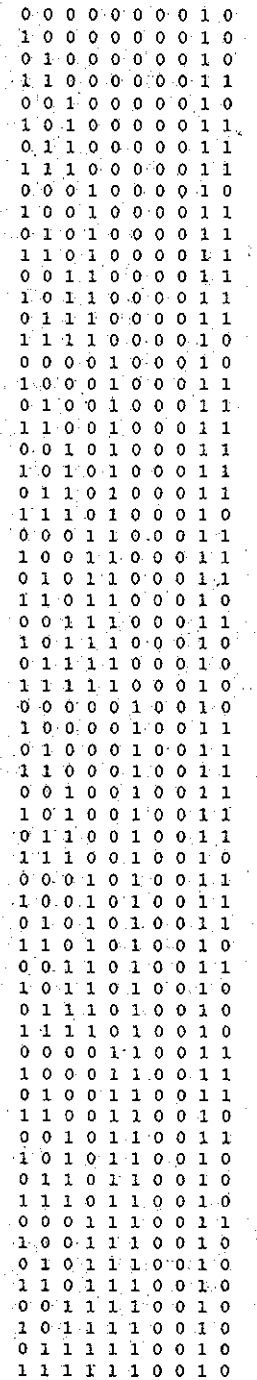


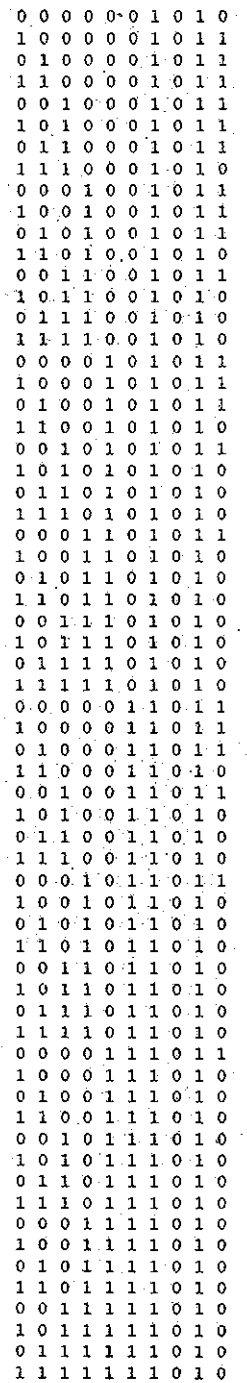












```
#|-----------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------
                              MINIMAX SEARCH
                             "minimax.lisp"
-----------------------------------------------------------------------|#

#|-----------------------------------------------------------------------

 Minimax game playing.

 This file contains functions for doing game-playing search.  This
 program will play any game for which the following functions and
 variables are defined:

        (print-board b)
        (movegen pos player)
        (opposite player)
        (static pos player)
        (won? pos player)
        (drawn? pos)
        (deep-enough pos depth)
        (make-move pos player move)
        *start*

 These functions are implemented for tic-tac-toe in the file
 "tictactoe.lisp".

-----------------------------------------------------------------------|#


;; Function MINIMAX performs minimax search from a given position (pos),
;; to a given search ply (depth), for a given player (player).  It returns
;; a list of board positions representing what it sees as the best moves for
;; both players.  The first element of the list is the value of the board
;; position after the proposed move.

(defun make-str (a b) (list a b))

(defun value (str) (car str))

(defun path (str) (cadr str))

(defun next-move (str) (car (path str)))


(defun minimax (pos depth player)
  (cond ((deep-enough pos depth)
         (make-str (static pos player) nil))
        (t
         (let ((successors (movegen pos player))
               (best-score -99999)
               (best-path nil))
           (cond ((null successors) (make-str (static pos player) nil))
                 (t
                  (do ((s successors (cdr s)))
                      ((null s))
                    (let* ((succ (car s))
                           (result-succ (minimax succ (1+ depth)
                                                 (opposite player)))
                           (new-value (- (value result-succ))))
                      (when (> new-value best-score)
                        (setq best-score new-value)
                        (setq best-path (cons succ (path result-succ))))))
                  (make-str best-score best-path)))))))


;; Function MINIMAX-A-B performs minimax search with alpha-beta pruning.
;; It is far more efficient than MINIMAX.

(defun minimax-a-b (pos depth player)
  (minimax-a-b-1 pos depth player 99999 -99999))

(defun minimax-a-b-1 (pos depth player use-thresh pass-thresh)
  (cond ((deep-enough pos depth)
         (make-str (static pos player) nil))
        (t
         (let ((successors (movegen pos player))
               (best-path nil))
           (cond ((null successors) (make-str (static pos player) nil))
                 (t
                  (do ((s successors (cdr s)) (quit nil))
                      ((or quit (null s)))
                    (let* ((succ (car s))
                           (result-succ (minimax-a-b-1 succ (1+ depth)
                                                       (opposite player)
                                                       (- pass-thresh)
                                                       (- use-thresh)))
                           (new-value (- (value result-succ))))
                      (when (> new-value pass-thresh)
                        (setq pass-thresh new-value)
                        (setq best-path (cons succ (path result-succ))))
                      (when (>= pass-thresh use-thresh) (setq quit t))))
                  (make-str pass-thresh best-path)))))))


;; Function PLAY allows you to play a game against the computer.  Call (play)
;; if you want to move first, or (play t) to let the computer move first.

(defun play (&optional machine-first?)
  (let ((b *start*))
    (when machine-first?
      (let ((m1 (minimax-a-b b 0 'o)))
        (setq b (next-move m1))))
    (do ()
        ((or (won? b 'x) (won? b 'o) (drawn? b))
         (format t "Final position: ~%")
         (print-board b)
         (when (won? b 'o) (format t "I win.~%"))
         (when (won? b 'x) (format t "You win.~%"))
         (when (drawn? b)  (format t "Drawn.~%")))
      (print-board b)
      (format t "Your move: ")
      (let ((m (- (read) 1)))
        (setq b (make-move b 'x m))
        (when (not (drawn? b))
          (print-board b)
          (let ((m1 (minimax-a-b b 0 'o)))
            (setq b (next-move m1))
```

Network structure:	(9 2 1)
Epochs:	500
Test after every N epochs:	10
Learning rate (eta):	0.35
Momentum (alpha):	0.90
Noise?:	0
Training data:	life.train
Testing data:	life.test

```
#|------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------
                          N-PUZZLE DOMAIN
                          "n-puzzle.lisp"
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------

This file contains code for the n-puzzle search problem.
The important functions are (s, s1 and s2 are states):

 (goal-state? s)                t if s is a goal state
 (eq-states s1 s2)              are s1 and s2 equal?
 (expand s)                     successor states of s
 (hash-state s)                 some has value for s
 (print-state s)                print function
 (destroy-state s)              dispose of state structure
 (heuristic s)                  heuristic distance between s and the goal
 (generate-problem)             randomly generated start-state
 (convert-list-to-state s)      function for creating states
 (cost-of-move s1 s2)           always returns 1

The important variables are:

 *sample-initial-state*
 *goal-state*

These functions and variables can all be called from an outside search program.
In fact, these are the functions called by our implementations of depth-first,
breadth-first, hill-climbing, A*, DFID, IDA*, and RTA* search.

