Sudoku Puzzles as a Constraint Satisfaction Problem.

4-18-2006
Aaron Christensen - chri0800@umn.edu
Gabe Emerson - emers089@umn.edu

Abstract.

In this paper, we explore methods of solving Sudoku logic puzzles using constraint satisfaction algorithms. Sudoku is commonly solved by brute-force methods, but is very well suited to CSP programming and can be solved much more efficiently by such methods. We describe our approaches to the problem, algorithmic and representational alternatives considered, and the results of our test code. We determined that the CSP approach greatly improves the efficiency of Sudoku solvers, and more closely models the intuitive solution methods used by humans.

Introduction.

Sudoku puzzles are a popular logic exercise found in many newspapers and other publications. The object is to complete a partially filled out Latin Square of size n^2 x n^2, in which every row, column, and internal nxn square contains unique values. While the solution does not directly involve mathematical manipulation (colors, letters, or symbols can be used instead of numbers), it is possible to find a solution through a variety of different search algorithms. The stipulations of the puzzle lend themselves especially well to constraint variable treatment, as the value of any particular grid element depends on the values of other elements in the same row, column, and square. A grid element's value can be inferred or derived from the known restrictions (previously filled elements) and remaining possibilities.

The majority of existing programmatical solutions utilize brute-force trial and error to complete Sudoku puzzles, with most interest focusing on solutions in different languages rather than improving efficiency. A wide variety of brute force and other simple solutions can be found on websites and message boards such as (1). Such forums are also an excellent source of research into exotic Sudoku styles, different approaches to element and grid representation, and puzzle-generation implementations. Our approach attempts to improve on the efficiency of Sudoku solution-finding, while exploring the application of constraint satisfaction algorithms.

The most common version of Sudoku has a 9x9 board with internal 3x3 squares, where possible elements are the digits 1-9. Other versions can include interlocking boards, non-square internal fields, or further color/shape/pattern restrictions, but the most interesting programmatically are simply variations on the size of the board. Our goal was to not only solve a standard nine by nine board, but to develop a program capable of solving size n by n boards.

Approach.

We chose to look at the Sudoku CSP as a variation of the standard n-queens problem, using example code found in the AIMA online resources as a framework[3] The problems are very similar, with both constraining element placement on a board based upon the placement of preexisting or pre-chosen elements. Essentially, one can represent a single unique value in a Sudoku board (such as a five) as one of the queens in the n-queens problem. All other fives on the board are then treated as other queens, the goal being to place all fives so that none is in the same row, column, or square as another. This

solution can be repeated for each possible value. Our research indicated that backtracking search is one of the most efficient for solving N-queens problems, so our primary focus was on adapting such an algorithm to the Sudoku domain.

Our code includes two test driver boards as well as a facility for solving newly created boards. The function Sudoku-solve can be called with an order two board (n=2) by running (Sudoku-solve s2), or with an order three board by running (Sudoku-solve s3). To solve a randomly generated puzzle, the function is called as (Sudoku-solve (Sudoku-make-puzzle)). As an experiment in timing board generation, a third driver containing all nil values (blank board) is available as s4.

The program defines an initial state consisting of a Sudoku board, represented by a list with elements in known order. This board can be predefined as a text list or generated by an external function. While it is possible to generate Latin squares from an initially blank board with our code, we did not implement any randomization and boards generated this way are always the same. Due to time constraints we decided to use a third-party generator, adding only a translation function to adapt the output to our internal representation. The original generator code was developed by Christopher Bowron and available freely online [2].

After defining the initial state, the program notes assigned and unassigned values for a given domain (row, column, square), and attempts to find the first value which can legally fill an element. At the heart of the program, the constraint function then checks each instance of the chosen value in the rest of the puzzle, to ensure that the locations of other instances do not conflict with the current element. The check-row, check-column, and check-square functions define how each element is located within domains, and which elements are members of the same domain. Defining this indexing method was one of the more difficult and interesting parts of the code.

Internal data for the Sudoku board is represented as a list, with the elements' domain membership defined as described. Unchosen (empty) elements are represented by nil, while chosen elements simply contain the chosen numeric value. A complete board has the form (n a_0 a_1 ... a_n^4 – 1), where a_i is the index to a row-based ordering of elements, and n is the size, or order, of the board. For example, our second-order test driver is stored as (2 1 nil 3 4 nil 4 nil nil nil 3 2 nil 2 nil 4 nil). There are many possible ways of representing the board data, including arrays, nested lists, and assorted representations for blank elements, but this format was chosen for simplicity.

