Musical Constraint Satisfication Programming Project

Release 1.0

Iain C.T. Duncan

Aug 18, 2024

CONTENTS:

The objective of this project is to implement an algorithmic musical line builder as a constraint satisfaction problem using a custom-buit domain-specific solver in the Scheme programming language such that it can be run in the Scheme for Max computer music environment. The solver will allow the user (i.e., the programmer using the solver in a Scheme for Max program) to specify constraints by registering predicate functions that take a standardized set of arguments and that work over symbolic musical representation. It is assumed that the user has familiarity with the Scheme language, but not necessarily with the details of CSP programming, and that the user is comfortable with basic music theory. The project will provide a Scheme message-passing-style object that acts as a CSP solver, along with a library of symbolic functions allowing the user to make new constraint predicates relatively easily, working with familiar musical symbols.

The test application for the project is a program that builds walking idiomatic jazz walking bass lines, given a key, a starting note, and a symbolic chord progression such as IImin7 V7 IMaj7. The test application will return the realized bass line as set of symbolic note names (e.g., D4 F4 A4).

CHAPTER

ONE

SPECIFIC GOALS

Specific goals of the implementation are:

- Constraints may be either local (run over one variable) or global. A solution will satisfy all local and global constraints.
- Constraints may be easily created in high-level Scheme using an included set of musical primitives.
- Multiple runs of the program will create different bass-lines, i.e., the CSP solver is not looking for the first or best solution, but rather a stochastically chosen acceptable solution.
- The solver will find a solution if one exists, this at the possible expense of execution speed.
- If the constraints given result in an unsolvable problem, the solver will determine this and return false.
- Solutions will be rendered bar-by-bar, such that it would be possible for the solver to be used to play algorithmically-generated lines in real time and for infinite time, so long as the solver can generate a bar's notes in less time than is used to play a bar.
- The solver will accept constraints using symbols of all note values (i.e., either C# or Db may be used) but will not necessarily return values translated into symbols appropriate to the key (i.e., it may use either Db or C# for note values, regardless of the context)

CHAPTER

TWO

PROGRAM STRUCTURE

The solver is implemented as a Scheme function-as-an-object that receives symbolic messages and that stores local values in private scope. This is implemented by using a builder function that creates a local let-block, and returns a lambda function with access to the values in the let. The returned function acts as the object, and methods can be called by passing symbolic messages, which are dispatched to similarly named functions with the local scope. By implementing the solver as a self contained object, the user of the solver must thus only load the code defining the csp solver builder function, and any musical library functions used to build constraints.

```
; example of the the message passing object structure used
(define (make-obj)
  ; object state is stored in a hash-table with keyword keys
  (let ((_ (hash-table
             :var-1 #t
             :var-2 #f)))

    ; a method to return a state var
    (define (get-var var-name)
      (_ var-name))

    ; a method to update a state var
    (define (set-var! var-name val)
      (set! (_ var-name) val))

    ; dispatch method is a lambda returned by make-obj
    (lambda (msg . args)
      (apply (eval msg) args))))

; make an object
(define my-obj (make-obj))
; set var-2 to value of var-1
(my-obj 'set-var :var-2 (my-obj 'get 'var-1))
```

The solver is set up via an explicit initialization method, allowing the same solver to be reused for subsequent bars by only setting the values that need to change for each bar by reinitializing the solver object.

Constraints are added via a registration function that accepts a predicate function, the variable over which it runs, and a symbolic name. Global constraints are similarly added, but are assumed to have access to all notes.

The solver's **solve** method runs the solver, returning a list of symbolic notes.

CHAPTER

THREE

HIGH-LEVEL IMPLEMENTATION OF THE SOLVER

The solver contains two kinds of domain variables, note variables and context variables.

Notes are stored in a list and are assumed to be empty prior to solving, though are not necessarily so. The domain of note variables is separately maintained in local state, and is set up during initialization to contain MIDI note-names across a pre-determined range (e.g., C0 to B5). The number of note variables is determined by an argument to the object builder, and the problem is considered to be solved when values have been successfully assigned to all note slots, satisfying all local and global constraints. The use of MIDI note symbols rather than note numbers was chosen to facilitate interactive debugging and exploratory coding.