------------------------------------------------------------------------|#

;; Variable *INFINITY* will be used as the largest possible number.
;; MOST-POSITIVE-FIXNUM is a Lisp symbol that provides it.

(defvar *infinity* most-positive-fixnum)

;; ------------------------------------------------------------------------
;; Variable *GOAL-STATE* is a standard goal configuration: tiles ordered,
;; blank in the upper-left hand corner.

(defvar *goal-state* nil)

;; Variable *PUZZLE-SIZE* gives the size of the puzzle along one dimension.
;; The 8-puzzle has size 3, the 15-puzzle has size 4, and the 24-puzzle
;; has size 5.

(defvar *puzzle-size* nil)
(setq *puzzle-size* 3)

;; Variable *PUZZLE-TILES* is *PUZZLE-SIZE* squared.

(defvar *puzzle-tiles* nil)
(setq *puzzle-tiles* (* *puzzle-size* *puzzle-size*))

;; Variable *BLANK-TILE* represents the empty position in the puzzle.

(defvar *blank-tile* nil)
(setq *blank-tile* 0)

;; Variable *FREE-STATES* is used for memory management.  It is a list of
;; no longer needed states.

(defvar *free-states* nil)

;; ------------------------------------------------------------------------
;; Structure PUZZLE-STATE stores a particular state of the puzzle.  It
;; stores two distinct but equivalent representations.  The first (board)
;; is a two-dimensional array, each element of which is a tile represented by
;; its number (0 being blank).  The second representation consists of two
;; arrays (xcoord/ycoord).  Each array is indexed by the tile number and
;; gives the coordinate of that tile.  Thus, it is easy to find out what tile
;; is located at position (x,y) and it is also easy to find out the position
;; of a particular tile.  Position (0,0) is the left lower side of the board.

(defstruct (puzzle-state (:print-function print-state))
      board
      xcoords
      ycoords)


;; ------------------------------------------------------------------------
;; Function MAKE-STATE creates an empty state.

(defun make-state ()
  (make-puzzle-state
        :board (make-array (list *puzzle-size* *puzzle-size*)
                           :element-type 'integer)
        :xcoords (make-array *puzzle-tiles*
                             :element-type 'integer)
        :ycoords (make-array *puzzle-tiles*
                             :element-type 'integer)))


;; Function CREATE-STATE returns a new state, either by retrieving one from
;; *FREE-STATES*, or by calling MAKE-STATE.
;;
;; Function DESTROY-STATE adds a state to *FREE-STATES*, so that it is
;; available whenever we need a state structure.

(defun create-state ()
  (cond ((null *free-states*) (make-state))
        (t (let ((s (car *free-states*)))
             (setq *free-states* (cdr *free-states*))
             s))))

(defun destroy-state (s)
   (setq *free-states* (cons s *free-states*)))

;; ------------------------------------------------------------------------
;; Function TILE-NUMBER takes a state and a pair of (x,y) coordinates, and
;; returns the tile number of the tile located there (0 if blank).
;;
;; Functions XCOORD, YCOORD take a state and a tile number, and return
;; coordinate locations of the tile.
;;
;; Functions SET-TILE-NUMBER, SET-XCOORD, and SET-YCOORD modify a state.
```

```
        (if (and (not (drawn? b))
                 (not (won? b 'o)))
            (format t "My move: ~%")))))))
```

```
#|------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------
                        PROPOSITIONAL RESOLUTION
                            "resolve.lisp"
------------------------------------------------------------------------|#

;; ------------------------------------------------------------------------
;; Variable *AXIOMS* holds the list of axioms in clause form.  For example,
;; the axioms below should be read:
;;
;;              p
;;              ~p \/ ~q \/ r
;;              ~s \/ q
;;              ~t \/ q
;;              t
;;
;; as in the example on page 150 of the text.

(defvar *axioms*)

(setq *axioms*
  '((p)
    ((not p) (not q) r)
    ((not s) q)
    ((not t) q)
    (t)))

;; ------------------------------------------------------------------------
;; Function PROVE attempts to prove a statement using propositional resolution.
;; It adds the negation of the statement to the axioms and tries to find a
;; contradiction.  It keeps a list of all axioms and inferred facts.  Every
;; time it finds a new fact, it pairs that fact with all other known facts
;; and adds the pairs to the end of pairs-to-resolve.  If a pair of facts
;; ever resolve to an empty clause (contradiction), then the program halts
;; and returns the list of all known facts.  If pairs-to-resolve is exhausted,
;; then the program returns nil, as a sign that the proof could not be
;; carried out.

(defun prove (statement)
  (let* ((new-axioms (cons (if (consp statement)
                               (cadr statement)
                               (list 'not statement))
                           *axioms*))
         (cf-axioms (mapcar #'convert-to-clause-form new-axioms))
         (pairs-to-resolve
            (do ((c cf-axioms (cdr c))
                 (p nil))
                ((null (cdr c)) p)
              (setq p (append p (mapcar #'(lambda (x)
                                            (list (car c) x))
                                        (cdr c)))))))
    (do ((success nil))
        ((or (null pairs-to-resolve) success)
         (nreverse success))
      (let* ((p (car pairs-to-resolve))
             (resolvents (resolve (car p) (cadr p))))
        (setq pairs-to-resolve (cdr pairs-to-resolve))
        (dolist (r resolvents)
          (setq pairs-to-resolve
                (append pairs-to-resolve
                        (mapcar #'(lambda (x) (list x r))
                                cf-axioms)))
          (setq cf-axioms (cons r cf-axioms)))
        (when (member nil cf-axioms)
          (setq success cf-axioms))))))

;; Function RESOLVE returns all ways of resolving two clauses that remove
;; literals.

(defun resolve (x1 x2)
  (let ((r nil))
    (dolist (elt x1)
      (when (and (atom elt)
                 (member (list 'not elt) x2 :test #'equal))
        (setq r (cons (append (remove elt x1)
                              (remove (list 'not elt) x2
                                      :test #'equal))
                      r))))
    (dolist (elt x2)
      (when (and (atom elt)
                 (member (list 'not elt) x1 :test #'equal))
        (setq r (cons (append (remove elt x2)
                              (remove (list 'not elt) x1
                                      :test #'equal))
                      r))))
    r))

(defun convert-to-clause-form (x)
  (cond ((eq (car x) 'not) (list x))
        (t x)))


;; Example:
;;
;; (prove 'r)  ->
;;
;; ((T) ((NOT T) Q) ((NOT S) Q) ((NOT P) (NOT Q) R) (P) ((NOT R))
;; ((NOT P) (NOT Q)) ((NOT Q) R) ((NOT S) (NOT P) R)
;; ((NOT T) (NOT P) R) (Q) ((NOT Q)) ((NOT S) (NOT P))
;; ((NOT T) (NOT P)) ((NOT Q)) ((NOT S) R) ((NOT T) R)
;; ((NOT S) (NOT P)) ((NOT S) R) ((NOT T) (NOT P)) ((NOT T) R)
;; ((NOT P) R) (R) ((NOT P)) ((NOT P) R) NIL)
```

```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                        RTA-STAR SEARCH
                          "rta.lisp"
-----------------------------------------------------------------------------|#

;;-----------------------------------------------------------------------------
;; Structure RTANODE contains information for a single search node for RTA*.
