

# Organon, A Symbolic Constraint Framework And Solver

Isaac Evans and Joseph Lynch

May 16, 2013

# Contents

|          |                                       |           |
|----------|---------------------------------------|-----------|
| <b>1</b> | <b>Introduction</b>                   | <b>3</b>  |
| <b>2</b> | <b>Problem</b>                        | <b>3</b>  |
| <b>3</b> | <b>Organon Library Framework</b>      | <b>3</b>  |
| 3.1      | Forms . . . . .                       | 3         |
| 3.2      | Constraints . . . . .                 | 4         |
| 3.3      | Solvers . . . . .                     | 5         |
| 3.3.1    | Solver 1: Exhaustive . . . . .        | 6         |
| 3.3.2    | Solver 2: Annealing . . . . .         | 6         |
| <b>4</b> | <b>Example Uses of Organon</b>        | <b>7</b>  |
| 4.1      | Ladder . . . . .                      | 7         |
| 4.2      | Big Bang . . . . .                    | 7         |
| 4.3      | Laffer Curve . . . . .                | 8         |
| 4.4      | Graph Coloring . . . . .              | 9         |
| <b>5</b> | <b>Future Work &amp; Conclusions:</b> | <b>10</b> |
| <b>A</b> | <b>Appendix A - Library</b>           | <b>11</b> |
| A.1      | forms.scm . . . . .                   | 11        |
| A.2      | constraints.scm . . . . .             | 13        |
| A.3      | solver.scm . . . . .                  | 15        |
| A.4      | util.scm . . . . .                    | 19        |
| A.5      | opengl-stream.scm . . . . .           | 21        |
| <b>B</b> | <b>Appendix B - Demos</b>             | <b>24</b> |
| B.1      | ladder.scm . . . . .                  | 24        |
| B.2      | big-bang.scm . . . . .                | 26        |
| B.3      | laffer.scm . . . . .                  | 28        |
| B.4      | node-coloring.scm . . . . .           | 29        |
| B.5      | harder-coloring.scm . . . . .         | 30        |

# 1 Introduction

Organon is an open source system for expressing and solving complex symbolic constraints between generic entities. It has three main components:

1. **Forms:** Abstract representations of the entities to be constrained
2. **Constraints:** Functions that symbolically express requirements on the relationships between forms as well as provide information for how a solver can improve upon the constraint's satisfaction.
3. **Solvers :** Functions which inspect instantiations of forms and manipulate them in an attempt to satisfy a set of objective constraints

Organon is designed to be very generic, which is why it ships with four demonstrations in entirely different domains. Organon should not constrain the programmers ability to phrase their constraints, it should only give them a framework to express their intentions.

## 2 Problem

The inspiration for Organon was a challenge faced during development work on MIT's entry to the third DARPA Robotics Challenge (DRC). The action authoring system for the DRC needed a way to express constraints between the robot and real-world objects. Ideally, these constraints would be specified symbolically, rather than as fully specified 3D pose offsets. These symbolic constraints could express high-level intents such as "these hands grasp the ladder rung at a point not so close to the edge that it cannot be reached, but not so close to the center that the hands are more than a shoulder-width apart." In our system, once such a constraint function is expressed, it is applicable to any object (for example, stairs instead of ladder rung) which has the properties required by the constraint function.

Our completed system accommodates this concept of symbolic constraints and is generic enough to allow a host of other constraint expressions that are completely unrelated to 3D concepts.

## 3 Organon Library Framework

The main Organon library provides functions needed to create, manipulate, and constrain forms, as well as two basic implementations of solvers that can iteratively improve form bindings such that constraints become more satisfied.

### 3.1 Forms

Forms represent a symbolic and straightforward way to represent the world. Forms have properties, and possibly they have values associated with those properties. Internally these are represented by **eq-property** lists so that we can store completely arbitrary properties. To make working with forms easier, Organon provides a basic typing system that allows the programmer to alias "types" to a set of properties. For example, one can declare the form type `'sphere` to have the properties of `'center` and `'radius` with the following code:

```
;; Declare type 'sphere to have a 'center and a 'radius
(declare-form-type 'sphere (list 'center 'radius))
```

It is important to note that this declaration does not limit the programmer to particular data types, `'center` can be a vector just as easily as it can be an integer. Once a type is declared, it is very easy to instantiate forms with all of the properties present in the form type declaration:

```
;; Declare a form 'ball that has all types in the type 'sphere (so 'center
and 'radius)
(declare-form 'ball 'sphere)
```

Additionally, Organon provides basic type inheritance, which allows the programmer to specify that a particular form has all the properties associated with another form. This is useful because it means that people can reuse common property definitions (such as those of a `3d-object`). To inform the system about a type inheritance, one can use the `declare-type-inherits` function to tell the system that one form type should have all properties of another type. For example, `sphere`'s are `3d-objects` made up of vertices, so it would be prudent to give spheres all the properties of `3d-objects`:

```
;; Declare that 'sphere inherits all properties of '3d-objects
(declare-type-inherits 'sphere '3d-objects)
```

Once a form is declared, it becomes available for the programmer to get and set properties.

```
;; Return the property of form, or #f if there has been no binding
(get-property form property)

;; Set the property of form to be value
(set-property form property value)
```

It is important to note that nothing about Organon's type system limits the programmers ability to set and get arbitrary properties, it just makes it easier for constraints to check that supplied forms have various properties.

## 3.2 Constraints

Constraints represent an abstraction that allow a programmer to specify how “satisfied” he or she is with the state of the world. Organon presents two types of constraints: basic and compound. The former expresses constraints over forms, and the latter over other constraints. These two types ought be sufficient to express the vast majority of real world constraints, but Organon allows the constraint system to be extended if needed.

To create a constraint the programmer must provide two and optionally a third piece of information:

1. **operands:** N operands of the constraint, a.k.a “dependencies” of that constraint.
2. **constraint-function:** A function of N arguments that when applied to operands yields a value between 0.0 and 1.0, where 0.0 is completely unsatisfied and 1.0 is completely satisfied.
3. (optional) **hint-function:** A function of N arguments that when applied to operands yields a set of potential new bindings for forms that would improve the constraint's satisfaction. Usually this is only provided for basic forms.

Both basic and compound constraints share the same basic method signature:

```
(make-constraint type operands constraint-function #!optional
  hint-function)
```

For convenience, Organon supplies `make-compound-constraint` and `make-basic-constraint` that automatically bind `type` to the correct entity. One can think of a constraint made in this fashion to be a node in a constraint graph, where edges connect that node to all depended forms or constraints. Nodes can be “dirty” if their dependencies have changed recently, or “clean” if their dependencies have not changed since the last evaluation. Organon keeps track of this information for the programmer and automatically marks nodes as clean and dirty as needed. For example, when a form binding is changed, that automatically marks basic constraints dependent on that form as dirty so that the next time they are evaluated they will use the new binding of that form.

Internally, `make-constraint` creates a scheme entity that in addition to storing the clean/dirty state, has the following calling convention:

```
;; Applies constraint-function to operands if the node is ``dirty'',
  otherwise returns the last known value
(constraint)
```

```
;; Applies the hint-function to operands
(constraint 'hint)
```

```
;; Returns the operands
(constraint 'children)
```

```
;; Applies constraint-function to alternative-arguments
(constraint 'eval alternative-arguments)
```

```
;; Returns the type of operands expected ('form or 'constraint)
(constraint 'type)
```

```
;; Forces application of constraint-function to operands regardless of the
  cleanliness of the constraint.
(constraint 'force)
```

The constraint system was architected to be highly generic, and to that end it is very easy to extend with additional types of constraints, e.g. those operating not over forms or other constraints. All state is maintained in the `mit-scheme` entity, and the vast majority of state is provided by the closure over the entity.

