Massachusetts Institute of Technology

6.034 Artificial Intelligence

Solutions # **5** 6034 Item # **??**

Problem 1

The constraint graph for the `*cube-diagram*` problem is illustrated below:

```
              e(var 4)
            /         \
          /             \
b(var 1)                  c(var 2)
    |    \              /     |
    |      \          /       |
    |        a(var 0)         |
    |            |            |
    |            |            |
g(var 6)         |        f(var 5)
        \        |      /
          \      |    /
             d(var 3)
```

Each link represents a constraint (`line-label-constraint-function`) between adjacent junctions that share a line. Recall that the links in the diagram above are encoded as directed arcs and so we have two arcs for each link in the graph.

Part A

```
(pp (vector->list *variable-names*))
(a b c d e f g) ; variable names
(pp (vector->list *variable-domains*))
;; the domains for each var, drawn from the *legal-junction-labels*
;; < = in, > = out, + = pos, - = neg
(((+ + +) (- - -) (- < >) (< > -) (> - <))
 ((+ + -) (< > +) (- - +))
 ((+ + -) (< > +) (- - +))
 ((+ + -) (< > +) (- - +))
 ((> <) (< >) (< +) (+ >) (- <) (> -))
 ((> <) (< >) (< +) (+ >) (- <) (> -))
 ((> <) (< >) (< +) (+ >) (- <) (> -)))
(pp *arcs*)
;; there are 9 lines in the diagram above, thus 18 constraints arcs.
;; #[compound-procedure x] has been replaced by []
((0 1 [] 17) (0 2 [] 16) (0 3 [] 15) (1 6 [] 14)
```

```
 (1 4 [] 13) (1 0 [] 12) (2 4 [] 11) (2 5 [] 10)
 (2 0 [] 9) (3 5 [] 8) (3 6 [] 7) (3 0 [] 6) (4 2 [] 5)
 (4 1 [] 4) (5 3 [] 3) (5 2 [] 2) (6 1 [] 1) (6 3 [] 0))
(pp (vector->list *arcs-from*))
;; the constraint arcs emanating from each variable
(((0 3 [] 15) (0 2 [] 16) (0 1 [] 17)) ; arcs from a to d,c,b
 ((1 0 [] 12) (1 4 [] 13) (1 6 [] 14)) ; arcs from b to a,e,g
 ((2 0 [] 9) (2 5 [] 10) (2 4 [] 11)) ; etc.
 ((3 0 [] 6) (3 6 [] 7) (3 5 [] 8))
 ((4 1 [] 4) (4 2 [] 5))
 ((5 2 [] 2) (5 3 [] 3))
 ((6 3 [] 0) (6 1 [] 1)))
(pp (vector->list *arcs-into*))
;; the constraint arcs terminating at each variable
(((3 0 [] 6) (2 0 [] 9) (1 0 [] 12)) ; arcs into a from d,c,b
 ((6 1 [] 1) (4 1 [] 4) (0 1 [] 17)) ; etc.
 ((5 2 [] 2) (4 2 [] 5) (0 2 [] 16))
 ((6 3 [] 0) (5 3 [] 3) (0 3 [] 15))
 ((2 4 [] 11) (1 4 [] 13))
 ((3 5 [] 8) (2 5 [] 10))
 ((3 6 [] 7) (1 6 [] 14)))
(pp (vector->list *arcs-added-for-new-var*))
(;; constraints in choosing value for a(v0) during backtrack
 ()
 ;; constraints in choosing value for b(v1) during backtrack
 ;; ie those involving variable 0
 ((1 0 [] 12) (0 1 [] 17))
 ;; constraints in choosing value for c(v2) during backtrack
 ;; ie those involving variable 0 (since no 1-2 arcs)
 ((2 0 [] 9) (0 2 [] 16))
 ;; etc.
 ((3 0 [] 6) (0 3 [] 15))
 ((4 1 [] 4) (4 2 [] 5) (2 4 [] 11) (1 4 [] 13))
 ((5 2 [] 2) (5 3 [] 3) (3 5 [] 8) (2 5 [] 10))
 ((6 3 [] 0) (6 1 [] 1) (3 6 [] 7) (1 6 [] 14)))
```

Part B

The function `line-label-constraint-function` returns a constraint function between two junctions that share a line. The corresponding line in each junction is indicated by the `pos1` and `pos2` indices. The actual constraint checks that the labels that each junction assigns to the shared line are compatible. Compatible means equal for `pos` and `neg` labels but it actually means opposite for the `in` and `out` labels, since these directions are defined relative to the junction.

```
(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))))))
```