;; It records the projected distance to the goal (h), the distance so far
;; from the start table (g), and the immediate predecessor of the node
;; (parent).

(defstruct rta-node
   h
   g
   parent)


;;-----------------------------------------------------------------------------
;; Constant *HIGHEST-STATIC-VALUE* is used to initialize a variable to find
;; the best successor of a node.

(defconstant *highest-static-value* *infinity*)


;; Variable *VISITED-STATES* is the hash table used by RTA* to remember
;; the states it has passed through.

(defvar *visited-states* nil)
(setq *visited-states* (make-array *hash-table-size*))
(defconstant *hash-table-size* 4000)

;; Variable *NUMBER-OF-STEPS-TAKEN* counts the number of real-time moves
;; made by RTA* during a search.  Because RTA* can return to a previously
;; visited state (looping), *NUMBER-OF-STEPS-TAKEN* may be larger than the
;; number of steps in the solution path that is returned.

(defvar *number-of-steps-taken* nil)


;;-----------------------------------------------------------------------------
;; Function RTA-STAR searches for a solution path from start to goal.  The
;; third parameter, horizon, tells RTA* how many moves it may look ahead
;; locally before deciding on an action to take.  The longer the horizon,
;; the shorter the solution path will be; however, because the search is
;; exponential, shorter paths may take longer to compute.

(defun rta-star (start horizon &optional verbose)
  (clear-hash)
  (setq *number-of-steps-taken* 0)
  (do ((node start) (prev nil))
      ((goal-state? node)
       (adjust-parent-pointers prev node)
       (extract-rta-path node))
    (adjust-parent-pointers prev node)
    (setq prev node)
    (setq node (best-local-step node horizon))
    (when verbose
        (format t "~4d:  h = ~2d.  Move to ~d~%" *number-of-steps-taken*
                (heuristic node) node))
    (setq *number-of-steps-taken* (1+ *number-of-steps-taken*))))


;; Function EXTRACT-RTA-PATH returns a solution path by following parent
;; pointers from the goal state to the start state.

(defun extract-rta-path (node)
   (do ((n node (get-rta-parent n))
        (path nil))
       ((null n) path)
     (setq path (cons n path))))


;; Function ADJUST-PARENT-POINTERS is called by RTA-STAR after each move
;; is made.  If the move was to a node already in the table, its parent
;; pointer is left undisturbed; otherwise its parent pointer is set to the
;; state before the move was made.  Thus, a chain of parent pointers will not
;; include any loops, and RTA-STAR will keep track of the shortest path from
;; the start state to each of the states in the table.

(defun adjust-parent-pointers (prev node)
   (cond ((null prev)
          (set-rta-g node 0)
          (set-rta-parent node nil))
         (t
          (let ((possible-new-g (+ (cost-of-move prev node)
                                   (get-rta-g prev))))
             (when (or (null (get-rta-g node))
                       (< possible-new-g (get-rta-g node)))
                (set-rta-g node possible-new-g)
                (set-rta-parent node prev)))))))


;;-----------------------------------------------------------------------------
;; Function CLEAR-HASH clears the table of visited states.

(defun clear-hash ()
   (dotimes (x *hash-table-size*)
        (setf (aref *visited-states* x) nil)))


;; Function INSERT-TABLE-ENTRY inserts an rta-node into the table of visited
;; states. The hash key is the state itself.

(defun insert-table-entry (s value)
  (let* ((hash-val (mod (hash-state s) *hash-table-size*))
         (p (assoc s (aref *visited-states* hash-val) :test #'eq-states)))
     (cond ((null p)
            (setf (aref *visited-states* hash-val)
                  (cons (cons s value)
                        (aref *visited-states* hash-val))))
           (t
            (setf (cdr p) value)))))


;; Function GET-TABLE-ENTRY retrieves from the hash table the rta-node
;; associated with state s.
```

```
(defun get-table-entry (s)
   (cdr (assoc s (aref *visited-states* (mod (hash-state s) *hash-table-size*))
               :test #'eq-states)))


;;-----------------------------------------------------------------------
;; Functions GET-RTA-G, GET-RTA-H, and GET-RTA-PARENT access fields of an
;; rta-node associated with a state s.
;; Functions SET-RTA-G, SET-RTA-H, and SET-RTA-PARENT modify those fields.

(defun get-rta-g (s)
   (let ((entry (get-table-entry s)))
      (if (null entry) nil (rta-node-g entry))))

(defun get-rta-h (s)
   (let ((entry (get-table-entry s)))
      (if (null entry) nil (rta-node-h entry))))

(defun get-rta-parent (s)
   (let ((entry (get-table-entry s)))
      (if (null entry) nil (rta-node-parent entry))))

(defun set-rta-g (s val)
    (let ((entry (get-table-entry s)))
       (cond ((null entry) (insert-table-entry s (make-rta-node :g val)))
             (t (setf (rta-node-g entry) val)))))

(defun set-rta-h (s val)
    (let ((entry (get-table-entry s)))
       (cond ((null entry) (insert-table-entry s (make-rta-node :h val)))
             (t (setf (rta-node-h entry) val)))))

(defun set-rta-parent (s val)
    (let ((entry (get-table-entry s)))
       (cond ((null entry) (insert-table-entry s (make-rta-node :parent val)))
             (t (setf (rta-node-parent entry) val)))))


;;-----------------------------------------------------------------------
;; Function BEST-LOCAL-STEP chooses a single move from start towards goal,
;; looking horizon levels ahead.  Following the RTA* algorithm, it sets the h
;; value of the current state to the heuristic score of the second best
;; successor.

(defun best-local-step (start horizon)
  (let ((succs (expand start t)))
    (do ((s succs (cdr s))
         (best-succ nil)
         (best-score *highest-static-value*)
         (second-best-score *highest-static-value*))
        ((null s) (mapc #'(lambda (sl)
                            (when (not (eq sl best-succ)) (destroy-state sl)))
                        succs)
                  (set-rta-h start second-best-score)
                  best-succ)
        (let* ((succ (car s))
               (estimate
                 (+ (cost-of-move start succ)
                    (or (get-rta-h succ)
                        (minimin-alpha succ horizon)))))
          (cond ((> best-score estimate)
                 (setq second-best-score best-score)
                 (setq best-score estimate)
                 (setq best-succ succ))
                ((> second-best-score estimate)
                 (setq second-best-score estimate)))))))


;;-----------------------------------------------------------------------
;; Function MINIMIN-ALPHA performs a depth-first search with alpha-pruning.
;; The search is limited to horizon levels deep. Alpha-pruning requires that
;; heuristic values be computed for internal nodes as well as leaf nodes.
;; Duplicate nodes along any given path are not expanded.