### 3.3 Solvers

All solvers conform to a common interface by convention. A solver takes as arguments the forms that represent the world state, and the objective constraints from the constraint network that the solver will attempt to satisfy. The solvers operate by use of “hint” functions provided by the constraints. The solver can apply a constraint’s hint function to improve constraint satisfaction; applying a hint from one constraint, however, might decrease the satisfaction of any number of other constraints. Thus the solver acts as an optimizer, exploring the possible hint applications in the hopes of finding either a perfect satisfaction or the one closest to fully satisfied. Since hints are full-fledged functions, not static recommendations, they may change as the solver runs, since they can depend on the current form bindings. This gives an enormous amount of power to the user to guide solvers into specific solutions.

The solver library also provides substantial utility functions to minimize the code required to write a specific solvers. Most notable is the `iteratively-score-hints` function, which takes a scoring function and a visualizer function as input and calls each of them after having applied one of the hints given to the world state. We developed two independent solvers for Organon, an iterative exhaustive solver, and an iterative annealing solver.

### 3.3.1 Solver 1: Exhaustive

```
(define (basic-iterative-solver forms objective-constraints)
```

The first is an exhaustive solver. Given the objective constraints, it explores the set of all possible subsets of the set of hints attached to the children of those objective constraints. For each subset, it applies the bindings given by the hint function, which triggers all of the constraint values to be recomputed. On each iteration, the exponential solver returns a listing mapping each set to its score (as determined by the provided scoring function, which may take into account weights on the objective constraints).

The exhaustive solver is exponential in complexity. For the set of bindings given by a hint, each binding is applied one at a time, and thus the constraint network updates after each binding. Each iteration of the exhaustive solver takes, in the worst case  $O(2^h * h * n)$  time, where  $h$  is the number of hints and  $n$  is the number of constraints. In the typical case where the height of the constraint tree is logarithmic in  $n$ , the complexity is  $O(2^h * h * \log(n))$ .

### 3.3.2 Solver 2: Annealing

```
(define (basic-annealing-solver forms objective-constraints iterations)
```

The second solver is an “annealing” solver. This solver takes the set of all hints and then randomly chooses a subset of those hints to apply. The interface to the annealing solver is identical, with an added parameter for the number of iterations to perform. Each iteration is  $O(h * \log(n))$  in typical complexity and  $O(h * n)$  in worst case complexity, where  $h$  is the number of hints generated and  $n$  is the size of the constraint graph. The intuition behind this solver is that there is no way to know which combination of hints lead to a solution state, but the hints recompute every iteration, so we can always fix ourselves later by taking a different set of hints. The annealing comes from decreasing this probability with time, which guarantees that after a certain amount of time we will settle at some state, even if that state is not globally optimal.

The annealing solver does not provably find a solution state, it instead relies on the hint functions to return good hints that move the state of the system quickly towards a solution. It is important to note that the hints are local as generally there is no knowledge of global state accessible to constraints. While this may make many computer scientists and mathematicians nervous because we cannot guarantee or prove much, in the real world this solver is often sufficient to find goal states and does so very quickly. In particular, this solver is useful because while it may not return a globally optimal solution, it does return a locally optimal solution, and those tend to be sufficient. For example, we implemented graph coloring with the annealing solver and while it can theoretically not find a solution, it usually quickly finds a decent solution and in all cases we tested it on, it found a correct coloring.

## 4 Example Uses of Organon

As the purpose of Organon was to provide a general framework to reason about constraints, we present four demonstrations of its capabilities in three different problem domains: engineering, economics, and mathematics.

For visualization purposes, we wrote `opengl-streamer.scm`, which makes a socket connection to a multithreaded C++ server, which parses the incoming message and converts it to OpenGL objects. The message protocol supports OpenGL primitives (box, sphere, cylinder) as well as raw vertex lists. This allows us to visualize constraint application in real time.

We also wrote a simplified REPL for loading and running demo programs.


### 4.1 Ladder

The ladder demo (`ladder.scm`) defines two constraints, `hands-far-away` and `hands-end-of-rung`. These constraints provide hints which move the hands incrementally farther away from each other and incrementally towards the end of the rung, respectively.

```
(define hands-far-away
  (make-basic-constraint
    '(left-hand right-hand desired-distance)
    ;; lots of calls in between to extract the 3D components of
    interest
    (make-binding-list
      (make-binding
        left-hand
        (list 'frame (make-frame (add-vector left-hand-vector
                                              left-hand-inverted)
                                  left-hand-quat)))
      ;; right hand is symmetric, omitted for brevity
```

Finally, the demo uses the compound constraint `hands-on-ladder` to wrap both of the subconstraints. The satisfaction of `hands-on-ladder` is the average of the satisfactions of the subconstraints.

```
(define hands-on-ladder
  (make-compound-constraint
    (list hands-far-away hands-end-of-rung)
    ...
```

When run with the exponential solver, the ladder demo rapidly explores the space of hinted hand positions and finally settles on a final binding that puts the hands on opposite ends of the rung, as intended. 

### 4.2 Big Bang

This demo (`big-bang.scm`) defines one hundred "star" forms, each of which have a center and radius. The forms begin at the origin, and there is a single basic constraint: the universe, which calculates center of mass of the universe and hints each star to move slightly away from that center of mass.

```
(define (universe-constraint d . forms)
  ...
```

```

(define (score-single form)
  (let* ((pos (get-property form 'center))
        (dis (distance center-of-mass pos)))
    (min 1.0 (/ dis (get-value d)))))
(/ (apply + (map score-single forms)) (length forms)))

```

The annealing solver is used to run this demo; obviously, the exponential solver would have bad time with a  $2^{100}$  search space. On each iteration, the annealing solver randomly moves the “star” forms further away. When the network-visualizer function is passed to the solver, the resulting universe expansion can be visualized in OpenGL. `jimagej`

### 4.3 Laffer Curve

This demo (`laffer.scm`) defines one hundred “tax-payer” forms, each of which have plausible weekly wage rates, maximum number of hours they are willing to work, and their liberalness (how willing they are to work given taxation). The forms are bound by a single constraint, the laffer constraint. This expresses that the government attempts to maximize tax revenue from taxpayers who work fewer hours based on the given tax rate:

```

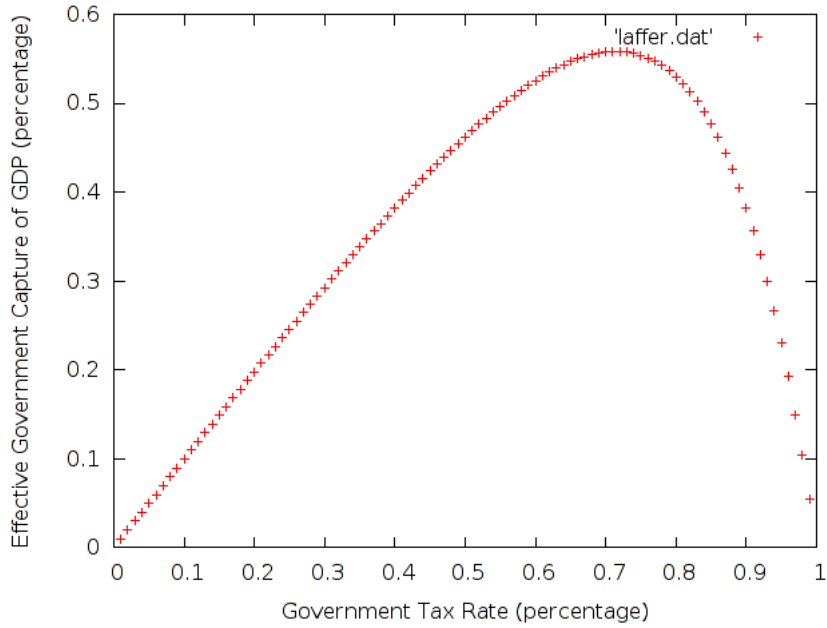
;; constraint returns a value in range [0.0, 1.0] is (revenue at tax-rate /
;; maximum possible revenue)
(define (laffer-constraint tax-rate . taxpayers)
  (/ (apply + (map (lambda (ith-taxpayer)
    ;; (1 - t^1) * maximum hours = number of hours worked
    ;; * hourly wages = output of this person
    ;; * tax rate = taxes collected by the government
    (* (- 1.0 (expt (get-value tax-rate) (get-property
      ith-taxpayer 'liberalness)))
      (get-property ith-taxpayer 'max-hours-worked)
      (get-property ith-taxpayer 'hourly-wage)
      (get-value tax-rate)))
    taxpayers))
    (apply + (map (lambda (ith-taxpayer)
    ;; Maximum possible output of the economy
    (* (get-property ith-taxpayer 'max-hours-worked)
      (get-property ith-taxpayer 'hourly-wage)))
    taxpayers))))