Locating a list element within it's proper row, column, and square is done by several formulas that we designed. For rows, R_j(i) locates the jth element in the row beginning with i, as so: R_0(i) = i - (i % n^2), and R_j(i) = R_0(i) + j, where 0 <= j < n^2. For columns, C_j(i) locates the jth element in the column beginning with i: C_0(i) = i % n^2, and C_j(i) = C_0(i) + j*n^2, where 0 <= j < n^2. Internal squares are more difficult to define, and required a more complex calculation. The formula S_k,j(i) locates the kth element in the jth row of the square that contains i: S_0,0(i) – ((i% n^2) – (i % n)) + (i- (i % n^2)) - (((i / n^2) % n) * n^2)), and S_k,j(i) = S_0,0(i) + k + j*n^2, where 0 <= k < n and 0 <= j <n.

We were unable to modify the chosen generator code sufficiently to produce arbitrary order puzzles, so we are only able to test with order n=3 (nine by nine) size boards, and the single n=2 size test board we created. Generation and translation time is prohibitive for creating a fourth order puzzle by hand.

We originally planned to implement minimum-remaining-value and forward checking into the backtracking search, and to compare the relative performance of each

method. Time and codebase constraints forced us to complete the program with only a simple backtracking search and forward checking implemented. Much of the code for implementing MRV is in place and could be completed with some additional time and effort.

Evaluation.

Efficiency of the Sudoku solver is determined by the code's run time. As an additional metric, the number of nodes expanded by the N-queens backtracking search is also listed for each solution. We compared the execution time of our code to the execution time of another lisp-based Sudoku solver, also written by Christopher Bowron [2]. We do not use any of Bowron's solution algorithms, simply calling his board generator to pass new random boards into our program. Bowron's solution utilizes a random hill-climbing approach, which we assumed to be slower than backtracking search. By timing both approaches using lisp's built-in time() method, we hoped to confirm this hypothesis in an unbiased manner. A small test function, Sudoku-test, runs each solution on a newly generated puzzle and compares their execution time.

In addition, we timed our program's generation of a new Latin square from a initially blank board, keeping in mind that without randomization we would only get a single sample. This angle of investigation was implemented late in the project, and is included out of academic interest. Without further development, it is probably not reasonable to base any conclusions off the board creation performance of our code vs other generators.

Testing and function timing was done on an ITlabs Linux machine (Ubuntu 5.04) with a 3.4ghz P4 CPU and 1gb of RAM. All time values are taken as estimated averages, as multiple runs varied slightly based on CPU and memory load. These values are likely to be system-dependent as well, but are assumed to be a good comparison of the execution time of different functions.

Results.

Our solver takes approximately 0.93 seconds of real time and 0.74 seconds of runtime to solve the third-order sample driver s3. It takes about 0.012 seconds real time and 0.007 seconds run time to solve the order 2 sample. Since these runtimes test performance on only a single pre-generated puzzle, it is more interesting to run the solver against a number of randomly generated puzzles and record the average time to reach a solution, since each puzzle will be a different level of difficulty to solve.

Run	***Our code (Sudoku-solve)***	***Bowron's code (solve-puzzle)***
1	real: 155.68, run: 96.47	real: 17.7, run: 8.82
2	real: 537.07, run: 344.09	real: 220.98, run: 143.18
3	real: 103.46, run: 88.01	real: 7.40, run: 6.34
4	real: 38.22, run: 29.91	real: 0.83, run: 0.70
5	real: 26.49, run: 22.49	real: 4.33, run: 3.67

Several sample runs of Sudoku-test, showing variable relative performance.

With our program generating a new board, (time(Sudoku-solve s4) returned on average a real time of : 1.50 seconds and a run time of approximately 1.19 seconds. Christopher Bowron's code was run as a pure Latin-square generator by calling (time(solve-puzzle-fast *blank*)) to generate a new board. This returned a real time of approximately 0.33 seconds and a runtime of approximately 0.30 seconds.

Conclusions:

Our code performed surprisingly poorly when compared to the Sudoku solver implemented by Christopher Bowron. Solution times varied greatly depending on the difficulty of the puzzle being generated, but in general our solver took at least twice as long to find a solution. The variation in relative performance indicates that each approach is more or less suited to certain initial state puzzles. Our code also took longer to generate new puzzles from a blank initial state, but we have too little data to determine what the average performance difference is.