(defvar *alpha* nil)

(defun minimin-alpha (start horizon)
  (setq *alpha* *highest-static-value*)
  (let ((depth 0)
        (cost-so-far 0))
    (minimin-alpha-1 start horizon depth cost-so-far nil))
  *alpha*)

(defun minimin-alpha-1 (start horizon depth cost-so-far parents)
   (cond ((goal-state? start)
          (setq *alpha* (min *alpha* cost-so-far)))
         ((= horizon depth)
          (setq *alpha* (min *alpha* (+ cost-so-far (heuristic start)))))
         (t
          (let ((succs (expand start)))
            (do ((s succs (cdr s)))
                ((null s) (mapc #'destroy-state succs) *alpha*)
              (let ((succ (car s)))
                (when (not (member succ parents :test #'eq-states))
                   (let* ((estimate (heuristic succ))
                          (f-value (+ estimate cost-so-far)))
                     (cond ((< f-value *alpha*)
                            (setq *alpha*
                              (min *alpha*
                                   (minimin-alpha-1
                                     succ
                                     horizon
                                     (1+ depth)
                                     (+ cost-so-far (cost-of-move start succ))
                                     (cons start parents))))))))))))))


;;-----------------------------------------------------------------------
;; Function RTA-STATS takes a number of trials, a horizon, and a filename.
;; It performs RTA* a number of times and writes the average results out
;; to the file.  Because of the large number of random decisions RTA* must
;; make, its performance can vary widely, even on the same problem.

(defun rta-stats (trials horizon outfile)
  (let ((*problems* nil))
    (dotimes (i trials) (setq *problems* (cons (generate-problem) *problems*)))
    (with-open-file (ofile outfile :direction :output :if-exists :append
                                   :if-does-not-exist :create)
      (do ((start-time (get-universal-time))
           (n 0 (1+ n))
           (avg-path-length 0)
           (avg-soln-length 0))
          ((= n trials)
           (format ofile "Trials:                    ~d~%" trials)
           (format ofile "Search horizon:            ~d~%" horizon)
           (let ((end-time (get-universal-time)))
             (format ofile "Time:                        ~d min.  ~d sec.~%"
                     (truncate (/ (- end-time start-time) 60))
```

```
                    (mod (- end-time start-time) 60)))
    (format ofile "Average number of steps: ~d~%"
                  (coerce (/ avg-path-length trials) 'float))
    (format ofile "Average solution length: ~d~%~%"
                  (coerce (/ avg-soln-length trials) 'float)))
  (format t "Solving problem ~d of ~d ...~%" (1+ n) trials)
  (let* ((start (nth n *problems*))
         (solution (rta-star start horizon)))
    (format t "  Number of steps: ~d~%" *number-of-steps-taken*)
    (format t "  Solution length: ~d~%" (1- (length solution)))
    (setq avg-path-length
      (+ avg-path-length *number-of-steps-taken*))
    (setq avg-soln-length
      (+ avg-soln-length (1- (length solution))))))))))
```

```
1 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1
1 1 1 0 1 0 1 1 1 1 0 0 1 0 1 1
1 1 0 1 1 0 1 1 1 1 0 0 1 0 1 1
1 1 0 0 0 0 1 1 1 1 0 0 1 0 1 1
1 1 0 0 1 1 1 1 1 1 0 0 1 0 1 1
1 1 0 0 1 0 0 1 1 1 0 0 1 0 1 1
1 1 0 0 1 0 1 0 1 1 0 0 1 0 1 1
0 1 0 1 1 0 0 0 0 0 1 0 1 1 0 0 0
1 1 0 1 1 0 0 0 0 1 0 1 1 0 0 0 0
0 0 0 1 1 0 0 0 0 0 1 0 1 1 0 0 0
0 1 0 1 0 0 0 0 0 0 1 0 1 1 0 0 0
0 1 0 1 1 1 0 0 0 1 0 1 1 0 0 0
0 1 0 1 1 0 1 0 0 1 0 1 1 0 0 0
0 1 0 1 1 0 0 1 0 1 0 1 1 0 0 0
0 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1
1 0 1 1 1 0 0 1 0 0 1 1 1 0 0 1
0 1 1 1 1 0 0 1 0 0 1 1 1 0 0 1
0 0 0 1 1 0 0 1 0 0 1 1 1 0 0 1
0 0 1 0 1 0 0 1 0 0 1 1 1 0 0 1
0 0 1 1 0 0 0 1 0 0 1 1 1 0 0 1
0 0 1 1 1 1 0 1 0 0 1 1 1 0 0 1
0 0 1 1 1 0 1 1 0 0 1 1 1 0 0 1
0 0 1 1 1 0 0 0 0 0 1 1 1 0 0 1
1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1
0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1
1 0 1 0 1 1 1 1 1 1 1 0 1 1 1 1
1 1 0 0 1 1 1 1 1 1 1 0 1 1 1 1
1 1 1 0 0 1 1 1 1 1 1 0 1 1 1 1
1 1 1 0 1 0 1 1 1 1 1 0 1 1 1 1
1 1 1 0 1 1 0 1 1 1 1 0 1 1 1 1
1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1
```

Network structure:	(8 4 8)
Epochs:	100
Test after every N epochs:	10
Learning rate (eta):	0.35
Momentum (alpha):	0.90
Noise?:	0
Training data:	politics.train
Testing data:	politics.test

Network structure:	(2 2 1)
Epochs:	2000
Test after every N epochs:	50
Learning rate (eta):	0.35
Momentum (alpha):	0.90
Noise?:	0
Training data:	or.train
Testing data:	or.test

```
0 1 0 0 1 0 1 1 1 1 0 0 1 0 1 1
1 0 0 0 1 0 1 1 1 1 0 0 1 0 1 1
0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 0
0 1 0 0 1 0 0 0 0 1 0 1 1 0 0 0
1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1
```

```
0 1 1
1 1 1
```

```
0 0 0
0 1 1
1 0 1
1 1 1
```

```
    s-new))

; ---------------------------------------------------------------------
;; Function PRINT-STATE prints a state in a linear sequence of characters,
;; e.g., #< 0 1 2 3 4 5 6 7 8 >.
;;
;; Function PRINT-STATE-ALTERNATE prints a state in two-dimensional format.
;; If you like the second format, rename this function PRINT-STATE.

(defun print-state (s &rest ignore)
   (declare (ignore ignore))
   (format *standard-output* "#<")
   (dotimes (y *puzzle-size*)
      (dotimes (x *puzzle-size*)
         (format *standard-output* "~3d" (tile-in-position s x y))))
   (format *standard-output*  ">"))

(defun print-state-alternate (s &rest ignore)
   (declare (ignore ignore))
   (dotimes (y *puzzle-size*)
      (format *standard-output* "~%")
      (dotimes (x *puzzle-size*)
         (format *standard-output* "~2d" (tile-in-position s x y))))
   (format *standard-output* "~%"))

; ---------------------------------------------------------------------
;; Function CONVERT-LIST-TO-STATE takes a list like '(0 1 2 3 4 5 6 7 8)
;; and creates a state structure out of it.

(defun convert-list-to-state (s)
  (let ((s-new (create-state)))
    (dotimes (x *puzzle-size*)
       (dotimes (y *puzzle-size*)
          (let ((n (+ (* y *puzzle-size*) x)))
             (set-tile-in-position s-new x y (nth n s))
             (set-xcoord s-new (nth n s) x)
             (set-ycoord s-new (nth n s) y))))
  s-new))


; ---------------------------------------------------------------------
;; Function HASH-STATE takes a state and returns some integer. The same
;; state always yields the same integer.