```

This constraint hints that taxes should decrease by 1 percent, and since taxation starts at 100%, this has the effect of causing the solver to explore the entire taxation rate from 100% down to 0%. The annealing solver is again used, and we can use the outputs of the solver to construct the laffer curve as seen in Figure 1



Figure 1: Simulated Laffer Curve

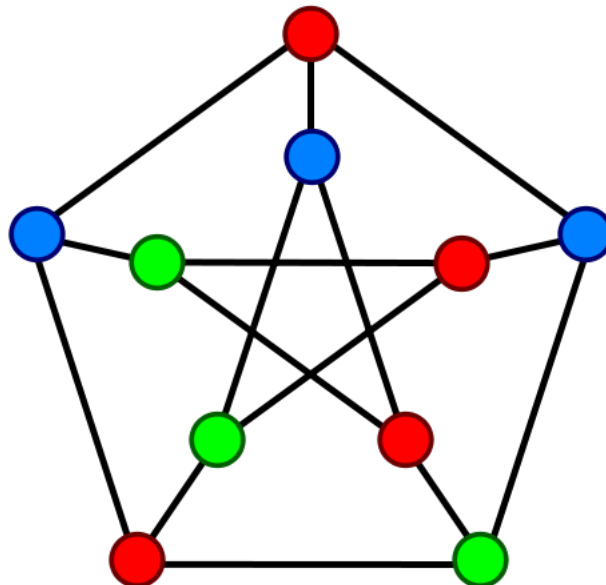


Apparently, with our simplistic model, an effective government tax rate of 71% would mean maximum government revenue. Note that this is not the same as maximizing societal welfare or happiness, and an interesting addition would be to add another constraint that optimizes for that.

#### 4.4 Graph Coloring

This demo (hard-coloring.scm) defines node forms that can have a color and then constraints for each edge between two nodes such that we construct the classic Peterson Graph seen in Figure 2. The annealing solver is then used to find a potential coloring, often completing within 100-120 iterations.

Figure 2: Peterson Graph with Valid 3-Coloring



All of the graph coloring demos use the same basic constraint and hint defined on any two nodes

```
;; Neighboring Nodes must be colored differently
(define (Node-constraint x y)
  (if (not (eq? (get-property x 'color) (get-property y 'color)))
      1.0
      0.0))

;; If two nodes are different colors, with high probability stay the same
;; otherwise switch one of them to another viable coloring
;;
;; If two nodes are the same color, change one of them to be a different
;; color
(define (Node-hint x y)
  (let ((x-color (get-property x 'color))
        (y-color (get-property y 'color)))
    (if (null? x-color)
        (make-binding-list
         (make-binding x (list 'color (car (get-value 'colors))))))
    (cond
     ;; Different colors already? with high probability hint at nothing
     ((and (eq? x-color y-color) (> (random 5) 2))
      (make-binding-list))
     (else
      ;; Hint at a randomly chosen alternative for y
      (make-binding-list
       (random-choice
        (map (lambda (color)
               (make-binding y (list 'color color)))
              (filter (lambda (z) (not (eq? z x-color))) (get-value
                                                           'colors))))))))))
```

This relies on local swaps with probabilistic transitions to color the graph. This is not provably correct, but in all of the demonstration graphs provided, it works very quickly (and finds an acceptable answer within 1 or 2 iterations). While this isn't particularly useful for proving that a graph is  $n$ -colorable, it is very useful for situations where coloring need only be approximate and you know the number of available colors, such as register allocation within a compiler.

## 5 Future Work & Conclusions:

Organon presents a powerful and generic way for programmers to specify constraints in real world problems as well as easily build solvers to explore those constraint domains. It relies on numerous concepts learned in 6.945 such as generic operations, propagators, and the idea that flexibility is sometimes more important than correctness. Although we cannot prove its usefulness, and we cannot prove its correctness, if there is one thing that 6.945 teaches, it is that nobody really can. Organon focuses on being generally useful, on allowing the end user to use the system in the ways that they see fit.

Organon's provided solvers are sufficient for many problems, but the most difficult constraint optimization and satisfaction problems require techniques far more advanced than those presented here. An interesting avenue of research would therefore be to explore additional optimization techniques such as complete simulated annealing with variable probabilistic transitions, hill climbing, particle swarm optimization, et. al.

The main disadvantage to Organon’s powerful constraint system is that the constraint functions—though often simple in concept—are not readable except to a reasonably sophisticated expert. In the future, we might build a dedicated constraint language to eliminate much of the verbosity of that is currently involved in ensuring that the forms passed to a constraint function are of the appropriate type and extracting and manipulating their properties.

All code is available at <https://github.com/jolynch/organon>.

## A Appendix A - Library

### A.1 forms.scm

```
(define *debug* #f)

;; Mapping of forms to dependent constraints
(define *forms* (make-eq-hash-table))
;; Mapping of constraints to dependent constraints
(define *constraints* (make-eq-hash-table))

;; Convenience methods to deal with constraint entity state
(define (dirty? entity)
  (eq? (car (entity-extra entity)) 'dirty))

(define (make-dirty entity)
  (set-entity-extra! entity '(dirty)))

(define (make-clean entity value)
  (set-entity-extra! entity (cons 'clean value)))

(define (get-cached entity)
  (cdr (entity-extra entity)))

;; Methods to cause propagation
(define (update-form form)
  (let ((constraints (eq-get *forms* form)))
    (propagate-to '() constraints)))

;; Causes all dependent constraints to fire
(define (propagate constraint)
  (let ((constraints (eq-get *constraints* constraint)))
    (propagate-to constraint constraints)))

;; Actually call the dependent constraints
;; TODO: pass list of calling dependencies to detect loops
(define (propagate-to orig-constraint constraints)
  (if constraints
    (for-each (lambda (constraint)
      (if (not (eq? orig-constraint constraint))
        (make-dirty constraint))
      (constraint))
    constraints)))

(define (show-form-mapping)
  (pp (cdr (eq-plist *forms*))))
```

```

(define (register-with node name item)
  (let ((val (eq-get node item)))
    (cond (val (eq-put! node item (cons name val)))
          (else (eq-put! node item (list name))))))

(define (register-form name form)
  (register-with *forms* name form))

(define (register-constraint name constraint)
  (register-with *constraints* name constraint))

;; Constraints have the following calling convention
;; (constraint) -> if the node is dirty, it will evaluate the constraint,
;; otherwise it will use the value last returned. This results in a float
;; between 0 and 1
;;
;; (constraint 'hint) -> Returns a list of hints where hints are an assoc
;; list
;; of form (symbol) -> list of bindings (assoc list), which provides
;; suggested
;; improvements to the bindings
;;
;; (constraint 'children) -> Returns a list of the constraints that are
;; children of this constraint
;;
;; (constraint 'eval arg1 arg2 ,,,) -> Returns the result of applying func
;; to
;; the supplied arguments
;;
;; (constraint 'type) -> Returns the type of the constraint, currently just
;; basic and compound are supported
;;
;; (constraint 'force) -> Force an evaluation of the function,
;; disregarding the
;; cached value and dirty status of the node
;;
;; (constraint 'leaf?) -> Tests whether this is a leaf node, used by the
;; solver
;; to select nodes dependent on forms

(define (dispatch-constraint type self args operands func hint-func)
  ;; Allows dispatch on supplied type
  (define (test-args pattern)
    (and (not (null? args)) (symbol? (car args)) (eq? (car args) pattern)))

  ;; We should allow people to use hint as a form
  (cond
    ((test-args 'hint)
     (apply hint-func operands))
    ((test-args 'children)
     operands)
    ((test-args 'eval)
     (apply func (cdr args)))
    ((test-args 'type)
     type)
    ((test-args 'leaf?)
     (eq? type 'form))
  ))