Context variables are symbolic state variables for the solver and are assumed to be pre-assigned prior to the solve method. While they can be changed in between calls to the solve method, the solver does not assign context variables as part of finding a solution.

As the solver solves bar-by-bar, it must be aware of the bar to which it is leading. In my test application, for example, to build an idiomatic jazz bass line with four notes per bar, note 5 of the solution will always be the tonic of the subsequent bar, and note 4 will always be either a chromatic neighbour tone of this target. Thus, we require the solver to have context for the current bar and the subsequent bar.

The context variables used are the following

tonic
: The home key for the progression, as a pitch-class symbol ('C, 'Db, etc.)

tonality
: The tonality of the home key, as a symbol of 'major, 'minor, or 'blues.

root
: The root of the chord for the line, as a Roman numeral symbol ('I, 'IV, etc.)

quality
: The chord quality for the line, as a chord symbol ('Maj7, 'Min7, etc)

target
: The chord root for the next bar, as a Roman numeral

target-q
: The chord quality of the target chord, as a chord symbol

Initialization prior to solving for bar thus consists of passing a hash-table of context variables to the **method**, which will copy values into the local context state, and initialize the note variables to empty.

```
; a simplified example of building and running a solver
; the first argument to the solver is how many notes should be returned per bar
(define csp (make-csp 5))

; add a local and global constraint
```

(continues on next page)

(continued from previous page)

```
(csp 'add-constraint chord-root? '(0)  'n0-root)
(csp 'add-global-constraint (intv-under? 'maj-3))

; initialize the csp with context variables
(csp 'init csp (hash-table
                'tonic 'C
                'tonality 'major
                'root 'I
                'quality 'Maj-7
                'target 'IV
                ))

 ; solve
 (define notes-out (csp 'solve))
```

CHAPTER

FOUR

DOMAIN MODEL FUNCTIONS

The project includes a file of domain specific helper functions necessary for building constraints with symbolic notes and rules. These allow us to work with:

note symbols
MIDI note symbols that indicate pitch and octave such as C0, Db0, etc.

pitch class symbols
Symbols for the pitch alone: C, Db, D, etc.

note numbers
MIDI note numbers from 0 to 127

roman numerals
Numerals to indicate chord function, I, II, III, etc.

chord quality symbols
Symbols for a type of chord: Maj7

interval symbols
Symbolic intervals, used to define qualities (min-3, prf-5, etc.)

The helper libary provides translation functions between these symbolic and numeric representations. A current limitation is that pitch classes are assumed to use flats only, but the structure of the library is such that this can be easily extended to include all symbolic note names.

Where possible, translation functions are implemented to use hash-tables as a form of caching, this to avoid unecessary calculation during constraint execution, which may run many times for a solve pass. Some of the translation functions are simply prepopulated in code, while others are populated dynamically when the helper library is imported.

The nomenclature used in helper function names is:

int
refers to an integer offset of a pitch (e.g., C == 0, D ==2)

note-num
refers to MIDI note numbers

pitch
refers to symbolic pitch class (C, D, . . .)

note
refers to a symbolic note of pitch class and octave

oct
refers to the integer octave as used in MIDI note names

interval
refers to a symbolic interval (min-3, prf-4, . . .)

rnum
refers to a Roman numeral symbol (I, II, . . .)

chord
refers to a chord quality symbol (Maj7, Min7, etc.)

Helper functions provided by the library include the below, where the components are the above, and -> indicates a translation. Hyphenated components and plurals indicate a list input or output.