(defvar *primes*
  '(2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73
    79 83 87 89 91 93 97))

(defun hash-state (s)
  (let ((hash-value 0))
    (dotimes (x *puzzle-size*)
      (dotimes (y *puzzle-size*)
        (setq hash-value (+ hash-value (* (tile-in-position s x y)
                                          (nth (+ (* x *puzzle-size*) y)
                                               *primes*))))))
    hash-value))


; ---------------------------------------------------------------------
;; Function GENERATE-PROBLEM returns a list of two elements: first is a
;; randomly generated start state, second is the goal state. The start
;; state is generated by simulating 1000 moves from the goal state.

(defun generate-problem ()
   (generate-random-state 1000))

(defun generate-random-state (n)
   (let ((s (copy-state *goal-state*)))
      (dotimes (i n)
         (let* ((succs (expand s))
                (next (nth (random (length succs)) succs)))
            (destroy-state s)
            (setq s next)
            (dolist (z succs)
               (when (not (eq s z))
                  (destroy-state z)))))
      s))


;; ---------------------------------------------------------------------
;; Variable *GOAL-STATE* is a standard goal configuration: tiles ordered,
;; blank in the upper-left hand corner.

(defun numbers-up-to (n)
    (cond ((= n 0) (list 0))
          (t (cons n (numbers-up-to (1- n))))))

(defvar *goal-state* nil)

(setq *goal-state*
       (convert-list-to-state (reverse (numbers-up-to (1- *puzzle-tiles*)))))

;; Variable *SAMPLE-INITIAL-STATE* is a state very close to the goal state.

(defvar *sample-initial-state*)

(setq *sample-initial-state*
       (generate-random-state 12))

;; ---------------------------------------------------------------------
;; Function GOAL-STATE? returns t if its argument meets the specification for
;; the goal state.

(defun goal-state? (s)
  (eq-states s *goal-state*))
```

```
(defun tile-in-position (s x y)
  (aref (puzzle-state-board s) x y))

(defun xcoord (s tile)
  (aref (puzzle-state-xcoords s) tile))

(defun ycoord (s tile)
  (aref (puzzle-state-ycoords s) tile))

(defun set-tile-in-position (s x y tile)
  (setf (aref (puzzle-state-board s) x y) tile))

(defun set-xcoord (s tile x)
  (setf (aref (puzzle-state-xcoords s) tile) x))

(defun set-ycoord (s tile x)
  (setf (aref (puzzle-state-ycoords s) tile) x))


;; ---------------------------------------------------------------------
;; Function EQ-STATES returns t if the two states s1 and s2 have the same
;; tiles in the same positions. It returns nil otherwise.

(defun eq-states (s1 s2)
  (do ((n 1 (1+ n))
       (fail nil))
      ((or fail (= n *puzzle-tiles*))
       (not fail))
    (when (or (not (= (xcoord s1 n) (xcoord s2 n)))
              (not (= (ycoord s1 n) (ycoord s2 n))))
      (setq fail t))))

;; ---------------------------------------------------------------------
;; Function HEURISTIC returns the Manhattan distance between states s and the
;; goal.  The Manhattan distance is calculated by adding up, for each non-blank
;; tile, the number of horizontal and vertical moves required to move it from
;; its position in s1 to its position in s2.

(defun heuristic (s)
  (manhattan s *goal-state*))

(defun manhattan (s1 s2)
  (do ((n 1 (1+ n))
       (total 0))
      ((= n *puzzle-tiles*) total)
    (setq total (+ total (abs (- (xcoord s1 n) (xcoord s2 n)))
                         (abs (- (ycoord s1 n) (ycoord s2 n)))))))

; ---------------------------------------------------------------------
;; Function EXPAND returns a list of all legal sucessor states of s.  It
;; looks for the position of the blank tile and moves adjacent tiles.
;; An optional parameter t may be supplied  -- in that case, the successors
;; are returned in a scrambled order.

(defun expand (s &optional randomize)
  (let ((blankx (xcoord s *blank-tile*))
        (blanky (ycoord s *blank-tile*))
        (boards nil))
    (when (> blanky 0)
      (setq boards (cons (move-tile s blankx blanky blankx (1- blanky))
                         boards)))
    (when (> blankx 0)
      (setq boards (cons (move-tile s blankx blanky (1- blankx) blanky)
                         boards)))
    (when (< blanky (1- *puzzle-size*))
      (setq boards (cons (move-tile s blankx blanky blankx (1+ blanky))
                         boards)))
    (when (< blankx (1- *puzzle-size*))
      (setq boards (cons (move-tile s blankx blanky (1+ blankx) blanky)
                         boards)))
    (if randomize (random-permute boards)
        boards)))


;; Function RANDOM-PERMUTE mixes up the elements of a list.

(defun random-permute (x)
  (do ((y x)
       (res nil))
      ((null y) res)
    (let ((next (nth (random (length y)) y)))
      (setq res (cons next res))
      (setq y (delete next y)))))


;; ---------------------------------------------------------------------
;; Function COST-OF-MOVE takes a state and its successor and returns the
;; cost of making that transition.  In the case of the n-puzzle, the
;; cost of sliding a tile is always 1.

(defun cost-of-move (state successor)
  (declare (ignore state successor))
  1)


;; ---------------------------------------------------------------------
;; Function MOVE-TILE takes a state (s), a set of coordinates indicating
;; the location of the blank (bx/by), and a set of coordinates indicating
;; the location of the tile to be moved (tx/ty). It copies the state and
;; moves the tile, returning a new state.

(defun move-tile (s bx by tx ty)
  (let ((c (copy-state s)))
    (swap-tiles c bx by tx ty)))

(defun swap-tiles (s bx by tx ty)
  (let ((tile (tile-in-position s tx ty)))
    (set-tile-in-position s tx ty *blank-tile*)
    (set-tile-in-position s bx by tile)
    (set-xcoord s *blank-tile* tx)
    (set-ycoord s *blank-tile* ty)
    (set-xcoord s tile bx)
    (set-ycoord s tile by)
    s))

; ---------------------------------------------------------------------
;; Function COPY-STATE creates and returns a new state structure which is
;; a copy of its input.

(defun copy-state (s)
  (let ((s-new (create-state)))
    (dotimes (x *puzzle-size*)
      (dotimes (y *puzzle-size*)
        (set-tile-in-position s-new x y (tile-in-position s x y))))
    (dotimes (x *puzzle-tiles*)
      (set-xcoord s-new x (xcoord s x))
      (set-ycoord s-new x (ycoord s x)))
```

```
#|------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------
                          TIC-TAC-TOE GAME
                          "tictactoe.lisp"
------------------------------------------------------------------------|#

#|------------------------------------------------------------------------

This file contains code for the game of tic-tac-toe.
The important functions are:

 (deep-enough pos depth)        t if the search has proceeded deep enough.
 (static pos player)            evaluation of position pos from player's
                                 point of view.
 (movegen pos player)           generate all successor positions to pos.
 (opposite player)              return the opposite player.
 (print-board pos)              print board position pos.
 (make-move pos player move)    return new position based on old position and
                                 player's move.
 (won? pos player)              t if player has won.
 (drawn? pos)                   t if pos is a drawn position.

The important variables are:

 *start*                        the initial board configuration.