```

```

((test-args 'force)
 (apply func operands))
(dirty? self)
(if *debug* (pp "Evaluating constraint")))
(let ((value (apply func operands)))
  (make-clean self value)
  (propagate self)
  value))
(else
 (if *debug* (begin (display "Using cached value for ") (write
  self)(newline)))
 (get-cached self))))

(define (make-basic-constraint forms func #!optional hint-func)
  (make-constraint 'form forms func hint-func))

(define (make-compound-constraint constraints func #!optional hint-func)
  (make-constraint 'constraint constraints func hint-func))

(define (make-constraint type operands func #!optional hint-func)
  (define me
    (make-entity
     (lambda (self . args)
       (dispatch-constraint type self args operands func hint-func))
     '(dirty)))
  (let ((register-function (eval (symbol 'register- type)
    user-initial-environment)))
    (for-each (lambda (operand) (register-function me operand)) operands))
  me)

```

## A.2 constraints.scm

```

(define (has-property? form property)
  (if (get-property form property) #t #f))

(define (get-property form property)
  (eq-get form property))

;; TODO - decide whether to convert this to a continuation form rather
;; than calling update-form immediately
(define (set-property form property value #!optional update)
  (let ((val (eq-put! form property value)))
    (if (default-object? update)
      (update-form form))
    val))

(define (capture-bindings form)
  (eq-plist-simple form))

(define (apply-bindings form bindings)
  (for-each (lambda (binding)
    (set-property form (car binding) (cdr binding))) bindings))

(define (same-type? form-a form-b)
  (equal? (eq-ordered-plist form-a) (eq-ordered-plist form-b)))

;; Tells if a form implements the interface provided by type
(define (is-type? form type)

```

```

(define (implements? form type)
  (if (eq-get 'form-types type)
      (let ((desired-properties (eq-get 'form-types type)))
        (reduce (lambda (x y) (and x y)) #t
                  (map (lambda (x) (has-property? form x))
                       desired-properties)))
      #f))
(reduce (lambda (x y) (and x y)) #t
        (map (lambda (x) (implements? form x)) (all-parents type))))

;; Gets all parent types for a given type
(define (all-parents type)
  (let find-parents ((result (list type))
                     (ntype type))
    (if (eq-get 'form-inherits ntype)
        (find-parents (cons (eq-get 'form-inherits ntype) result) (eq-get
                                                                    'form-inherits ntype))
        result)))

;; Check if a form is many types
(define (is-multiple-type? form types)
  (reduce (lambda (x y) (and x y)) #t
          (map (lambda (x) (is-type? form x)) types)))

(define (pp-form form) (pp (capture-bindings form)))

;; TODO
;;(define (sub-type? form-a form-b)
;;
;;
;;)

;; makes a form from a properties-and-values, a list of tuples (name,
;; value)
(define (make-form-by-properties name properties-and-values)
  (for-each (lambda (x)
              (let ( (property-name (first x)) (property-value (second x)) )
                (eq-put! name property-name property-value)
              )) properties-and-values))

;; declare-form-by-properties - declares the form and initializes all the
;; properties to null (ie the empty list)
(define (declare-form-by-properties name properties)
  (make-form-by-properties name (map (lambda (x) (list x '())) properties)))

;; register a type with a list of properties
(define (declare-form-type type properties) (eq-put! 'form-types type
                                                       properties))

(define (make-form name type properties)
  (make-form-by-properties name (zip (eq-get 'form-types type)
                                     properties)))

;; declaring a form creates its superclass properties. making a form does
;; not.
(define (declare-form name type)
  (if (not (equal? #f type))
      (let ( (parent-type (eq-get 'form-inherits type))
```

```

      (my-type (eq-get 'form-types type)) )
    (if (not (equal? parent-type #f)) (declare-form name parent-type))
    (declare-form-by-properties name my-type))))

(define (declare-type-inherits type-a type-b)
  (eq-put! 'form-inherits type-a type-b))

```

### A.3 solver.scm

```

(define *solver-debug* #f)

;; pretty print debug
(define (ppd . x) (if (or *debug* *solver-debug*) (apply pp x)))
;; pretty print debug short (no newline)
(define (ppds . x) (if (or *debug* *solver-debug*) (apply display x)))

;; Iteratively apply constraints to the objects in the world. The action
  taken
;; when a constraint fails is implementation/mode specific. Rely on
  constraint
;; propagation to actually propagate changes through the network.
  Backtracking,
;; if needed, will occur at the constraint level.
(define (iteratively-score-hints list-of-hints scoring-function
  visualizer-function)
  ;; forms-to-hints is an assoc list mapping a form to an list of hints
  (ppds "list-of-hints-is") (ppd list-of-hints)
  (map (lambda (form-bindings-pair)
    (ppds "form->bindings-assoc-list-is") (ppd form-bindings-pair)
    (let ( (form (car form-bindings-pair)) (bindings (car (cdr
      form-bindings-pair))) )
      (ppds "binding-are") (ppds bindings)
      (ppds "examining-form:") (ppd form)
      (apply-bindings form bindings)
      (display "bound-form:") (display form) (pp-form form) (newline)
      (visualizer-function)
      (scoring-function)
    )
  ) list-of-hints))

;; given a single objective, returns an list of leaf constraints (which may
;; include the objective constraint, if it is a leaf)
;; TODO: convert to a set so we don't have duplicate leaves included
(define (get-constraint-leaves objective-constraint)
  (if (objective-constraint 'leaf?)
    (list objective-constraint)
    (map (lambda (x) (car (get-constraint-leaves x)))
      (objective-constraint 'children))))

(define (get-hints target-constraint)
  (target-constraint 'hint))

(define (join-lists list-of-lists)
  (fold-right (lambda (a b) (append a b)) '() list-of-lists))

(define (network-visualizer all-forms)
  (if *use-network-visualizer*
    (write-forms all-forms)))

