Name:

Assignment # **5** Due: **7/Oct/97** 6.034 Item # **10**

Please indicate the section you were assigned to, or if you do not remember, put a tick mark on the last line.

Time	Room	Instructor	TA	Tick
10–11	34-302	Randall Davis	Brad Banks	
10–11	34-303	Tomás Lozano-Pérez	Lisa Tucker-Kellogg	
11–12	34-302	Randall Davis	Brad Banks	
11–12	36-156	Tomás Lozano-Pérez	Lisa Tucker-Kellogg	
12–1	34-301	Paul Viola	Constantine Papageorgiou	
12–1	34-302	Robert Berwick	Adrian Danieli	
1–2	24-407	Paul Viola	Constantine Papageorgiou	
1–2	34-303	Robert Berwick	Adrian Danieli	
2–3	34-303	David Cliff	Oded Maron	
3–4	34-303	David Cliff	Oded Maron	
—	—	—		

Refined Guidelines for Homework Solutions

What to Hand In

The Cover Sheet: Please detach the cover sheet from each homework handout, fill it in, and attach it *securely* to your solutions. In addition, please put your name on every page of your problem set.

Code: If you are modifying or extending existing code, you should include in your solutions only the code that you have modified or extended, not the entire program. Please clearly mark the changes or additions that you have made.

Sample Outputs: For most problems, sample outputs of your runs will be required. When the test cases are not fully specified in the homework, choose a few pertinent cases that best illustrate the full behavior of your code. Even if your algorithm is correct, we cannot give you full credit for the problem without outputs demonstrating that your code works correctly across the range of relevant cases.

Commenting Your Code

Comments are a critical part of the code that you will write. For each abstraction that you write or modify (programs, functions, data structures, rules, objects, etc.), we expect you to include an English summary of *what* it does (including input–output behavior, if appropriate); *how* it does what it does; and if there were alternative implementations, *why* you chose the one you chose.

More generally, we expect that you will place comments within the body your code to *bridge the gap* between what you are trying to do and what your code actually does. Comments are especially important to us when we assign partial credit.

Collaboration

We ask for no vow of silence; you can talk with others about the problems and learn from each other. *You can show each other your finished code, and you can ask for help with debugging if you are having difficulty.*

On the other hand, you are not to copy the work of others; everything you hand in must be your own work. *More specifically, you are expected to do your own programming. Jointly prepared programs are not acceptable.*

If you do collaborate with someone, specify the name of that person(s) and the nature of the collaboration.

Massachusetts Institute of Technology

6.034 Artificial Intelligence

Assignment # **5** Due: **7/Oct/97** 6.034 Item # **10**

In this problem set, you are asked to provide several explanations. We want BRIEF explanations; one or two sentences should do for most of them! WE WILL NOT READ OR GRADE MORE THAN 200 WORDS AS THE ANSWER TO ANY PROBLEM OR SUBPART (except for last problem). So, please say what you need to say clearly and concisely. We're generally looking for a key insight as to WHY things work rather than bulk descriptions of HOW.

Goals

This assignment is intended to familiarize you with constraint propagation/satisfaction methods. We assume you have read Chapter 12 in the text (handed out as notes). In this problem, we formulate a variety of problems as **constraint satisfaction problems** and look at different methods of solving them.

Code

The files mentioned below can be found in `/mit/6.034/ps5`. The constraint functions mentioned below can be found in `constraints.scm`. The file `backtrack-dynamic.scm` implements a version of backtracking that dynamically orders variables and values. The code for the example domains can be found in `examples.scm`. A file with some auxiliary definitions can be found in `useful.scm`. Loading the file `load.scm` will load all these files.

To do this problem set, you will need to understand the code, especially the code in `constraints.scm` and `backtrack-dynamic.scm`. You will not need to write any new code.

Note the `*verbose*` switch that controls how much is printed by several functions. It starts out `#f` but you might want it to be `#t` to better understand what is going on.

Background

In this problem set we examine the problem of finding a consistent set of assignments for a set of *variables*, $X_0, \ldots, X_{n-1}$, where each variable X_i takes its values from a finite *domain* of values D_i. The variable assignments are constrained by a set of pairwise *constraints* between some or all of the variables. The constraints are implemented by user-supplied functions and so they can be quite general.