These functions and variables are all called from minimax.lisp.

------------------------------------------------------------------------|#

;; Function NULL-BOARD creates an empty tic-tac-toe board. The board is
;; stored as a list of nine elements.  Elements are either 'x, 'o, or nil
;; (empty).

(defun null-board ()
  (list nil nil nil nil nil nil nil nil nil))

;; Variable *START* is the starting board position.

(defvar *start* nil)
(setq *start* (null-board))


;; Function MAKE-MOVE takes a board position (pos), a player (player, which
;; is 'x or 'o), and a move (which is a number between 0 and 8). It returns
;; a new board position.

(defun make-move (pos player move)
  (let ((b (copy-list pos)))
    (setf (nth move b) player)
    b))


;; Function MOVEGEN takes a position and a player and generates all legal
;; successor positions, i.e., all possible moves a player could make.

(defun movegen (pos player)
  (mapcan
    #'(lambda (m)
        (if (null (nth m pos))
            (list (make-move pos player m))
            nil))
    '(0 1 2 3 4 5 6 7 8)))


;; Function WON? returns t is pos is a winning position for player,
;; nil otherwise.

(defun won? (pos player)
  (or (and (eq (first pos) player)
           (eq (second pos) player)
           (eq (third pos) player))
      (and (eq (fourth pos) player)
           (eq (fifth pos) player)
           (eq (sixth pos) player))
      (and (eq (seventh pos) player)
           (eq (eighth pos) player)
           (eq (ninth pos) player))
      (and (eq (first pos) player)
           (eq (fourth pos) player)
           (eq (seventh pos) player))
      (and (eq (second pos) player)
           (eq (fifth pos) player)
           (eq (eighth pos) player))
      (and (eq (third pos) player)
           (eq (sixth pos) player)
           (eq (ninth pos) player))
      (and (eq (first pos) player)
           (eq (fifth pos) player)
           (eq (ninth pos) player))
      (and (eq (third pos) player)
           (eq (fifth pos) player)
           (eq (seventh pos) player))))

;; Function DRAWN? returns t if pos is a drawn position, i.e., if there are
;; no more moves to be made.

(defun drawn? (pos)
  (not (member nil pos)))


;; Function OPPOSITE returns 'x when given 'o, and vice-versa.

(defun opposite (player)
  (if (eq player 'x) 'o 'x))


;; Function STATIC evaluates a position from the point of view of a
;; particular player.  It returns a number -- the higher the number, the
;; more desirable the position.  The simplest static function would be:
;;
;;      (defun static (pos player)
;;        (cond ((won? pos player) 1)
;;              ((won? pos (opposite player)) -1)
;;              (t 0)))
;;
;; However, this heuristic suffers from the problem that minimax search
;; will not "go in for the kill" in a won position.  The following static
```

```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                            TERM UNIFICATION
                           "term-unify.lisp"
-----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------

The main call is:

        (unify l1 l2)                   unifies two terms

See examples at the end of this file.

-----------------------------------------------------------------------------|#

;; Function UNIFY returns a set of substitutions that make its two input
;; terms identical.  Terms are represented as lists.  For example, the
;; term f(x,g(a)) is represented as (f x (g a)).


(defun unify (l1 l2)
  (cond ((or (constant? l1) (variable? l1)
             (constant? l2) (variable? l2))
         (cond ((eq l1 l2) nil)
               ((variable? l1)
                (if (contains l2 l1) 'fail (list (list l1 '<- l2))))
               ((variable? l2)
                (if (contains l1 l2) 'fail (list (list l2 '<- l1))))
               (t 'fail)))
        ((or (not (eq (car l1) (car l2)))
             (not (= (length l1) (length l2))))
         'fail)
        (t
         (do ((subst nil)
              (args1 (cdr l1) (cdr args1))
              (args2 (cdr l2) (cdr args2))
              (failed? nil))
             ((or failed? (null args1))
              (if failed? 'fail (nreverse subst)))
           (let ((s (unify (car args1) (car args2))))
             (cond ((eq s 'fail) (setq failed? t))
                   ((null s) nil)
                   (t (setq args1 (apply-substitution s args1))
                      (setq args2 (apply-substitution s args2))
                      (setq subst (append s subst)))))))))

(defun constant? (a)
  (and (atom a)
       (not (null a))
       (not (variable? a))))

(defun variable? (a)
  (and (atom a)
       (member a '(x y z x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12))))

(defun contains (tree item)
  (cond ((null tree)
         nil)
        ((atom tree)
         (equal tree item))
        (t                          ;  tree is a list
         (or (contains (car tree) item)
             (contains (cdr tree) item)))))


(defun apply-substitution (substs expr)
  (let ((c (copy-tree expr)))
    (mapc #'(lambda (s)
              (setq c (nsubst (third s) (first s) c)))
          substs)
    c))


;; (unify '(f x (g a))  '(f (g a) (g y))) ->
;;  ((X <- (G A)) (Y <- A))
```

```
#|-----------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
                        RAILROAD TRAVEL DOMAIN
                            "travel.lisp"
----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------
This file contains code for a city-to-city railroad travel problem.  The
task is to travel from San Francisco to Los Angeles, perhaps usign the
shortest path.  The important functions are (s, s1 and s2 are states):

 (goal-state? s)              t if s is a goal state
 (eq-states s1 s2)            are s1 and s2 equal?
 (expand s)                   successor states of s
 (hash-state s)               some has value for s
 (print-state s)              print function
 (destroy-state s)            dispose of state structure
 (heuristic s)                heuristic distance between s and the goal
 (generate-problem)           randomly generated start-state
 (cost-of-move s1 s2)         cost of moving from s1 to s2

The important variables are:

 *sample-initial-state*
 *goal-state*

These functions and variables can all be called from an outside search program.
In fact, these are the functions called by our implementations of depth-first,
breadth-first, hill-climbing, A*, DFID, IDA*, and RTA* search.

The problems in this domain are not particularly difficult to solve, but
clearly illustrate different search strategies.  To watch different strategies
in action, load the files "dfs.lisp, "bfs.lisp", "dfid.lisp", "hill.lisp",
"a-star.lisp", and "rta-star.lisp" and execute the following commands:

        (dfs 'san-francisco t)
        (dfs-avoid-loops 'san-francisco t)
        (bfs-graph 'san-francisco t)
        (dfid 'san-francisco t)
        (ida-star 'san-francisco t)
        (a-star 'san-francisco t)
        (hill-climbing 'san-francisco t)
        (rta-star 'san-francisco 1 t)

----------------------------------------------------------------------------|#

#|-----------------------------------------------------------------------------

                    MAP OF CITIES AND RAIL DISTANCES


                                                                   *BOS

                                           272   *DET                    229
                                        *CHI
*SEA                                                                     *NY
                                          184     303
                                                                          91

900                                          *IND   371   *PIT  348   *PHI
                                                                      133

                                                                      *WAS
*SF    821     *SLC                       518

                         *TPK
470            700           60
                               *KC                                761

*LA  450
        *PHX                          517                  *BIR
               440
                   *EP    646    *DAL                   355
                                            *NO                  804
                    827     264         363
                                  *HOU

                                                                   *MIA
-----------------------------------------------------------------------------|#

;; Variable *INFINITY* will be used as the largest possible number.