```

```

;; iterate recursively over the objective constraints, descending into
    their
;; children (not implemented yet) and calling the hint-iterator on each of
    them
;; with the passed scoring-function.
(define (iterative-solver forms objective-constraints scoring-function)
  (let* ( (root-bindings (map capture-bindings forms))
          (restore-root-bindings (lambda () (apply-bindings forms
                                                                root-bindings)))
          (all-constraint-leaves (car (map get-constraint-leaves
                                             objective-constraints)))
          (all-hints (join-lists (map get-hints all-constraint-leaves))))

    (display "all-constraint-leaves_") (pp all-constraint-leaves)
    (display "all_hints") (pp all-hints)
    (display "all_subsets_of_hints") (pp (non-empty-subsets all-hints))

    ;; generate the set of all possible subsets of hints, then compute
        their scores
    (let* ((all-hints-subsets (non-empty-subsets all-hints))

           (accumulated-hint-scores (map (lambda (hint-subset)
                                           (restore-root-bindings)
                                           (iteratively-score-hints
                                            hint-subset scoring-function
                                            (lambda ()
                                              (network-visualizer
                                               forms)))
                                           ) all-hints-subsets)) )

      (restore-root-bindings)
      (pp accumulated-hint-scores)
    )))

;;
;; BEGIN ANNEALING SOLVER
;;
;;
;; Choose hints to apply
(define (anneal-choose hints iteration prob)
  (let ((chosen-hints
        (let loop ((result '())
                    (remaining hints))
          (if (null? remaining) result
              (let ((value (car remaining)))
                (if (< (random 1.0) prob)
                    (loop (cons value result) (cdr remaining))
                    (loop result (cdr remaining)))))))
    (better-bindings chosen-hints)))

;; Takes bindings of the following form and applies them:
;; ((form property value) ...)
(define (apply-better-bindings chosen-bindings)
  (for-each
   (lambda (binding)
     (set-property (first binding) (second binding) (third binding)))
   chosen-bindings))

```



```

;; PPrints the state so you can see what's going on
(define (show-state forms)
  (for-each (lambda (form)
              (display "Form:~") (write form) (newline)
              (display "Bindings:~") (pp-form form)) forms))

;; The annealing solver, which tries to maximize the scoring function,
  cooling
;; down over time and making fewer transitions. If we hit iterations then
  we
;; stop and return the best answer so far
(define (annealing-solver o-forms objectives scoring temperature
  iterations)
  (let solve ((best-binding (map capture-bindings o-forms))
              (best-value (scoring))
              (forms o-forms)
              (objective-constraints objectives)
              (scoring-function scoring)
              (temp temperature)
              (iter iterations))
    (let* ((all-constraint-leaves (car (listify (map get-constraint-leaves
      objective-constraints))))
           (all-hints (join-lists (map get-hints all-constraint-leaves)))
           (converted-hints (filter (lambda (result) (not (null? result)))
      (map (lambda (hint)
              (better-bindings (list hint)))
      all-hints)))
           (all-bindings (remove-dups converted-hints)))
      ;; generate the set of all possible hints then apply them randomly
      ;; based on the temperature
      (let ((chosen-bindings (anneal-choose all-hints 0 temp)))
        (apply-better-bindings chosen-bindings)
        (network-visualizer forms)
        (let ((score (scoring-function)))
          (cond
            ((< iter 0)
             (pp "Exceeded maximum iterations, best answer is:")
             (for-each (lambda (binding)
                         (apply-bindings (car binding) (cdr binding)))
                       best-binding)
             (show-state forms)
             (display "Got top score:~") (display best-value) (newline))
            (> score .98)
             (display "Found solution state with score~|")
             (display score) (display "~| after ~#") (display (- iterations
              iter))
             (display "~iterations.") (newline)
             (show-state forms))
            (else
             (display "Trying again~") (write score) (display "~is not good~
              enough!~with temp~")
             (write temp) (newline)
             (if (> score best-value)
               (begin
                (solve (map (lambda (form) (cons form (tree-copy
                  (capture-bindings form)))) forms)

```

```

        score
        forms objective-constraints scoring-function (*
            .9999 temp) (- iter 1)))
    (solve best-binding
        best-value
        forms objective-constraints scoring-function (*
            .9999 temp) (- iter 1)))))))))

;; Scoring functions
(define (simple-scoring-func objective-constraints
    objective-constraint-weights )
    (lambda ()
        (let* ( (scores (map (lambda (x) (x)) objective-constraints))
            (weights-and-scores (zip scores objective-constraint-weights))
            )
            (apply + (map (lambda (x) (apply * x)) weights-and-scores))
            )))

;; wrapper to create a solver with a basic scoring function
(define (basic-iterative-solver forms objective-constraints)
    (iterative-solver forms objective-constraints
        (simple-scoring-func objective-constraints (make-list
            (length objective-constraints) 1))))

(define (weighted-iterative-solver forms objective-constraint
    objective-constraint-weights)
    (iterative-solver forms objective-constraints
        (simple-scoring-func objective-constraints
            objective-constraint-weights)))

(define (basic-annealing-solver forms objective-constraints iterations)
    (pp "Initial_state:")
    (show-state forms)
    (newline)
    (annealing-solver forms objective-constraints
        (simple-scoring-func objective-constraints (make-list
            (length objective-constraints) 1)) 1 iterations))

(define (weighted-annealing-solver forms objective-constraints
    objective-constraint-weights iterations)
    (pp "Initial_state:")
    (show-state forms)
    (newline)
    (annealing-solver forms objective-constraints
        (simple-scoring-func objective-constraints
            objective-constraint-weights) 1 iterations))

;; we will have two initial solver implementations

;; hill-climber - tries to maximize a weighted sum of objective-constraint
;; satisfaction ratings by examining all of the various constraint hints
;; (perhaps using simulated annealing)

;; absolute-solver - if it is discovered that all objective-constraints
;; have
;; been satisfied (are rated 1.0), terminate immediately. if it is
;; discovered
;; that one constraint cannot be satisfied (is rated < 1.0), give up

```

```
;; immediately.
```

## A.4 util.scm

```
;; Make nodes a list if it's not a list
(define (listify nodes)
  (cond
    ((list? nodes) nodes)
    (else (list nodes))))

;; subsets based on
http://pages.cs.wisc.edu/~fischer/cs538.s08/lectures/Lecture13.4up.pdf
(define (subset-extend L E)
  (append L (subset-distrib L E)))

(define (subset-distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L)) (subset-distrib (cdr L) E))))

(define (subsets L)
  (if (null? L)
      (list ())
      (subset-extend (subsets (cdr L)) (car L))))

(define (non-empty-subsets L)
  (filter (lambda (x) (not (null? x))) (subsets L)))

(define (assert expression)
  (if (not (expression)) (error (string-append "assertion failed for "
expression:~" expression))))

;; Binding convenience methods so that we don't have to constantly do list
;; constructions in the hint functions
;;
;; Bindings are of the form:
;; ((property . binding-value) (property . binding-value) ...)
;;
;; Binding-values are of the form
;; ((form . binding) (form . binding) ...)
;;
(define (make-binding-list . bindings)
  bindings)

(define (make-property-binding property value)
  (cons property value))

(define (make-binding form . property-bindings)
  (cons form
    (list (map (lambda (pb) (make-property-binding (car pb) (cadr pb)))
      property-bindings))))

(define (bindings-for bindings form)
  (assoc-get form bindings))

(define (binding-form binding)
```

```

(car binding))

(define (binding-properties binding)
  (sort (map car (cadr binding)) symbol<?))

(define (assoc-get object alist)
  (let ((value (assoc object alist)))
    (cond
      ((eq? value #f) #f)
      ((list? value) (cadr value))
      ((pair? value) (cdr value))
      (else #f))))

(define (better-bindings bindings)
  (remove-duplicates (convert-bindings bindings)))

;; Takes a list of bindings and converts them to a list of
;; (form property value)
(define (convert-bindings bindings)
  (map
    (lambda (triple)
      (list (car triple) (cadr triple) (caddr triple)))
    (join-lists (map
      (lambda (binding)
        (map (lambda (var) (cons (binding-form binding) var))
          (cadr binding))) bindings))))

;; Says if a particular form, property tuple is present in a given list of
triples
(define (fp-present? lst form property)
  (if (null? lst) #f
    (let ((item (car lst)))
      (or (and (equal? (car item) form)
        (equal? (cadr item) property))
        (fp-present? (cdr lst) form property)))))

;; Remove duplicate assignments
(define (remove-duplicates triples)
  (let accum ((result '())
    (remaining triples))
    (if (null? remaining) result
      (let ((value (car remaining)))
        (if (fp-present? result (car value) (cadr value))
          (accum result (cdr remaining))
          (accum (cons value result) (cdr remaining)))))))

;; Remove duplicates in general
(define (remove-dups lst)
  (let accum ((result '())
    (remaining lst))
    (if (null? remaining) result
      (let ((value (car remaining)))
        (if (member value result)
          (accum result (cdr remaining))
          (accum (cons value result) (cdr remaining)))))))

;; Take a random choice from a list
(define (random-choice lst)