- pitch->int
- int->pitch
- interval->semitones
- semitones->interval
- note->pitch
- note->oct
- note->note-num
- semitones-between
- interval-between
- chord-intervals
- chord->semitones
- rnum->int
- root-pitch
- root-chord->note-nums
- root-chord->pitches

CHAPTER

FIVE

LOCAL CONSTRAINTS

Constraints are implemented as boolean predicate functions with standard arguments that give the predicate access to the csp solver's state (and thus context). Local constaints are passed a reference to the csp object, an integer corresponding the note variable index, and the value being tested as a MIDI note symbols.

```
; a local constraint example
; checks if the value under test is the same pitch-class as the context
; variable for the tonic of key
(define (is-tonic? csp var val)
  (let* ((tonic (csp 'get-var 'tonic))
         ; the note->pitch helper returns a pitch-class symbol from a note symbol
         (res   (eq? tonic (note->pitch val))))
    res))

; registering the constraint, using the registration name 'is-tonic
; it is registered as running over the tonic context var and note 0
(csp 'add-constraint is-tonic? '(tonic 0) 'is-tonic)
```

As constraints are registered using a reference to a function, we can also implement higher-level constraint builders.

```
(define (is-chord-factor? csp var val factor)
"return true if note pitch class is the given factor of the chord"
  (let* ((root  (csp 'get-var 'root))
         (chord-q (csp 'get-var 'quality))
         (chord-pitches (root-chord->pitches root chord-q))
         (val-pitch (note->pitch val))
         (res (enh-eq? val-pitch (chord-pitches factor))))
    res))

; register a function that checks if the note is the third of the chord
(csp 'add-constraint
  (lambda (csp var val) (is-chord-factor? csp var val 1))
  '(1) 'is-third)
```

During the solve process, the solver users the **get-applicable-constraints** method to find all local constraints that can be run on a given assignment pass, running them prior to the assignment of value to a note variable.

CHAPTER

SIX

GLOBAL CONSTRAINTS

Global constraints are implemented as predicates that have access to the entire list of note variables, and are run post-assignment, with assignement rolled-back on failure. They do not have any notion of the current variable being assigned, and must thus be able to handle the case of incomplete note lists by returning true to indicate a pass. A future improvement would be to enable global constraint registration to indicate which variables must be filled prior to running so that the constraint code itself does not need to handle this.

For example, in the constraint to check if the last note of the bar is a chromatic neighbour follwed by the target root, the constraint passes if we are missing any of the necessary domain variables, and checks the relationship between the fourth and fifth note if we have them.

```
(define (target-from-cn? csp notes)
  (if (or (not (csp 'get-var 'target)) (not (notes 3)) (not (notes 4)))
    ; we don't have enough to run the constraint, pass
    #t
    ; if we do, check that it is a chromatic step between notes 4 and 5
    (let* ((n-note (notes 3))
           (t-note (notes 4))
           (n-num (note->note-num n-note))
           (t-num (note->note-num t-note))
           (res (= 1 (abs (- t-num n-num)))))
      res)))
```

With the current implementation, one could thus implement a brute-force solver using only global constraints.

The library of constraints includes an imperative implementation of all-diff, as well as a more specific implementation that accepts a list of notes which must be different. This was necessary for the test program, for example, as the chromatic neighbour of a target root may or may not be already in the chord. We thus need a way to specify, “all notes should be different, unless the duplicated note is the chromatic neighbour as note 4”.

```
; fail on any already assigned notes that are duplicates
(define (all-diff? csp notes)
  (let ((len   (length notes))
        (pass  #t))
    (do ((i 0 (+ 1 i))) ((= i len))
      (do ((j 0 (+ 1 j))) ((= j len))
        (if (and (eq? (notes i) (notes j)) (not (= i j)) (not (false? (notes i))))
          (set! pass #f))))
    pass))

; diff is like all-diff but takes list of note vars
(define (diff? note-var-list)
  (lambda (csp notes)
```

(continues on next page)

(continued from previous page)

```
  (let ((len   (length notes))
        (pass  #t))
    (do ((i 0 (+ 1 i))) ((= i len))
      (do ((j 0 (+ 1 j))) ((= j len))
        (if (and
              (in? i note-var-list)
              (in? j note-var-list)
              (eq? (notes i) (notes j))
              (not (= i j))
              (not (false? (notes i))))
          (begin
            (set! pass #f)))))
    pass)))
```