;; MOST-POSITIVE-FIXNUM is a Lisp symbol that provides it.

(defvar *infinity* most-positive-fixnum)

;;-----------------------------------------------------------------------------
;; Variable *CITIES* holds the list of cities that will be used as states
;; in the search.
;;
;; Variable *NUMBER-OF-CITIES* is the total number of cities.
;;
;; Variable *RAIL-DISTANCES* is a list of adjacent cities and their distances
;; by rail.
;;
;; Variable *AIRLINE-DISTANCES* stores, for each pair of cities, the Euclidean
;; distance between them.

(defvar *cities*)
(setq *cities*
  '(san-francisco  los-angeles  salt-lake-city  phoenix  el-paso  houston
    dallas  kansas-city  topeka  new-orleans  birmingham  miami  washington
    philadelphia  pittsburgh  indianapolis  chicago  detroit  new-york
    boston  seattle))

(defvar *number-of-cities*)
(setq *number-of-cities* (length *cities*))

(defvar *rail-distances*)
(setq *rail-distances*
  '(
    (san-francisco (seattle 900) (salt-lake-city 821) (los-angeles 470))
    (los-angeles (san-francisco 470) (phoenix 450))
    (salt-lake-city (san-francisco 821) (phoenix 450))
    (phoenix (el-paso 440) (salt-lake-city 700) (los-angeles 450))
    (el-paso (phoenix 440) (houston 827) (dallas 646))
    (houston (el-paso 827) (new-orleans 363) (dallas 264))
    (dallas (el-paso 646) (kansas-city 517) (houston 264))
```

```
;; function weighs quick wins more highly than slower ones; it also
;; ranks quick losses more negatively than slower ones, allowing the
;; program to "fight on" in a lost position.

(defun static (pos player)
  (cond ((won? pos player)
         (+ 1 (/ 1 (length (remove nil pos)))))
        ((won? pos (opposite player))
         (- (+ 1 (/ 1 (length (remove nil pos))))))
        (t 0)))


;; Function DEEP-ENOUGH takes a board position and a depth and returns
;; t if the search has proceeded deep enough.  The implementation below
;; causes the search to proceed all the way to a finished position.  Thus,
;; minimax search will examine the whole search space and never make a
;; wrong move.  A depth-limited search might look like:
;;
;;       (defun deep-enough (pos depth)
;;         (declare (ignore pos))
;;         (if (> depth 3) t nil))

(defun deep-enough (pos depth)
  (declare (ignore depth))
  (or (won? pos 'x)
      (won? pos 'o)
      (drawn? pos)))


;; Function PRINT-BOARD prints a two-dimensional representation of the board.

(defun print-board (b)
  (format t "~% ~d ~d ~d~% ~d ~d ~d~% ~d ~d ~d~%~%"
          (or (first b) ".") (or (second b) ".") (or (third b) ".")
          (or (fourth b) ".") (or (fifth b) ".") (or (sixth b) ".")
          (or (seventh b) ".") (or (eighth b) ".") (or (ninth b) ".")))
```

```
(defun heuristic (s)
   (aref *airline-distances*
         (position s *cities*)
         (position *goal-state* *cities*)))

(defun destroy-state (s)
  (declare (ignore s))
  nil)

(defun generate-problem ()
  (nth (random *number-of-cities*) *cities*))

(defun cost-of-move (s1 s2)
  (cadr (assoc s2
               (cdr (assoc s1 *rail-distances*)))))
```

```
    (kansas-city (topeka 60) (indianapolis 518) (dallas 517))
    (topeka (kansas-city 60))
    (new-orleans (houston 363) (birmingham 355))
    (birmingham (washington 761) (new-orleans 355) (miami 804))
    (miami (birmingham 804))
    (washington (philadelphia 133) (birmingham 761))
    (philadelphia (washington 133) (pittsburgh 348) (new-york 91))
    (pittsburgh (philadelphia 348) (indianapolis 371))
    (indianapolis (pittsburgh 371) (kansas-city 518) (detroit 303)
                  (chicago 184))
    (chicago (indianapolis 184) (detroit 272))
    (detroit (indianapolis 303) (chicago 272))
    (new-york (philadelphia 91) (boston 229))
    (boston (new-york 229))
    (seattle (san-francisco 900))
  ))

(defvar *airline-distances*)
(setq *airline-distances*
  (make-array (list *number-of-cities* *number-of-cities*)
    :initial-contents
      '(
        (   0  347  600  653 1202 1645 1483 1506 1456 1926 2385
         2594 2442 2523 2264 1949 1858 2091 2571 2699  678)

        ( 347    0  579  357  800 1374 1240 1356 1306 1673 2078
         2339 2300 2394 2136 1809 1745 1983 2451 2596  959)

        ( 600  579    0  504  877 1200  999  925  885 1434 1805
         2089 1848 1925 1668 1356 1260 1492 1972 2099  701)

        ( 653  357  504    0  402 1017  887 1049 1009 1316 1680
         1982 1983 2083 1828 1499 1453 1690 2145 2300 1114)

        (1202  800  877  402    0  756  625  936  896 1121 1278
         1957 1997 2065 1778 1418 1439 1696 2147 2358 1760)

        (1645 1374 1200 1017  756    0  225  644  664  318  692
          968 1220 1341 1137  865  940 1105 1420 1605 1891)

        (1483 1240  999  887  625  225    0  451  486  443  653
         1111 1185 1299 1070  763  803  999 1374 1551 1681)

        (1506 1356  925 1049  936  644  451    0   50  680  703
         1241  945 1038  781  453  414  645 1097 1251 1506)

        (1456 1306  885 1009  896  664  486   50    0  720  753
         1291  995 1088  821  473  424  665 1137 1291 1466)

        (1926 1673 1434 1316 1121  318  443  680  720    0  347
          892 1098 1241 1118  839  947 1101 1330 1541 2606)

        (2385 2078 1805 1680 1278  692  653  703  753  347    0
          777  751  894  792  492  657  754  983 1194 2612)

        (2594 2339 2089 1982 1957  968 1111 1241 1291  892  777
            0 1101 1239 1250 1225 1390 1409 1328 1539 3389)

        (2442 2300 1848 1983 1997 1220 1185  945  995 1098  751
         1101    0  123  192  494  597  396  205  393 2329)

        (2523 2394 1925 2083 2065 1341 1299 1038 1088 1241  894
         1239  123    0  259  585  666  443   83  271 2380)

        (2264 2136 1668 1828 1778 1137 1070  781  821 1118  792
         1250  192  259    0  330  410  205  317  483 2183)

        (1949 1809 1356 1499 1418  865  763  453  473  839  492
         1225  494  585  330    0  165  240  646  807 1872)

        (1858 1745 1260 1453 1439  940  803  414  424  947  657
         1390  597  666  410  165    0  279  840  983 2052)

        (2091 1983 1492 1690 1696 1105  999  645  665 1101  754
         1409  396  443  205  240  279    0  671  702 2339)

        (2571 2451 1972 2145 2147 1420 1374 1097 1137 1330  983
         1328  205   83  317  646  840  671    0  213 2900)

        (2699 2596 2099 2300 2358 1605 1551 1251 1291 1541 1194
         1539  393  271  483  807  983  702  213    0 3043)

        ( 678  959  701 1114 1760 1891 1681 1506 1466 2606 2612
         3389 2329 2380 2183 1872 2052 2339 2900 3043    0)
      )))

;; ------------------------------------------------------------------------


(defvar *sample-initial-state* 'san-francisco)

(defvar *goal-state* 'new-york)

;; ------------------------------------------------------------------------
;; Functions required by search programs in the railroad travel domain.