```

```
(let ((v (list->vector lst)))
  (vector-ref v (random (length lst)))))
```

## A.5 opengl-stream.scm

```
;; Make nodes a list if it's not a list
(define (listify nodes)
  (cond
    ((list? nodes) nodes)
    (else (list nodes))))

;; subsets based on
http://pages.cs.wisc.edu/~fischer/cs538.s08/lectures/Lecture13.4up.pdf
(define (subset-extend L E)
  (append L (subset-distrib L E)))

(define (subset-distrib L E)
  (if (null? L)
      ()
      (cons (cons E (car L)) (subset-distrib (cdr L) E))))

(define (subsets L)
  (if (null? L)
      (list ())
      (subset-extend (subsets (cdr L)) (car L))))

(define (non-empty-subsets L)
  (filter (lambda (x) (not (null? x))) (subsets L)))

(define (assert expression)
  (if (not (expression)) (error (string-append "assertion failed for "
expression:~" expression))))

;; Binding convenience methods so that we don't have to constantly do list
;; constructions in the hint functions
;;
;; Bindings are of the form:
;; ((property . binding-value) (property . binding-value) ...)
;;
;; Binding-values are of the form
;; ((form . binding) (form . binding) ...)
;;
(define (make-binding-list . bindings)
  bindings)

(define (make-property-binding property value)
  (cons property value))

(define (make-binding form . property-bindings)
  (cons form
    (list (map (lambda (pb) (make-property-binding (car pb) (cadr pb)))
      property-bindings))))

(define (bindings-for bindings form)
  (assoc-get form bindings))
```

```

(define (binding-form binding)
  (car binding))

(define (binding-properties binding)
  (sort (map car (cadr binding)) symbol<?))

(define (assoc-get object alist)
  (let ((value (assoc object alist)))
    (cond
      ((eq? value #f) #f)
      ((list? value) (cadr value))
      ((pair? value) (cdr value))
      (else #f))))

(define (better-bindings bindings)
  (remove-duplicates (convert-bindings bindings)))

;; Takes a list of bindings and converts them to a list of
;; (form property value)
(define (convert-bindings bindings)
  (map
    (lambda (triple)
      (list (car triple) (cadr triple) (cddr triple)))
    (join-lists (map
      (lambda (binding)
        (map (lambda (var) (cons (binding-form binding) var))
          (cadr binding))) bindings))))

;; Says if a particular form, property tuple is present in a given list of
triples
(define (fp-present? lst form property)
  (if (null? lst) #f
    (let ((item (car lst)))
      (or (and (equal? (car item) form)
        (equal? (cadr item) property))
        (fp-present? (cdr lst) form property)))))

;; Remove duplicate assignments
(define (remove-duplicates triples)
  (let accum ((result '())
    (remaining triples))
    (if (null? remaining) result
      (let ((value (car remaining)))
        (if (fp-present? result (car value) (cadr value))
          (accum result (cdr remaining))
          (accum (cons value result) (cdr remaining)))))))

;; Remove duplicates in general
(define (remove-dups lst)
  (let accum ((result '())
    (remaining lst))
    (if (null? remaining) result
      (let ((value (car remaining)))
        (if (member value result)
          (accum result (cdr remaining))
          (accum (cons value result) (cdr remaining)))))))

;; Take a random choice from a list

```

```
(define (random-choice lst)
  (let ((v (list->vector lst)))
    (vector-ref v (random (length lst)))))
```

## B Appendix B - Demos

### B.1 ladder.scm

```
(define *demo-debug* #t)
(define *debug* #f)
(define *use-network-visualizer* #f)

(declare-form 'left-hand '3D-hand-form)
(declare-form 'right-hand '3D-hand-form)
(declare-form 'rung '3D-rung)
(declare-form 'desired-distance 'basic)
(declare-form 'desired-closeness 'basic)
(declare-form 'axis 'basic)

;; Set up the initial conditions

(set-property 'left-hand 'vertices (list (make-vertex -2 -1 0)
                                           (make-vertex -2 1 0)
                                           (make-vertex 0 1 0)
                                           (make-vertex 0 -1 0)))

(set-property 'left-hand 'frame
              (make-frame (make-vertex -1 0 1) (make-identity-quaternion)))

(set-property 'right-hand 'vertices (list (make-vertex 2 -1 0)
                                           (make-vertex 2 1 0)
                                           (make-vertex 0 1 0)
                                           (make-vertex 0 -1 0)))

(set-property 'right-hand 'frame
              (make-frame (make-vertex 1 0 1) (make-identity-quaternion)))

(set-property 'rung 'frame
              (make-frame (make-vertex 0 0 0) (make-quaternion 0 0.707 0
                                                                0.707)))
(set-property 'rung 'left-rung (make-vertex -10 0 0))
(set-property 'rung 'right-rung (make-vertex 10 0 0))
(set-property 'rung 'length 20.0)
(set-property 'rung 'radius 1.0)

(set-property 'desired-distance 'value 5)
(set-property 'axis 'value (make-vector 1 0 0))
(set-property 'desired-closeness 'value .5)

(define hands-far-away
  (make-basic-constraint
   '(left-hand right-hand desired-distance)
   (lambda (left-hand right-hand d)
     (cond ((and (is-type? left-hand '3D-hand-form)
                  (is-type? right-hand '3D-hand-form)
                  (is-type? d 'basic))
            (let* ((left-origin (car (get-property left-hand 'frame)))
                   (right-origin (car (get-property right-hand 'frame)))
                   (dis (distance left-origin right-origin)))
              (min 1.0 (/ dis (get-value d))))))
     (else 0.0))))
```



```

(lambda (left-hand right-hand d)
  (let* ((left-hand-old (get-property left-hand 'frame))
         (right-hand-old (get-property right-hand 'frame))
         (left-hand-vector (frame-vector left-hand-old))
         (right-hand-vector (frame-vector right-hand-old))
         (left-hand-quat (frame-quat left-hand-old))
         (right-hand-quat (frame-quat right-hand-old))
         (left-hand-inverted (scale-vector
                               (unit
                                (sub-vector right-hand-vector
                                              left-hand-vector))
                               -1))
         (right-hand-inverted (scale-vector
                                (unit
                                 (sub-vector left-hand-vector
                                              right-hand-vector))
                                -1)))
    (make-binding-list
     (make-binding
      left-hand
      (list 'frame (make-frame (add-vector left-hand-vector
                                              left-hand-inverted)
                                left-hand-quat)))
     (make-binding
      right-hand
      (list 'frame (make-frame (add-vector right-hand-vector
                                              right-hand-inverted)
                                right-hand-quat)))))))

(define hands-end-of-rung
  (make-basic-constraint
   '(left-hand right-hand desired-closeness rung)
   ;; Constraint is that the left hand should be near the left rung goal
   ;; and the right hand should be near the right rung goal
   (lambda (left-hand right-hand d rung)
     (cond
      ((and (is-type? left-hand '3D-hand-form)
            (is-type? right-hand '3D-hand-form)
            (is-type? d 'basic)
            (is-type? rung 'cylinder-form))
       (let* ((left-origin (car (get-property left-hand 'frame)))
              (right-origin (car (get-property right-hand 'frame)))
              (left-rung (get-property rung 'left-rung))
              (right-rung (get-property rung 'right-rung))
              (disl (distance left-origin left-rung))
              (disr (distance right-origin right-rung)))
         (min
          1.0
          (+ (/ (min 1.0 (/ (get-value d) disl)) 2.0)
             (/ (min 1.0 (/ (get-value d) disr)) 2.0))))))
      (else 0.0)))
   ;; Hint generates four possible solution points between each hand and
   ;; the goal rung
   (lambda (left-hand right-hand d rung)
     (let* ((left-hand-old (get-property left-hand 'frame))
            (right-hand-old (get-property right-hand 'frame))
            (left-hand-quat (frame-quat left-hand-old))
            (right-hand-quat (frame-quat right-hand-old))