We are currently unsure why our solving algorithm performs so poorly against what should be a slower method of solving Sudoku. One possibility is that the two algorithms have different critical thresholds, or points where a certain number of predetermined elements makes finding remaining elements easier. We could test this hypothesis more fully by implementing a randomized blank-board solver (puzzle or Latin square generator) , but unfortunately time constraints keep us from investigating this further.

Another possibility is that Bowron's method of optimization is better at completing domains than our algorithm. His code attempts to solve domains recursively, so while it is essentially a randomized approach, it may solve individual rows, columns, and squares faster to attain an overall higher efficiency.

It is probable that adding a minimum-remaining-value algorithm to our code would improve the speed and performance. We initially planned to implement this algorithm, but were unable to fit it into the AIMA CSV framework adequately. With more development time, it is likely that this solver could be improved to match or exceed the performance of other solvers using randomized brute-force and other algorithms.

Sources Cited:

1. Sudoku Programmer's Forum.
http://www.setbb.com/phpbb/index.php?mforum=Sudoku

2. Bowron, Chris. Sudoku In Lisp.
http://www.setbb.com/phpbb/viewtopic.php?t=195&highlight=lisp&mforum=sudoku

3. Russell, Stuart, and Peter Norvig. Artificial Intelligence, a Modern Approach. Pearson Education, New Jersy 2003. Online Code Repository,
http://aima.cs.berkeley.edu/lisp/doc/overview-SEARCH.html

Other Sources:

Yato, Takayuk, and Takarior Seta. Complexity and Completeness of Finding Another Solution and Its Application to Puzzles. Masters thesis, Unversity of Tokyo, Department of Information Science, 2003. , Jan 2003

Appendix A: Installation and execution notes.

Extract the bzip archive to the desired location, and edit the file “aima.lisp” within the “aima_clisp” directory so that line 15 contains the path to the current location. From the “proj” directory, run clisp and type (load “proj.lisp”). This script loads all the necessary files for our sudoku solver. During loading, the aima framework files cause a continuable error, which can be skipped by typing “continue” at the prompt. The other components will load without issue. Our code was developed and run in a Windows XP cygwin environment, and was tested and benchmarked on an IT Labs Ubuntu linux machine.

Puzzles can be generated and translated into our input format by typing (sudoku-make-puzzle), and our solver can be called by (sudoku-solve). To solve a randomly generated new puzzle, call (sudoku-solve (sudoku-make puzzle)). To test our solver with the built-in drivers, call (sudoku-solve s2) for the 2nd order puzzle, and (sudoku-solve s3) for the thrid-order puzzle.

Appendix B: Project code.

Directories and files in bzip archive "proj" are:
<aima_clisp>: AIMA framework and CSP functions from
http://aima.cs.berkeley.edu/lisp/
<Sudoku>: solution, generator, and translation functions.
proj.lisp: loading script to call necessary files.

<Sudoku> contains:
Sudoku-generator.lisp: Christopher Bowron's generator and solver used for comparisons.
sudoku.lisp: Main functions
Sudoku-problem.lisp: Development notes.

All files (except the Aima framework available online) are included here.
The file "aima.lisp" within the aima_clisp directory must be edited so that line 15 contains the path to the current installed location.

proj.lisp

```
;;;; Simple lisp file to load necessary stuff.

; Load the AIMA framework
(load "aima_clisp/aima.lisp")

; Load the  search stuff.  Seems to bug out, so needs a 'continue'
(aima-load 'search)
```

```
; Load our sudoku-problem implementation.  This plugs into the AIMA framework
(load "sudoku/sudoku-problem.lisp")

; Load the Christopher Bowron's generator code
(load "sudoku/sudoku-generator.lisp")

; Load some auxillary functions to run and print things.
(load "sudoku/sudoku.lisp")
```