CHAPTER

SEVEN

CONSTRAINTS FOR THE TEST PROBLEM

Given the above constraints, we have what we need to specify an arpeggiated bass line that always leads chromatically into the next chord. Our rules are:

- The first note must be the root of the chord
- The second and third notes must in the chord
- None of the first three notes may be the same
- Adjacent notes are under a major 3rd apart
- The final note is the root of the target chord
- The penultimate note is a chromatic neighbour of the target

In the test program, this is implemented as a function to add these constraints.

```
(define (add-constraints csp)
  (csp 'add-constraint chord-root? '(0)  'n0-root)
  (csp 'add-constraint in-chord? '(1) 'in-chord-1)
  (csp 'add-constraint in-chord? '(2) 'in-chord-2)
  (csp 'add-constraint target-root? '(4) 'target-root)
  (csp 'add-global-constraint (diff? '(0 1 2)))
  (csp 'add-global-constraint (intv-under? 'maj-3))
  (csp 'add-global-constraint target-from-cn?)
)
```

SOLVER IMPLEMENTATION

The solving algorithm is a recursive tree walk, where the depth corresponds to the index within the list of notes. This is implemented in the csp object's **solve** method. The solve method contains an inner recursive function, **recursive-search**, which is passed a boolean result and a depth, but also as access to the note variable assignment list as this is in scope of the function-object.

```
; the main search method, fills and returns the notes vars on success
; can take optional positional arg of hash-table of preassignments
(define (solve . args)
  (if (> (length args) 0)
    (pre-assign (args 0)))
  (post "csp::(solve) ctx:" (_ :cxt-assignments) "notes:" (_ :note-assignments))

  ; inner recursive function
  (define (recursive-search result depth)
  ...
```

The first check is whether all notes have been assigned, and if this is the case, a **#true** value is returned up the stack.

```
; inner recursive function
(define (recursive-search result depth)
  ;if assignment complete, we are done return assignment
  ;(post "")
  (post "csp::solve::(search) depth:" depth "notes:" (_ :note-assignments))
  (cond
    ; case done, vector of 4 notes filled, return success
    ; executes when we get to the bottom of recuring down
    ((all-notes-assigned?)
       ;(post "  - all note assigned, returning #t up stack")
       #t)
    ; else we still have notes to fill
    (else
      ...
```

If it is not, the next variable to solve is selected using the **select-var** method, and the solver then iterates through all possible domain values for this variable, using a named-let block as a loop, with a recursively reduced list of domain values passed into each iteraction (a.k.a. "a list eater function", though implemented as a named-let rather than a lambda function).

If all domain values have been exhausted (the list is null), we return **#f** up the stack.

```
; else we still have notes to fill
(else
  ; get the next far to fill
  (let ((var (select-var)))
    ; iterate through domain values for i
    (let* domain-val-loop ((vals (get-domain-values var)))
      ;(post "domain-val-loop: domain-vals:" vals)
      (if (null? vals)
        ; case ran out of domain vals, return failure back up
        (begin
          ;(post " - out of possible domain values, return #f up stack")
          #f)
        ; else, try assigning the val, check constraints we can run so far
```

If there is a domain value remaining to be tested, it is checked against constraints applicable to the current value, and assigned on pass. On failure, iteration continues. Assuming the local constraints pass, the note variable is assigned and we proceed to global constraints.

If global constraints pass, the note assignment is kept, and we proceed down one more depth by recursively calling **recursive-search**, passing in the new depth and a value of **#true** for the result.