```

```

(left-hand-vector (frame-vector left-hand-old))
(right-hand-vector (frame-vector right-hand-old))
(left-hand-rung (get-property rung 'left-rung))
(right-hand-rung (get-property rung 'right-rung))
(left-hand-goal (interpolate left-hand-vector left-hand-rung
2))
(right-hand-goal (interpolate right-hand-vector
right-hand-rung 2))
(left-hand-goal-frame
(map (lambda (goal)
(list 'frame (make-frame goal left-hand-quat)))
left-hand-goal))
(right-hand-goal-frame
(map (lambda (goal)
(list 'frame (make-frame goal right-hand-quat)))
right-hand-goal))
(join-lists (list (apply make-binding-list
(map (lambda (goal)
(make-binding left-hand goal))
left-hand-goal-frame))
(apply make-binding-list
(map (lambda (goal)
(make-binding right-hand goal))
right-hand-goal-frame))))))

(define hands-on-ladder
(make-compound-constraint
(list hands-far-away hands-end-of-rung)
(lambda (hfa heor)
(let ((h1 (hfa))
(h2 (heor)))
(/ (+ h1 h2) 2.0)))))

(pp "Final_hands-on-ladder_value:")(write (hands-on-ladder))(newline)

(if *use-network-visualizer* (begin (pp "making_connection")
(make-connection)))

(basic-iterative-solver '(rung left-hand right-hand) (list
hands-on-ladder))

(if *use-network-visualizer* (close-connection))

;;(iterative-solver '(left-hand right-hand) '(hands-on-ladder))

;; Two constraints, each with two hints - 4 hints total. 2^4 = 16 subsets.
And
;; indeed, we see 16 output scores. Each score is a list that shows the
score we
;; got from applying to first form that was in the hint, then the first two
;; forms that were in the hint, then the first three forms, etc.

```

## B.2 big-bang.scm

```

(define *use-network-visualizer* #f)
(define *demo-debug* #t)
(define *debug* #f)

```

```

(declare-form 'desired-distance 'basic)
(set-property 'desired-distance 'value 10)

(for-each (lambda (i) (declare-form (symbol 'star- i) 'star)) (range 0
100))
(declare-form 'goal-star 'star)
(set-property 'goal-star 'center (make-vertex 0 0 0))
(set-property 'goal-star 'radius (get-value 'desired-distance))

;; Set up the initial conditions

(for-each (lambda (i)
            (set-property (symbol 'star- i) 'radius 1)) (range 0 100))

(for-each (lambda (i)
            (set-property (symbol 'star- i)
                          'center
                          (make-vertex (random 1.0)
                                         (random 1.0)
                                         (random 1.0)))) (range 0 100))

(define (universe-constraint d . forms)
  (let* ((avgx (/ (apply + (map (lambda (f) (vx (get-property f 'center)))
                                forms))
                  (length forms)))
        (avgy (/ (apply + (map (lambda (f) (vy (get-property f 'center)))
                                forms))
                  (length forms)))
        (avgz (/ (apply + (map (lambda (f) (vz (get-property f 'center)))
                                forms))
                  (length forms)))
        (center-of-mass (make-vector avgx avgy avgz)))
    (define (score-single form)
      (let* ((pos (get-property form 'center))
             (dis (distance center-of-mass pos)))
        (min 1.0 (/ dis (get-value d)))))
    (/ (apply + (map score-single forms)) (length forms)))

(define (universe-hint d . forms)
  (let* ((avgx (/ (apply + (map (lambda (f) (vx (get-property f 'center)))
                                forms))
                  (length forms)))
        (avgy (/ (apply + (map (lambda (f) (vy (get-property f 'center)))
                                forms))
                  (length forms)))
        (avgz (/ (apply + (map (lambda (f) (vz (get-property f 'center)))
                                forms))
                  (length forms)))
        (center-of-mass (make-vector avgx avgy avgz)))
    (define (hint-single form)
      (let* ((pos (get-property form 'center))
             (pos-inverted (unit (sub-vector pos center-of-mass))))
        (make-binding form (list 'center (add-vector pos pos-inverted)))))
    (map hint-single forms)))

(define universe-exploded
  (make-basic-constraint

```

```

    (cons 'desired-distance (map (lambda (i) (symbol 'star- i)) (range 0
100)))
    universe-constraint
    universe-hint))

(if *use-network-visualizer* (begin (pp "making_connection")
    (make-connection)))

(basic-annealing-solver (cons 'goal-star (map (lambda (i) (symbol 'star-
i)) (range 0 100)))
    (list universe-exploded)
    100)

(if *use-network-visualizer* (close-connection))

;;(iterative-solver '(left-hand right-hand) '(hands-on-ladder))

;; Two constraints, each with two hints - 4 hints total. 2^4 = 16 subsets.
And
;; indeed, we see 16 output scores. Each score is a list that shows the
score we
;; got from applying to first form that was in the hint, then the first two
;; forms that were in the hint, then the first three forms, etc.