sudoku.lisp

```
(defun print-sudoku (sudoku-list)
  "Print out the primitive sudoku list.  It is of the form:
   (n i0 i1 i2 ... in^4-1) where n is the order and i are the
   indices."
  (let* ((n (first sudoku-list))
         (board (rest sudoku-list))
         (n^4 (expt n 4))
         (n^2 (expt n 2))
         (nn^2 (* n n^2))
         (a (log (loop with e = 10  ; a is max len of n^2 in chars
                   when (< n^2 e)
                     return e
                   do (setq e (* e 10)))
                 10))
         (f (concatenate 'string "~" (format nil "~d" a) "d"))
                                    ; format string for vals
         (d (+ 1 (* n^2 (+ 1 a))))) ; a*n^2 + (n + 1) + (n*(n -1))

    (loop for i from 0 to (- n^4 1)
      when (= (mod i n) 0)
        when (= (mod i n^2) 0)
          when (not (= i 0))
            do (format t "|~%")
          end
          and when (= (mod i nn^2) 0)
            do (loop repeat d
                   do (format t "-"))
            and do (format t "~%")
          end
        end
        and do (format t "|")
      else ; i % n != 0
        do (format t " ")
      end
      do (if (nth i board)
             (format t f (nth i board))
             (loop repeat a
```

```
          do (format t " "))))
   (format t "|~%")
   (loop repeat d
     do (format t "-"))
   (format t "~%")))

(defun node-to-sudoku-list (n node)
  "Converts an AIMA Problem node into a sudoku-list"
  (let ((vars (sort (CSP-state-assigned (node-state node))
                    #'<
                    :key #'(lambda (var)
                             (CSP-var-name var))))
        (n^4  (expt n 4)))
    (nconc (list n)
      (loop for i from 0 to (- n^4 1)
        collect (CSP-var-value (nth i vars))))))

(defun sudoku-solve (sudoku-list)
  (let* ((n (first sudoku-list))

         ;; Simple, no algorithmic embellishment.
         (simple-prob (make-sudoku-problem :sudoku-list sudoku-list))
         (simple-node (csp-backtracking-search simple-prob))
         (simple-solved (node-to-sudoku-list n simple-node))
         (simple-expanded (problem-num-expanded simple-prob))

         ;; Forward checking
         ;(fc-prob (make-sudoku-problem :sudoku-list sudoku-list))
         ;(fc-node (csp-forward-checking-search fc-prob))
         ;(fc-solved (node-to-sudoku-list n fc-node))
         ;(fc-expanded (problem-num-expanded fc-prob))

         ;; Minimum Remaining Value
         ;(mrv-prob (make-sudoku-problem :sudoku-list sudoku-list
                                        ;:variable-selector #'hmmmm))
         ;(mrv-node (csp-backtracking-search mrv-prob))
         ;(mrv-solved (node-to-sudoku-list n mrv-node))
         ;(mrv-expanded  (problem-num-expanded mrv-prob))

         ;; FC-MRV