Using this framework, `examples.scm` defines the CSP problems described below.

The line/junction labelling problem in Chapter 12 is represented as follows:

- The variables correspond to junctions, which come in four types: L, Fork, T, Arrow.
- The domain of a variable is the set of valid junction labels for a junction of that type.
- The constraints between junctions connected by a line (see below).

Another example is the *map-coloring problem* (color a map using a small number of colors so that neighboring regions have different colors):

- The variables represent regions in a map, e.g. the contiguous 48 states of the US.
- The domains of all the variables is a set of available colors.
- The constraints between the variables representing adjacent regions is that their colors be different.

Yet another example is the course scheduling problem:

- The variables are courses to be taken.
- The domain of the variables are the available "term slots" within each term e.g. (fall 1 1) (fall 1 2) etc., indicating the slots within the fall term of the first year. So (fall 1 2) refers to the second slot of the fall term of the first year. With these slots, we can enforce the constraint that no more than, say 4, courses are taken in a given term.
- The constraints between the variables are that (a) all the prerequisites of a course be taken before the course and, importantly (b), that the values assigned to the courses (the term slots) are all different!

Yet another example is the N-queens problem (place N queens on an $N \times N$ chess board so that no queen can capture another, i.e. they do not share a common row, column or diagonal).

- The variables are the rows where queens are to be placed. Each queen is identified with a column of the board, which is ok since queens cannot share columns. This representation means that we do not need to explicitely constrain the queens to avoid sharing columns.
- The domain of each variables are the available N row indices.
- The constraints between the variables are that the corresponding queens cannot capture each other, i.e., do not share a row or a diagonal.

There are many other problems that fit into this mold, e.g. other scheduling problems, crossword puzzles, molecular structure from measurements, object recognition in pictures, and others.

We can view all of these problems as defining a *constraint graph*, in which the variables are the nodes and the constraints are represented by the arcs. Note that the constraints are represented as *oriented* arcs (also known as directed edges). If we have the constraint $C(i, j)$ where i and j are variables, the constraint $C(j, i)$ is treated as a different constraint (it, in general, will be the "inverse" constraint). Note that there may be different types of arcs in such a graph, each representing a different type of constraint. For example, pre-requisite arcs (which constrain the pre-requisite of a course to a term previous to the course) and not-same? arcs (which prevent courses from occupying the same time slot). In our implementation, the graph is represented as follows:

- Variables are represented (primarily) by two vectors:
  - `*variable-names*`, is a vector whose entries hold the symbols that are the names of the variables. This is used only for producing output.
  - `*variable-domains*`, is a vector of lists, the i-th element of which is the current-list of valid values for the i-th variable.
- Constraints are encoded as:
  - a list `*arcs*`, each of whose entries is of the form `(i j C id)` where i, j are variable indices, C is a Scheme function that tests the constraint, and *id* is an arc index. The constraint function C takes 4 arguments, an index i for variable X_i, a value for that variable, an index j for variable X_j and a value for that variable, and returns `#t` when the constraint holds and `#f` otherwise. For example, in course-scheduling, the function C would be `not-same?` defined at the top of `examples.scm`.
  - a vector `*arcs-from*`, where entry i holds a list of all the members of `*arcs*` of the form `(i k C id)` for any k and C. That is, the i-th entry is the list of (directed) arcs going out from the i-th variable's node in the constraint graph.
  - a vector `*arcs-into*`, where entry k holds a list of all the members of `*arcs*` of the form `(i k C id)` for any i and C. That is, the k-th entry is the list of (directed) arcs going into the k-th variable's node in the constraint graph. This is used to speed up the implementation of `revise`.
  - a vector `*arcs-added-for-new-var*`, where entry k holds a list of all the members of `*arcs*` where the index of one of the variables is k and the other index is less than or equal to k. That is, arcs connecting the k-th variable to variables with smaller index numbers. These arcs are used to speed up the implementation of `consistent?` (see below).