```

### B.3 laffer.scm

```

(define *use-network-visualizer* #f)
(define num-taxpayers 100)
(define exponent 10)

(declare-form 'govt-tax-rate 'basic)
(set-property 'govt-tax-rate 'value 1.00)

(declare-form-type 'taxpayer (list 'max-hours-worked 'hourly-wage
'liberalness))
(for-each (lambda (i) (declare-form (symbol 'taxpayer- i) 'taxpayer))
    (range 0 num-taxpayers))

;; each taxpayer earns a random amount between $0/hour and $60/hour
(for-each (lambda (i)
    (set-property (symbol 'taxpayer- i) 'hourly-wage (* 60 (random 1.0)))
    (set-property (symbol 'taxpayer- i) 'liberalness (+ 1 (random 10.0))))
    (range 0 num-taxpayers))

;; each taxpayer has a different linear work function that maps their tax
;; rate to the number of hours they are willing to work
(for-each (lambda (i)
    (set-property (symbol 'taxpayer- i) 'max-hours-worked (* 40 (+ 1 (random
1.0))))) (range 0 num-taxpayers))

;; constraint returns a value in range [0.0, 1.0] is (revenue at tax-rate /
;; maximum possible revenue)
(define (laffer-constraint tax-rate . taxpayers)
    (/ (apply + (map (lambda (ith-taxpayer)
        ;; (1 - t^1) * maximum hours = number of hours worked

```

```

;; * hourly wages = output of this person
;; * tax rate = taxes collected by the government
(* (- 1.0 (expt (get-value tax-rate) (get-property
    ith-taxpayer 'liberalness)))
    (get-property ith-taxpayer 'max-hours-worked)
    (get-property ith-taxpayer 'hourly-wage)
    (get-value tax-rate)))
taxpayers))
(apply + (map (lambda (ith-taxpayer)
    ;; Maximum possible output of the economy
    (* (get-property ith-taxpayer 'max-hours-worked)
        (get-property ith-taxpayer 'hourly-wage)))
    taxpayers))))

;; TODO: if the hints allow base-hours worked to change, this version of
the
;; constraint will consider the overall gov't revenue as well as the
happiness of
;; the citizens (presumably happiness will decrease as max-hours-worked
goes up)

;; four hints: + tax rate, - tax rate, + max-hours-worked, -
max-hours-worked
(define (laffer-hint tax-rate . taxpayers)
  ;; Everyone wants lower taxes!
  (make-binding-list (make-binding tax-rate (list 'value (max 0.01 (-
      (get-value tax-rate) 0.01)))))

(define laffer-curve
  (make-basic-constraint
    (cons 'govt-tax-rate (map (lambda (i) (symbol 'taxpayer- i)) (range 0
        num-taxpayers)))
    laffer-constraint
    laffer-hint))

(pp "starting")

;;(iterative-solver (map (lambda (i) (symbol 'taxpayer- i)) (range 0
    num-taxpayers)) (list laffer-curve))

;; maximize govt revenue as a percentage of GDP
(basic-annealing-solver (cons 'govt-tax-rate (map (lambda (i) (symbol
    'taxpayer- i)) (range 0 num-taxpayers))) (list laffer-curve) 100)

(display "tax_rate_settled_at_")
(pp (get-value 'govt-tax-rate))

```

## B.4 node-coloring.scm

```

(define *use-network-visualizer* #f)

(declare-form-type 'node (list 'color))
(declare-form 'colors 'basic)
(set-property 'colors 'value '(red blue green))

(declare-form 'A 'node)
(declare-form 'B 'node)
(declare-form 'C 'node)

```

```

(declare-form 'D 'node)

(define (Node-constraint x y)
  (if (not (eq? (get-property x 'color) (get-property y 'color)))
      1.0
      0.0))

(define (Node-hint x y)
  (let ((x-color (get-property x 'color))
        (y-color (get-property y 'color)))
    (if (null? x-color)
        (make-binding-list
         (make-binding x (list 'color (car (get-value 'colors)))))
        (cond
         ((and (eq? x-color y-color) (> (random 5) 2))
          (make-binding-list))
         (else
          (make-binding-list
           (random-choice
            (map (lambda (color)
                    (make-binding y (list 'color color)))
                  (filter (lambda (z) (not (eq? z x-color))) (get-value
                                                                'colors))))))))))

;; Graph of the form
;; A
;; | \
;; | C — D
;; | /
;; B
;;
;; Need to define a constraint for each pair of nodes
(define AB (make-basic-constraint '(A B) Node-constraint Node-hint))
(define AC (make-basic-constraint '(A C) Node-constraint Node-hint))
(define BC (make-basic-constraint '(B C) Node-constraint Node-hint))
(define CD (make-basic-constraint '(C D) Node-constraint Node-hint))

(define all-colored
  (make-compound-constraint
   (list AB AC BC CD)
   (lambda (one two three four)
     (let ((c1 (one))
           (c2 (two))
           (c3 (three))
           (c4 (four)))
       (/ (+ c1 c2 c3 c4) 4.0)))))

(basic-annealing-solver '(A B C D) (list all-colored) 1000)

```

## B.5 harder-coloring.scm

```

(define *use-network-visualizer* #f)

(declare-form-type 'node (list 'color))
(declare-form 'colors 'basic)
(set-property 'colors 'value '(red blue green))

```

```

(declare-form 'A 'node)
(declare-form 'B 'node)
(declare-form 'C 'node)
(declare-form 'D 'node)
(declare-form 'E 'node)
(declare-form 'F 'node)
(declare-form 'G 'node)
(declare-form 'H 'node)
(declare-form 'I 'node)
(declare-form 'J 'node)

;; Neighboring Nodes must be colored differently
(define (Node-constraint x y)
  (if (not (eq? (get-property x 'color) (get-property y 'color)))
      1.0
      0.0))

;; If two nodes are different colors, with high probability stay the same
;; otherwise switch one of them to another viable coloring
;;
;; If two nodes are the same color, change one of them to be a different
;; color
(define (Node-hint x y)
  (let ((x-color (get-property x 'color))
        (y-color (get-property y 'color)))
    (if (null? x-color)
        (make-binding-list
         (make-binding x (list 'color (car (get-value 'colors)))))
        (cond
         ;; Different colors already? with high probability hint at nothing
         ((and (eq? x-color y-color) (> (random 5) 2))
          (make-binding-list))
         (else
          ;; Hint at a randomly chosen alternative for y
          (make-binding-list
           (random-choice
            (map (lambda (color)
                   (make-binding y (list 'color color)))
                 (filter (lambda (z) (not (eq? z x-color))) (get-value
                                                                'colors))))))))))

;;
;; Peterson graph, uh, this is hard to do in ascii art
;;
;;      A
;;     / | \
;;    /  |  \
;;   /   |   \
;;  /    |    \
;; B---G ***** H---E
;;  \    |    /
;;   \   |   /
;;    \  |  /
;;     \ | /
;;      C-----D
;;
;; The *** section really isn't fully connected, see
;; http://goo.gl/6R0hm
;;
;; for a better idea of what it looks like

```

```

(define AB (make-basic-constraint '(A B) Node-constraint Node-hint))
(define BC (make-basic-constraint '(B C) Node-constraint Node-hint))
(define CD (make-basic-constraint '(C D) Node-constraint Node-hint))
(define DE (make-basic-constraint '(D E) Node-constraint Node-hint))
(define EA (make-basic-constraint '(E A) Node-constraint Node-hint))
(define BG (make-basic-constraint '(B G) Node-constraint Node-hint))
(define AF (make-basic-constraint '(A F) Node-constraint Node-hint))
(define HE (make-basic-constraint '(H E) Node-constraint Node-hint))
(define DJ (make-basic-constraint '(D J) Node-constraint Node-hint))
(define IC (make-basic-constraint '(I C) Node-constraint Node-hint))
(define GH (make-basic-constraint '(G H) Node-constraint Node-hint))
(define GJ (make-basic-constraint '(G J) Node-constraint Node-hint))
(define FI (make-basic-constraint '(F I) Node-constraint Node-hint))
(define FJ (make-basic-constraint '(F J) Node-constraint Node-hint))
(define IH (make-basic-constraint '(I H) Node-constraint Node-hint))

(define (coloring-constraint . edges)
  (let ((const (map apply edges)))
    (/ (apply + const) (length const))))

(define all-colored
  (make-compound-constraint
    (list AB BC CD DE EA BG AF HE DJ IC GH GJ FI FJ IH)
    coloring-constraint))

(basic-annealing-solver '(A B C D E F G H I J)
  (list all-colored) 1000)

```