         ;(fc-mrv-prob (make-sudoku-problem :sudoku-list sudoku-list
                                           ;:forward-checking? t
                                           ;:variable-selector #'hmmmm))
         ;(fc-mrv-node (csp-backtracking-search fc-mrv-prob))
         ;(fc-mrv-solved (node-to-sudoku-list n fc-mrv-node))
         ;(fc-mrv-expanded (problem-num-expanded fc-mrv-prob))
        )
```

```
    ;(format t "Sudoku Board~%")
    ;(print-sudoku sudoku-list)

    ;(format t "Simple Solution (Nodes Expanded: ~d)~%" simple-expanded)
    ;(print-sudoku simple-solved)

    ;(format t "Forward Checking Solution (Nodes Expanded: ~d)~%" fc-expanded)
    ;(print-sudoku fc-solved)

    ;(format t "Minimum Remaining Value Solution (Nodes Expanded: ~d)~%" mrv-
expanded)
    ;(print-sudoku mrv-solved)

    ;(format t "FC/MRV Solution (Nodes Expanded: ~d)~%" fc-mrv-expanded)
    ;(print-sudoku fc-mrv-solved)
))

(defun sudoku-translate (puzzle) ;;Gabe's translation function.
  (setq n 3)              ;;this could be useful later, for now use a dummy value.
  (setq output puzzle)    ;;generator output
  (setq input nil)
  (loop while output      ;;combine output lists into single list.
    do (setq input (append input (first output)))
    do (setq output (rest output))
  )
  ;(setq repeat (* n (* n (* n n)))) ;;n^4
  ;;(print repeat)  ;;debug
  ;(dotimes (counter repeat)     ;;iterate over list
    ;(if (eq (nth counter input) nil)
    ;(setf (nth counter input) '_)
    ;)
  ;)
  (nconc (list n) input) ;;this is the input for our solver.
)


(defun sudoku-make-puzzle ()
  (sudoku-translate (generate-puzzle)))

(defun sudoku-test (&optional (i 1))
  (loop repeat i
    do (setq puzzle (generate-puzzle))
          do (setq sudoku-list (sudoku-translate puzzle))
          (format t "Our time to solve:")
          do (time (sudoku-solve sudoku-list))
          (format t "Bowron's time to solve:")
          do (time (solve-puzzle puzzle))
          (format t "--------------------------")
```

```
   )
)


; This stuff is a test driver
;(setq a (list 2 1 2 3 4 5 6 7 8 9 1 nil 3 4 5 6 7))
(setq s2 '(2 1 nil 3 4 nil 4 nil nil nil 3 2 nil 2 nil 4 nil))
(setq s3 '(3
  1   nil 9     nil 6   8     3   nil nil
  nil 3   5     nil 7   1     nil nil nil
  8   nil 7     nil 2   nil   nil 5   nil

  nil nil 6     4   nil nil   nil nil 7
  3   nil nil   nil 1   nil   nil nil 8
  4   nil nil   nil nil 6     5   nil nil

  nil 5   nil   nil 4   nil   9   nil 6
  nil nil nil   6   5   nil   8   4   nil
  nil nil 4     8   3   nil   7   nil 5
))
(setq s4 '(3 nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil
nil nil nil nil nil nil nil nil nil))

;(print-sudoku a)
;(setq a (nconc '(3)
;  (loop for i from 1 to 81
;    collect (+ 1 (mod i 9)))))
;(print-sudoku a)
```