We have also provided you with the functions:

- `(initialize-map-coloring map)` — initializes variables and arcs for a map-coloring problem. If no argument is given, it defaults to `*US-48-states-map*`, a map-coloring problem for the contiguous 48 states.

- `(initialize-course-scheduling courses)` — initializes variables and arcs for a course-scheduling problem. If no argument is given, it defaults to `*course-requirements*`, an example based on the Course VI MEng requirements.
- `(initialize-line-labeling diagram)` — initializes variables and arcs for a line-labeling problem. If no argument is given, it defaults to `*tower-diagram*`. If you are curious, you can see a picture of this diagram in `/mit/6.034/ps5/tower.ps` (or from our ftp site). This example is due to Peter Norvig.
- `(initialize-n-queens n)` — initializes an instance of the N-queens problem.

One of the most straightforward ways of solving a constraint satisfaction problem, without using constraint propagation, is by *backtracking search.* This is basically a depth-first search over all possible partial assignments of values to variables; a partial assignment is like a partial path. Assume we have a list of variables $(X_0, \ldots, X_{n-1})$ and their corresponding domains $(D_0, \ldots, D_{n-1})$. Then, given a partial assignment of domain values to variables, e.g.

$$(d_0, d_1, \ldots, d_{i-1}, X_i, \ldots, X_{n-1})$$

where $d_j \in D_j$, we can extend this assignment by picking a value for X_i from its domain. Such an extension is valid if it satisfies all the constraints that involve only the first $i+1$ variables. If the extension is valid, then it can be extended again, e.g. by picking a value for X_{i+1} etc. If it is not valid, one has to *backtrack*, that is, undo the most recent extension and try another value for that variable. If one runs out of values for this variable, X_i, then we have to backtrack to the previous variable, X_{i-1}, assign a new value and try again.

Backtrack implementations make use of the function `(consistent?  i values)` provided in `constraints.scm`. This function tests whether the partial assignment (a list of values of length i which represents assignments for first i variables) is valid. Note that `consistent?` assumes that the values are in *reverse* order, that is $(...d_i, d_i - 1, ..d_2, d_1)$, so as to simplify an implementation using `cons` to extend the partial assignments – like that used in `search.scm` and the recursive implementation.

The functions `constraint-propagate` and `revise` implement an *arc-consistency* enforcement algorithm known as `ac-3` (originally due to Alan Mackworth). This algorithm is an implementation (and generalization) of Waltz's constraint propagation procedure described in the text (`Chapter 12`). You should have heard about this in recitations.

Note that `consistent?` (and other functions that evaluate constraints) increment the global variables `*number-of-arc-tests*` that keeps count of how many arcs are tested. We will use these numbers as a way of comparing performance of different backtrack methods.

Problem 1

Initialize a trivial map-coloring problem by doing `(initialize-map-coloring *trivial-map*)`. You might want to look at this function and the definition of `*trivial-map*` in `examples.scm`. Now, print out the contents of the constraint graph representation by evaluating each of these expressions (`pp` is a "pretty-printer"):

```
(pp (vector->list *variable-names*))
(pp (vector->list *variable-domains*))
(pp *arcs*)
(pp (vector->list *arcs-from*))
(pp (vector->list *arcs-into*))
(pp (vector->list *arcs-added-for-new-var*))
```

Write a description of the contents of each of the variables, relate the descriptions specifically to a picture of the constraint graph of the *trivial-map* problem. Each description should be a brief sentence.

Problem 2

The following code fragments are drawn from the definition of the line-labeling examples in `examples.scm`.

```
;;; These are the legal junction labels (as in Chap 12).
;;; pos = +, neg = -, in = -<- (pointing in to junction) and out = ->-
;;; (pointing out of junction).
(define *legal-junction-labels*
  '((L (out in) (in out) (in pos) (pos out) (neg in) (out neg))
    (T (out in neg) (out in pos) (out in in) (out in out))
    (Fork (pos pos pos) (neg neg neg) (neg in out) (in out neg) (out neg in))
    (Arrow (pos pos neg) (in out pos) (neg neg pos))))

(define *cube-diagram*
  '((a Fork b c d)
    (b Arrow g e a)
    (c Arrow e f a)
    (d Arrow f g a)
    (e L c b)
    (f L d c)
    (g L b d)))

(define (line-label-constraint-function pos1 pos2)
  (lambda (junction1 label1 junction2 label2)
    (let ((symbol1 (list-ref label1 pos1))
          (symbol2 (list-ref label2 pos2)))
      (cond ((eq? symbol1 'pos) (eq? symbol2 'pos))
            ((eq? symbol1 'neg) (eq? symbol2 'neg))
            ((eq? symbol1 'in) (eq? symbol2 'out))
            ((eq? symbol1 'out) (eq? symbol2 'in))))))
```