(defun goal-state? (s)
  (eq-states s *goal-state*))

(defun eq-states (s1 s2)
  (eq s1 s2))

(defun expand (s &optional randomize)
  (let ((succs (mapcar #'car (cdr (assoc s *rail-distances*)))))
    (if randomize
        (random-permute succs)
        succs)))


;; Function RANDOM-PERMUTE mixes up the elements of a list.

(defun random-permute (x)
  (do ((y x)
       (res nil))
      ((null y) res)
    (let ((next (nth (random (length y)) y)))
      (setq res (cons next res))
      (setq y (delete next y)))))


(defun hash-state (s)
   (sxhash s))          ; built-in Common Lisp function

(defun print-state (s &rest ignore)
   (declare (ignore ignore))
   (format *standard-output* "~d" s))
```

```
(setq *johns-car*
   '((origin japan) (mfr honda) (color blue) (decade 1980) (type economy)))

(setq *marys-car*
   '((origin japan) (mfr toyota) (color green) (decade 1970) (type sports)))

(setq *herbs-car*
   '((origin japan) (mfr toyota) (color blue) (decade 1990) (type economy)))

(setq *allens-car*    ; corrected from first printing (blue, not red)
   '((origin usa) (mfr chrysler) (color blue) (decade 1980) (type economy)))

(setq *marvins-car*
   '((origin japan) (mfr honda) (color white) (decade 1980) (type economy)))

(setq *attribute-values*
   '((origin japan usa britain germany italy)
     (mfr honda toyota ford chrysler jaguar bmw fiat)
     (color blue green red white)
     (decade 1950 1960 1970 1980 1990 2000)
     (type economy luxury sports)))

(defun all-attribute-values (attrib)
  (cdr (assoc attrib *attribute-values*)))

(defun all-attribute-names ()
  (mapcar #'car *attribute-values*))


;; To test, evaluate the following five expressions:
;;
;; (initialize-s-and-g *johns-car*)
;; (accept-instance *marys-car* nil)
;; (accept-instance *herbs-car* t)
;; (accept-instance *allens-car* nil)
;; (accept-instance *marvins-car* t)
```

```
#|---------------------------------------------------------------------------
Artificial Intelligence, Second Edition
Elaine Rich and Kevin Knight
McGraw Hill, 1991

This code may be freely copied and used for educational or research purposes.
All software written by Kevin Knight.
Comments, bugs, improvements to knight@cs.cmu.edu
---------------------------------------------------------------------------|#

#|---------------------------------------------------------------------------
            CANDIDATE ELIMINATION ALGORITHM FOR VERSION SPACES
                              "vs.lisp"
---------------------------------------------------------------------------|#

;; Version space learning with the candidate elimination algorithm.
;; Concepts are represented as lists of attribute-value pairs.  See
;; bottom of file for examples.

(defvar *s-set*)
(defvar *g-set*)
(defvar *attribute-values*)
(defvar *most-general-concept* nil)


;; Function INITIALIZE-S-AND-G accepts a positive instance of the concept
;; to be learned, and initializes the S and G sets.

(defun initialize-s-and-g (first-positive-example)
  (setq *g-set* (list *most-general-concept*))
  (setq *s-set* (list (order-attributes first-positive-example)))
  (status-report))

;; Function ACCEPT-INSTANCE incrementally accepts positive (t) and negative
;; (nil) instances of the target concept.

(defun accept-instance (inst positive?)
  (if positive?
      (accept-positive-instance inst)
      (accept-negative-instance inst))
  (status-report))

(defun accept-positive-instance (inst)
  (setq *g-set* (remove-if #'(lambda (g-elt)
                               (not (more-general-than g-elt inst)))
                           *g-set*))
  (setq *s-set* (mapcar #'(lambda (s-elt)
                            (generalize s-elt inst))
                        *s-set*)))

(defun accept-negative-instance (inst)
  (setq *s-set* (remove-if #'(lambda (s-elt)
                               (more-general-than s-elt inst))
                           *s-set*))
  (setq *g-set* (mapcan #'(lambda (g-elt)
                            (specialize g-elt inst))
                        *g-set*)))

(defun status-report ()
  (format t "~%~%New S set: ~d. ~%~%" *s-set*)
  (format t "New G set: ~d. ~%~%" *g-set*)
  (when (equal *s-set* *g-set*)
     (format t "Done. Learned concept = ~d. ~%~%" (car *s-set*)))
  (when (or (null *s-set*) (null *g-set*))
     (format t "Inconsistent data. ~%~%")))


;; Function MORE-GENERAL-THAN returns t iff concept C1 is a more general
;; description than C2.

(defun more-general-than (c1 c2)
   (subsetp c1 c2 :test #'equal))

;; Function ORDER-ATTRIBUTES reorders the attributes in a concept to an
;; alphabetical standard.

(defun order-attributes (c)
  (sort (copy-tree c)
        #'(lambda (attr1 attr2)
            (string<
               (format nil "~d" (car attr1))
               (format nil "~d" (car attr2))))))

;; Function GENERALIZE returns the generalization of two concepts.

(defun generalize (s-elt inst)
   (order-attributes (intersection s-elt inst :test #'equal)))

;; Function SPECIALIZE returns a list of specializations of G-ELT that do
;; not cover the negative instance INST.

(defun specialize (g-elt inst)
  (mapcar #'order-attributes
          (let* ((attributes-to-specialize
                    (set-difference (all-attribute-names)
                                    (mapcar #'car g-elt)
                                    :test #'equal))
                 (possible-specializations
                    (mapcan #'(lambda (attrib)
                                (mapcar #'(lambda (value)
                                            (cons (list attrib value)
                                                  g-elt))
                                        (all-attribute-values attrib)))
                            attributes-to-specialize)))
            (remove-if #'(lambda (potential-g-elt)
                           (or (not (some #'(lambda (s-elt)
                                              (more-general-than
                                                 potential-g-elt
                                                 s-elt))
                                          *s-set*))
                               (more-general-than potential-g-elt inst)))
                       (cons g-elt possible-specializations)))))


;; ---------------------------------------------------------------------------
;;  EXAMPLE
;;
;;  Positive and negative examples of the concept "Japanese economy car".
;;  cf. page 467.

(defvar *johns-car*)     ; +
(defvar *marys-car*)     ; -
(defvar *herbs-car*)     ; +
(defvar *allens-car*)    ; -
(defvar *marvins-car*)   ; +
```

Network structure:	(2 2 1)
Epochs:	2000
Test after every N epochs:	50
Learning rate (eta):	0.35
Momentum (alpha):	0.90
Noise?:	0
Training data:	xor.train
Testing data:	xor.test

```
0 1 1
1 1 0
```