sudoku-problem.lisp

```
;;;; Sudoku Puzzle as a Constraint Satisfaction Problem

(defstructure (sudoku-problem (:include CSP-problem)
                              (:constructor create-sudoku-problem))
  "A structure to represnt the Sudoku Problem.  Uses aima-lisp defstructure
```

```
  to  define this to fit into that framework.  We just define simple things
  since we can easily fit Sudoku into a CSP-problem"
 n           ;; The order of this sudoku puzzle (usu. 3)
 sudoku-list) ;; The sudoku list.  Original format of original problem.


(defun make-sudoku-problem (&rest args &key sudoku-list
                                    (forward-checking? nil)
                                    (variable-selector #'first))
  "Calls create-sudoku-problem to make our CSP-problem.  Pass an initial-state
   to that function which represents Sudoku.  initial-state is a problem member.
   Allows args to be passed in so other members can be set.  initial-state is set
   by calling sudoku-initial-state which makes the actual call to make-CSP-state."
  (apply #'create-sudoku-problem
    :initial-state (sudoku-initial-state (first sudoku-list) (rest sudoku-list))
    :forward-checking? forward-checking?
    :variable-selector variable-selector
    args))

(defun sudoku-initial-state (n sudoku-list)
  "Calls make-CSP-state and defines the initial state of the sudoku board just
   before starting to solve it.  As is typical, delegates off to other functions
   to figure out what is and is not yet assigned, and also to name the variables
   for CSP-var."
  (make-CSP-state
    :unassigned (sudoku-unassigned n sudoku-list)
    :assigned   (sudoku-assigned n sudoku-list)
    :constraint-fn #'(lambda (var1 val1 var2 val2)
                       (sudoku-constraint-fn n var1 val1 var2 val2))))

(defun sudoku-unassigned (n sudoku-list)
  "Collects and returns each as-of-yet unassigned suodku value. It loops through
   each item in the list and when a nil value is  found it makes a new CSP-var
   whose name is the current iteration and whose domain (possible values) is
   [1-n]."
  (let ((n^2 (expt n 2))
        (n^4 (expt n 4)))
    (loop for i from 0 to (- n^4 1)
      unless (nth i sudoku-list)
        collect (make-CSP-var :name i :domain (iota n^2 1)))))

(defun sudoku-assigned (n sudoku-list)
  "Collects and returns all previously assigned values, which are what make the
   Sudoku puzzle interesting to solve.  Loops through each index in the
   sudoku-list and if any are non-nil it makes a new CSP-var and assigns it the
   name of the current iteration and sets the value to that in the sudoku-list."
  (let ((n^4 (expt n 4)))
    (loop for i from 0 to (- n^4 1)
```

```
      when (nth i sudoku-list)
       collect (make-CSP-var :name i :value (nth i sudoku-list)))))

(defun sudoku-constraint-fn (n var1 val1 var2 val2)
  "Returns true if var1/val1 and var2/val2 are compatible.  This is true only if
   1. They're in differnt rows or in the same row with different values
   2. They're in different columns or in the same column with different values
   3. They're in different squares or in the same square with different valueos
   The check is done by delegating to other functions, of course."
   (and (sudoku-check-row    n var1 val1 var2 val2)
        (sudoku-check-column n var1 val1 var2 val2)
        (sudoku-check-square n var1 val1 var2 val2)))

(defun sudoku-check-row (n var1 val1 var2 val2)
  "Check for the same row and same value.  If this isn't the case, then the
   variables are row-wise compatible."
  (not (and (= val1 val2)
            (sudoku-same-row n var1 var2))))

(defun sudoku-check-column (n var1 val1 var2 val2)
  "Check for the same column and same value.  If this isn't the case, then
   the variables are column-wise compatible."
  (not (and (= val1 val2)
            (sudoku-same-column n var1 var2))))

(defun sudoku-check-square (n var1 val1 var2 val2)
  "Check for the same square and same value.  If this isn't the case, then
   the variables are row-wise compatible."
  (not (and (= val1 val2)
            (sudoku-same-square n var1 var2))))

(defun sudoku-same-row (n var1 var2)
  "Checks if two variables are in the same row by checking if the first
   element in each's row is the same.  Calculating the 0th element in
   the row is delegated"
   (= (sudoku-row-0 n var1)
      (sudoku-row-0 n var2)))

(defun sudoku-same-column (n var1 var2)
  "Checks if two variables are in the same column by checking if the first
   element in each's column is the same.  Calculating the 0th element in
   the column is delegated"
   (= (sudoku-column-0 n var1)
      (sudoku-column-0 n var2)))

(defun sudoku-same-square (n var1 var2)
  "Checks if two variables are in the same square by checking if the first
   element in each's square is the same.  Calculating the 0th element in
```

```
   the square is delegated"
   (= (sudoku-square-0 n var1)
      (sudoku-square-0 n var2)))

(defun sudoku-row-0 (n var)
  "Calculates the 0th index in var's row.  Assumes row-based indexing."
  (let ((n^2 (expt n 2)))
    (- var (mod var n^2))))

(defun sudoku-column-0 (n var)
  "Calculates the 0th index in var's column.  Assumes row-based indexing."
  (let ((n^2 (expt n 2)))
    (mod var n^2)))

(defun sudoku-square-0 (n var)
  "Calculates the 0th index in the var's square.  This index is that first
   one in row-based indexing for the n^2 sized square containing var.  Of
   course, it assumes row-based indexing.  This calculation is fun!"
  (let ((n^2 (expt n 2)))
    (+ (- (mod var n^2) (mod var n)) ; Computes column
       (- (- var (mod var n^2)) ; Computes 1st index in row
          (* (mod (truncate var n^2) n) n^2))))) ; Computes 1st row in square
```