The first fragment defines the legal labelings for each type of junction. The second fragment codes the required junction and line information for a cube. This data is processed by `diagram-names-and-domains` and `diagram-constraint-arcs` to produce the variable and arc descriptions. You should evaluate `(initialize-line-labeling *cube-diagram*)` and examine the resulting representation as you did in the previous problem, although you need not write anything about this. Instead explain briefly (2-3 sentences) how the function `line-label-constraint-function` (the third fragment above) is used to generate constraints on the arcs and why the resulting constraint functions look the way they do.

Note that the function `list-ref` accesses elements of a list by 0-base index, e.g. `(list-ref '(a b c) 1) -> b`. The function `position`, defined in `useful.scm`, is the inverse of the `list-ref` function, that is, it returns the index of an element in a list, e.g. `(position b '(a b c)) -> 1`.

Problem 3

Let's try to familiarize ourselves with the operation of the arc-consistency (constraint-propagation) code by looking at some simple examples.

Part A

Execute the following:

```
(trace revise)
(set! *colors* '(red white blue))
;; first test
(initialize-map-coloring *trivial-map*)
(propagate-constraints *arcs*)
(vector->list *variable-domains*)
;; second test
(initialize-map-coloring *trivial-map*)
(vector-set! *variable-domains* 0 '(blue))
(propagate-constraints *arcs*)
(vector->list *variable-domains*)
(vector-set! *variable-domains* 1 '(white))
(propagate-constraints *arcs*)
(vector->list *variable-domains*)
(untrace)
```

Write a brief explanation of what you see, specifically the (interesting) values in the domains and which calls to `revise` return `#t` or `#f`. Don't just repeat the output in words, explain WHY.

Part B

Explain why (in 1-2 sentences) we can get away with the following condition

```
 (not (= (arc-tail a) head-var)) ;??
```

in the definition of `propagate-constraints`

Part C

As you saw above, running `propagate-constraints` does not guarantee that there will be a single valid assignment (or even any valid asignment). Therefore, additional work is required to construct valid assignments, whether it be to find at least one, find them all or to guarantee that there are none. In particular, we can use backtracking to find one or more solutions (more on this later).

What special "trick" (additional constraint) in the line/junction-labeling domain helps explain Waltz's observation that simply running `propagate-constraint` often produces a single valid interpretation. Hint: the "trick" is described in Chapter 12 of the text (think about the legal labelings of a simple cube, for example). Explain in 2-3 sentences why this "trick" works.

Problem 4

The file `constraints.scm` has two implementations of backtracking strategies. The simpler one, called `backtrack`, is a recursive implementation of the classic backtrack strategy. Note that it relies on the use of the `consistent?` test to check the validity of a partial assignment.

The other implementation, `backtrack-fc`, combines a simple form of constraint propagation, namely `forward-check`, with backtracking.

Part A

Explain briefly (2-3 sentences) why `backtrack-fc` does not call `consistent?`.

Part B

Explain briefly (1-2 sentences) why `backtrack-fc` uses `list-domains` and `restore-domains` the way it does. Recall that we are interested in getting at the distinction between `backtrack-fc` and `backtrack`.

Problem 5

The file `backtrack-dynamic.scm` has another implementation of a backtracking strategy. This one is a generalization of `backtrack-fc`, called `backtrack-fc-dynamic`, which dynamically orders the variables and the values in the domains as the search progresses. `backtrack-fc-dynamic` takes as input a function for picking the next variable and a function for sorting the domain values of a given variable.