In the case that assignment and local constraints succeed but global constraints fail, the assigned note is unassigned, and the domain value loop continues.

```
; get the next far to fill
(let ((var (select-var)))
  ; iterate through domain values for i
  (let* domain-val-loop ((vals (get-domain-values var)))
    ;(post "domain-val-loop: domain-vals:" vals)
    (if (null? vals)
      ; case ran out of domain vals, return failure back up
      (begin
        ;(post " - out of possible domain values, return #f up stack")
        #f)
      ; else, try assigning the val, check constraints we can run so far
      (let ((passed (assign-if-valid var (first vals))))
        (cond
          ; didn't pass precheck, on to next possible domain value
          ((not passed)
            (domain-val-loop (cdr vals)))
          ; passed precheck, failed globals: unset and continue domain val loop
          ((not (check-global-constraints))
            (set! (_ :note-assignments var) #f)
            (domain-val-loop (cdr vals)))
          ; passed everything, found value, recurse onwards
          ; if recursing fails, unset var and continue looking
          ((not (recursive-search passed (+ 1 depth)))
            (set! (_ :note-assignments var) #f)
            (domain-val-loop (cdr vals)))
          (else
            ;(post " - found passing domain val:" passed "for depth" depth "returning #t")
            #t))))))))
```

In the case that a solution is found, the enclosing method, **solve** returns the note assignments in addition to the side effect of having the local note assignment list populated, this to simplify using the csp object. In the event of failure, it returns false.

The complete implementation of the backtracking search is shown below:

```scheme
; the main search method, fills and returns the notes vars on success
; can take optional positional arg of hash-table of preassignments
(define (solve . args)
  (if (> (length args) 0)
    (pre-assign (args 0)))
  (post "SOLVING")
  (post "csp::(solve) ctx:" (_ :cxt-assignments) "notes:" (_ :note-assignments))

  ; this works with the object vals, not so sure if that is better than having it separate
  (define (recursive-search result depth)
    ;if assignment complete, we are done return assignment
    ;(post "")
    (post "csp::solve::(search) depth:" depth "notes:" (_ :note-assignments))
    (cond
      ; case done, vector of 4 notes filled, return success
      ; executes when we get to the bottom of recuring down
      ((all-notes-assigned?)
         ;(post "  - all note assigned, returning #t up stack")
         #t)
      ; else we still have notes to fill
      (else
        ; get the next far to fill
        (let ((var (select-var)))
          ; iterate through domain values for i
          (let* domain-val-loop ((vals (get-domain-values var)))
            ;(post "domain-val-loop: domain-vals:" vals)
            ; test first value, if good, use it and recurse
            (if (null? vals)
              ; case ran out of domain vals, return failure back up
              (begin
                ;(post " - out of possible domain values, return #f up stack")
                #f)
              ; else, try assigning the val, check constraints we can run so far
              (let ((passed (assign-if-valid var (first vals))))
                (cond
                  ; didn't pass precheck, on to next possible domain value
                  ((not passed)
                    (domain-val-loop (cdr vals)))
                  ; passed precheck, failed globals: unset and continue domain val loop
                  ((not (check-global-constraints))
                    (set! (_ :note-assignments var) #f)
                    (domain-val-loop (cdr vals)))
                  ; passed everything, found value, recurse onwards
                  ; if recursing fails, unset var and continue looking
                  ((not (recursive-search passed (+ 1 depth)))
                    (set! (_ :note-assignments var) #f)
                    (domain-val-loop (cdr vals)))
                  (else
                    ;(post " - found passing domain val:" passed "for depth" depth "returning #t")
                    #t)))))))))
```