sudoku-generator.lisp

```
;;; The following code is borrowed from Christopher Bowron.  We use it
;;; solely for the purpose of generating new puzzles.  Our solver borrows
;;; no code and does not knowingly borrow any  concepts or ideas from this
;;; code (it was developed completely independantly).
;;;
;;; Available from:
;;; http://www.setbb.com/phpbb/viewtopic.php?t=195&highlight=lisp&mforum=sudoku



;;; SUDOKU solver and generator
;;; Christopher "Puzzle Man" Bowron <puzzleman@bowron.us>

(proclaim '(optimize (speed 3)))

(defun solve-puzzle (puzzle &key debug randomize reverse)
  (dotimes (row 9)
    (dotimes (col 9)
      (unless (get-number puzzle row col)
   (let ((possibilities
          (get-possibilities puzzle row col)))
     (when debug
       (format t "~%~%row = ~A, col = ~A~%" row col)
       (format t "possibilities:: ~A~%" possibilities)
```

```
      (format t "~A~%" puzzle))

    (dolist (poss (cond
          (reverse (nreverse possibilities))
          (randomize (randomize-list possibilities))
          (t possibilities)))
      (let ((working-copy (copy-tree puzzle)))
        (setf (nth col (nth row working-copy)) poss)
        (let ((r (solve-puzzle working-copy
              :debug debug
              :randomize randomize
              :reverse reverse)))
     (when r
       (return-from solve-puzzle r))))))
   (return-from solve-puzzle nil))))
  puzzle)

;;; optimize by solving for rows/cols/squares with the fewest
;;; possibilties first
(defun solve-puzzle-fast (puzzle &key debug randomize reverse)
  (let ((ordering (get-ordering puzzle)))
    (dolist (item ordering)
      (let ((row (car (car item)))
       (col (cdr (car item))))
   (unless (get-number puzzle row col)
     (let ((possibilities (get-possibilities puzzle row col)))
       (when debug
         (format t "~%~%row = ~A, col = ~A~%" row col)
         (format t "possibilities:: ~A~%" possibilities)
         (format t "~A~%" puzzle))
       (dolist (poss (cond
          (reverse (nreverse possibilities))
          (randomize (randomize-list possibilities))
          (t possibilities)))
         (let ((working-copy (copy-tree puzzle)))
     (setf (nth col (nth row working-copy)) poss)
     (let ((r (solve-puzzle-fast working-copy
               :debug debug
               :randomize randomize
               :reverse reverse)))
       (when r
         (return-from solve-puzzle-fast r))))))
    (return-from solve-puzzle-fast nil)))))
  puzzle)

;;; generate a solution by "solving" a blank grid using
;;; random permutation
;;; then randomly remove items that allow for the puzzle to still
```

```
;;; be solved to the original solution
;;; returns the puzzle and the solution
(defun generate-puzzle (&key debug min)
  (let ((solution (solve-puzzle-fast *blank* :randomize t)))
    (let ((puzzle (copy-tree solution)))
      (dolist (index (randomize-list (number-list 0 80)))
   (when (or (not min) (< min (count-numbers puzzle)))
     (let ((working-copy (copy-tree puzzle))
      (row (floor (/ index 9)))
      (col (mod index 9)))
       (setf (nth col (nth row working-copy)) nil)
       (when (and
         (equal solution (solve-puzzle-fast working-copy))
         (equal solution
           (solve-puzzle-fast working-copy :reverse t)))
         (when debug (print working-copy))
         (setq puzzle working-copy)))))
      (values puzzle solution))))


;;; returns a list of numbers from min to max inclusive
(defun number-list (min max)
  (let ((numbers nil)
   (n (+ 1 (- max min))))
    (dotimes (i n)
      (setq numbers (cons (+ min i) numbers)))
    (nreverse numbers)))

;;; returns a list in random order
(defun randomize-list (some-list)
  (do ((randomized-list nil))
      ((null some-list) randomized-list)
    (let ((item-number (random (length some-list))))
      (push (elt some-list item-number)
            randomized-list)
      (setf some-list
            (remove (elt some-list item-number) some-list)))))

(defvar *all-possible-numbers-forward*
  (list 1 2 3 4 5 6 7 8 9))

(defvar *all-possible-numbers-backward*
  (list 9 8 7 6 5 4 3 2 1))

(defun possible-numbers (list)
  (set-difference *all-possible-numbers-forward* list))

(defun get-column (puzzle column)
```

```
  (map 'list #'(lambda (x) (nth column x)) puzzle))

(defun get-row (puzzle row)
  (nth row puzzle))

(defun get-number (puzzle r c)
  (nth c (nth r puzzle)))

;; gets the contents of the 3x3 square in which this (row,col) resides
(defun get-ninth (puzzle row col)
  (let ((r (floor (/ row 3)))
   (c (floor (/ col 3))))
    (let ((ninth nil))
      (dotimes (x 3)
   (dotimes (y 3)
     (push
      (get-number puzzle (+ (* r 3) y) (+ (* c 3) x))
      ninth)))
      (nreverse ninth))))

;;; Count how many missing numbers there are in the list
(defun count-missing (list)
  (count-if #'null list))

;;; count the numbers in a row or puzzle
(defun count-numbers (list)
  (cond ((numberp list) 1)
   ((null list) 0)
   (t (reduce #'+ (map 'list #'count-numbers list)))))

(defun get-possibilities (puzzle row col)
  (possible-numbers
   (union
    (union (get-row puzzle row)
      (get-column puzzle col))
    (get-ninth puzzle row col))))

(defun get-ordering (puzzle)
  (let ((ordering nil))
    (dotimes (row 9)
      (dotimes (col 9)
   (unless (get-number puzzle row col)
     (let ((value
       (length (get-possibilities puzzle row col))))
       (push (cons (cons row col) value) ordering)))))
    (sort ordering #'< :key #'cdr)))

(defvar
```

blank
```
'((nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)
  (nil nil nil nil nil nil nil nil nil)))
```