Part A

Briefly (1 sentence), explain the differences (if any) between the backtracking search one gets by executing

```
(backtrack-fc-dynamic pick-next domain-of)
```

and that from `(backtrack-fc)`.

Part B

Briefly (2-3 sentences), explain the key differences between the search one gets by executing

```
(backtrack-fc-dynamic
  pick-most-constrained sort-least-constrained-values-first)
```

versus the one by executing

```
(backtrack-fc-dynamic pick-next domain-of)
```

Please focus on expected impact on parameters that affect performance, such as expected branching factor and depth of the search.

Part C

Explain (1-2 sentences) why one might think that the searches in part B above (using `pick-most-constrained` and `sort-least-constrained-values-first` to order variables and values dynamically) might be more efficient than that in part A.

Problem 6

Generally one uses the number of "arc tests" (evaluation of arc constraint functions) as a machine-independent measure of how much work a particular CSP strategy does on a particular problem. Below is the number of arc tests that we get from running the three backtrack functions we have given you on some of the examples in `examples.scm`. In some cases, we ran out of patience and killed the run after some time and we report $> 10^6$ to give a bound on the number but it may be much, much bigger than reported (or maybe it was just ready to be finished when we killed it...).

Problem	backtrack	backtrack-fc	backtrack-fc-dynamic
line-labeling	125	167	469
reversed line-labeling	1418	244	512
5 color map-coloring	350	533	2,759
5 color reversed map-coloring	348	534	2,742
4 color map-coloring	$> 10^6$	$> 10^6$	$> 10^6$
4 color reversed map-coloring	$> 10^6$	$> 10^6$	$> 10^6$
course-schedule	5,188	11,074	$> 10^6$
reversed course-schedule	$> 10^6$	$> 10^6$	$> 10^6$
10 queens	3,510	1,321	3,902
20 queens	$> 10^6$	$> 10^6$	45,165
50 queens	$> 20 \times 10^6$	$> 20 \times 10^6$	1,226,851

The version of `backtrack-fc-dynamic` used for these tests used variable (`pick-most-constrained`) and value (`sort-least-constrained-values-first`) ordering.

What conclusions (about the various CSP methods and problems) do you draw from this data? Think about the merits of constraint propagation, ordering of variables and values, branching factors and degree of constraint. Also think about why some problems seem to be a lot harder than others. You don't need to write an essay, but this question is more open-ended and so we understand you might need to write a bit more than in the other questions.

By the way, if you want to run your own tests, look at functions `test-line-labeling`, `test-n-queens`, etc. in `examples.scm`. You will need calls such as:

```
(test-map-coloring backtrack (reverse *US-48-states-map*))
(test-line-labeling backtrack-fc 20)
(test-n-queens
  (lambda ()
    (backtrack-fc-dynamic
       pick-most-constrained sort-least-constrained-values-first))
  20)
```

For your edification (no questions involved), we have included an additional file `min-conflicts.scm` which implements a hill-climbing (without backup) strategy for CSP problems. This code starts with a complete assignment to the variables, e.g. a random one, and then repairs it (that is tries to reduce conflicts). It selects a variable involved in a conflict during each cycle and assigns it the value from its domain that minimizes the conflicts involving that variable (ties are broken randomly).

This strategy is remarkably effective on many problems. It can however get into infinite loops. One way of minimizing (but not abolishing) this possibility, the code keeps a weight associated with each conflict and increments it each during each cycle that the conflict is present. When picking values for a variable it weighs each conflict produced by a value choice by its cumulative weight. This way it tries to avoid persistent problems (and thereby infinite loops).

You can try this strategy with calls such as the following one.

```
(test-map-coloring
  (lambda ()
    (min-conflict-hill-climb 500 (random-assignment))
```

This runs the min-conflict strategy for at most 500 (or whatever) cycles starting from a random assignment of the variables. If you try this, you will see that it typically solves even the hardest problems we gave you with relatively few arc tests. In fact, one can do MUCH better still by being smarter about the initial assignment. The performance of this strategy is an interesting example of how giving up on guaranteed solutions can lead to finding practical solutions for many (or even most) instances of some very hard problems.