CHAPTER

NINE

LIMITATIONS

9.1 Propagation

The current implementation is essentially a brute-force solver and does not (yet) include any forward-propagation of constraints. When constraints fail at a given depth, the solver will next try alternative domain values, and if all fail, will backtrack up a depth. There is no implementation of pruning domain values and undoing a prune when backtracking up a level. I did develop code to create a persistent tree representation of all paths that have been explored, creating a recursively generated tree with double-linked nodes such that paths from a node can be traversed up or down the tree. This structure could be used to store domain changes from visting node, which could be used to undo domain pruning when backtracking. However, time constraints (ha!) prevented me from incorporating this into the solver.

```
; make a tree object that can return values recursing up the tree from a leaf
(define (make-tree)
  (let ((depth 0)
        (root (hash-table :children '() :depth 0 :value #f))
       )
    (define (add-node parent value)
      (let* ((parent-node (if parent parent root))
             (node-depth (+ 1 (parent-node :depth)))
             (node (hash-table
                     :children '()
                     :depth node-depth
                     :parent parent-node
                     :value value)))
        (set! (parent-node :children) (cons node (parent-node :children)))
        (if (> node-depth depth) (set! depth node-depth))
        node))

    (define (print)
      (post "tree depth:" depth "structure" root))

    (define (values-from-node node)
      ;(post "values-from-node, starting at depth:" (node :depth))
      ; recurse up the tree from a node to get list of values
      (let* rec-loop ((n node)
                      (vals '()))
        (cond
          ((eq? #f (n :parent))
            vals)
          (else
            (rec-loop (n :parent) (cons (n :value) vals))))))
```

(continues on next page)

(continued from previous page)

```
(lambda (msg . args)
```

9.2 Variable Selection

The implementation of choosing the next variable to be solved is naive; it simply iterates through the note list in order. Allowing solving order to be indicated or optimized would reduce paths explored. For example, in bass line implementation, one could work either forwards or backwards from the target.

9.3 Value Selection

Similar to variable selection, value selection at present simply iterates through the domain values. Work is underway to improve this with random urn implementation.

CHAPTER

TEN

FUTURE WORK

There are a number of areas of future work I intend to explore.

10.1 Constraint Propagation

As discussed above, propagation of constraints is the the first major deficiency. Constraint propagation will significantly speed up execution, though execution time does not seem to be much of an issue even with the current implementation. This could be an issue solving longer lines.

10.2 Advanced Pruning

Given the domain-specific constraint implementation, it would be reasonably straight-forward to add a special type of constraint intended to prune variable domains prior to solving, which could speed up execution. For example, a user might specify that all notes up the targeting final note should be diatonic to the tonic key. Such a constraint could thus prune the domain for notes 1 to 3 to be diatonic only, cutting the number of branches to be explored down significantly.

10.3 Assignment

Notes are assigned one at a time, but this is not an ideal heuristic for simulating how musicians think. For example, when playing a bass line, a musicain is likely to think “I can walk or arpeggiate”, determing which based on available range and what they have recently done, and evaluating whether several candidates will work at one time. The solver could draw from a bank of patterns to try various compound selections first rather than brute force traversal through all paths of the state space.

10.4 Constraint Macro Framework

While implementing the constraints as boolean predicates is straightforward, it does require the developer to be somewhat familiar with the implementation details of the csp solver. A higher-level framework built on Lisp macros could be implemented to reduce the amount of code necessary and to reduce how much must be understood by the authors of the constraints.

10.5 Solver Instantiation

Creating, initializing, and setting up a solver with constraints and context variables is currently a fairly verbose affair. This is an area where Lisp macros could again allow a much more succint and readable API for users of the solver.

10.6 Stochastic Heuristics

When choosing a next candidate domain value, the current implementation either takes the next one, or a random value. The addition of preference constraints to help choose candidates in more musically meaningful ways would be a fruitful area to explore. This could potentially allow one to implement preference rules such as *“if the last three notes have ascended, increase the probability of the next note descending”*.

CHAPTER

ELEVEN

CONCLUSION

As a first attempt at solving musical line building with a constraint solver, I feel the project has been a success. Execution is fast enough to allow generating a bass line in perpetuity. While still somewhat verbose, specifying rules in a similar way to how one thinks of musical constraints works well. One can read, for example, the set of constraint registrations to make an arpeggiated bass line and understand right away what the outcome is intended to be.

Breaking up the constraints into (solved) notes and (pre-assigned) context variables is also successful, and makes domain specific problems much easier to reason about and to design constraints for then would be the case with all numerical variables, with no destinction between context and outcome. It is also more reflective of how improvisers and composers think of the problems: the chord progression and key are a given, the notes to realize it are improvised.

The project runs well in Scheme for Max and, I believe, has the potential to be a valuable contribution to the available offerings for computer assisted composition in the Max environment, such as the Bach project.