Problem 2

Part A

The output has been reformatted to make it easier to understand what's going on. In particular, revisions which produce no change are dropped.

```
;; first test
(initialize-line-labeling *cube-diagram*)
(propagate-constraints *arcs*)
;; the var revised is the first listed, e.g. 0 in the first call
Revising (0 1 [] 17).
;; the first entry in the label for 0 must match the last entry in the
;; label for 1.  The legal labels for 1 (Arrow) all have + or - in the
;; last entry, so we end up with labels for 0 (Fork) that whose first
;; entries match, etc.
((+ + +) (- - -) (- < >) (< > -) (> - <)) => ((+ + +) (- - -) (- < >))
Revising (0 2 [] 16).
((+ + +) (- - -) (- < >)) => ((+ + +) (- - -))
Revising (4 2 [] 5).
((> <) (< >) (< +) (+ >) (- <) (> -)) => ((> <) (+ >) (- <) (> -))
Revising (1 4 [] 13).
((+ + -) (< > +) (- - +)) => ((< > +) (- - +))
Revising (0 1 [] 17).
((+ + +) (- - -)) => ((+ + +))
Revising (2 0 [] 9).
((+ + -) (< > +) (- - +)) => ((< > +) (- - +))
Revising (4 2 [] 5).
((> <) (+ >) (- <) (> -)) => ((> <) (- <) (> -))
Revising (3 0 [] 6).
((+ + -) (< > +) (- - +)) => ((< > +) (- - +))
Revising (5 3 [] 3).
((> <) (< >) (< +) (+ >) (- <) (> -)) => ((> <) (- <) (> -))
Revising (6 1 [] 1).
((> <) (< >) (< +) (+ >) (- <) (> -)) => ((> <) (- <) (> -))
;; Note that all the domains except one remain ambiguous (more than
;; one label).  So, we have shrunk the domains with
;; propagate-constraints but not obtained a single answer.
(print-domains)
0 Dom(a)=((+ + +))
1 Dom(b)=((< > +) (- - +))
2 Dom(c)=((< > +) (- - +))
3 Dom(d)=((< > +) (- - +))
4 Dom(e)=((> <) (- <) (> -))
5 Dom(f)=((> <) (- <) (> -))
6 Dom(g)=((> <) (- <) (> -))
;; second test
(initialize-line-labeling *cube-diagram*)
;; Constrain one of the domains (of f) to a single value, we get
;; some more domain shrinking...
(vector-set! *variable-domains* 5 '((> <)))
(propagate-constraints *arcs*)
Revising (0 1 [] 17).
((+ + +) (- - -) (- < >) (< > -) (> - <)) => ((+ + +) (- - -) (- < >))
Revising (0 2 [] 16).
((+ + +) (- - -) (- < >)) => ((+ + +) (- - -))
```

```
Revising (2 5 [] 10).
((+ + -) (< > +) (- - +)) => ((< > +))
Revising (0 2 [] 16).
((+ + +) (- - -)) => ((+ + +))
Revising (1 0 [] 12).
((+ + -) (< > +) (- - +)) => ((< > +) (- - +))
Revising (3 5 [] 8).
((+ + -) (< > +) (- - +)) => ((< > +))
Revising (4 2 [] 5).
((> <) (< >) (< +) (+ >) (- <) (> -)) => ((> <) (> -))
Revising (6 1 [] 1).
((> <) (< >) (< +) (+ >) (- <) (> -)) => ((> <) (- <) (> -))
Revising (6 3 [] 0).
((> <) (- <) (> -)) => ((> <) (- <))
;; but the situtation is still ambiguous.
(print-domains)
0 Dom(a)=((+ + +))
1 Dom(b)=((< > +) (- - +))
2 Dom(c)=((< > +))
3 Dom(d)=((< > +))
4 Dom(e)=((> <) (> -))
5 Dom(f)=((> <))
6 Dom(g)=((> <) (- <))
;; Constrain another of the domains (of g) to a single value, we get
;; some more shrinking...
(vector-set! *variable-domains* 6 '((> <)))
(propagate-constraints *arcs*)
Revising (1 6 [] 14).
((< > +) (- - +)) => ((< > +))
Revising (4 1 [] 4).
((> <) (> -)) => ((> <))
;; and now the interpretation is unambiguous.
(print-domains)
0 Dom(a)=((+ + +))
1 Dom(b)=((< > +))
2 Dom(c)=((< > +))
3 Dom(d)=((< > +))
4 Dom(e)=((> <))
5 Dom(f)=((> <))
6 Dom(g)=((> <))
```

Part B

The special "trick" (additional constraint) in the line/junction-labeling domain that helps cut ambiguity substantially is that we can label the outermost lines as boundary lines. You can see this in Figures 12-15 and 12-16 of the text. This is basically saying that each figure is isolated, e.g. it is not glued to some infinite wall – which is perfectly reasonable most of the time. By pinning down the labels on these lines to a unique value, this triggers quite a bit of reduction in the neighboring domains.

Problem 3

Part A

`backtrack-fc` does not call `consistent?` because `forward-check` guarantees that any values that would cause `consistent?` to fail have been removed from the domain of the variables. Note that `consistent?`, when used by `backtrack`, fails only when a previously assigned variable (call it j), has a value inconsistent with the value being considered for the current variable (call it k). However, when `backtrack-fc` assigns a value to j, it removes conflicting values from all variables constrained by j – including k. So, when `backtrack-fc` gets to variable k all values that conflict with previously assigned values have been removed from the domain of k. By the way, its this checking of variables *forward* in time that gives `forward-check` its name. Plain backtracking always checks variables that are backwards in time.

Part B

The operation of `forward-check` is destructive, that is, it permanently modifies the domains of the variables. However, when `backtrack-fc` calls `forward-check` we only have a "tentative" assignment for the variables. We may need to backup and try a different value later. Therefore, we need to save the domains (via `list-domains`) before we run `forward-check` and then restore them (via `restore-domains`) when we fail and have to backup.

Part C

If you run the suggested test on this simple problem, you will find that the full constraint propagation does a lot more arc tests.

```
(do-test backtrack-fc) -> 42 arc tests
(do-test backtrack-pc) -> 126 arc tests
```

This observation holds in more general cases. There are two basic problems. One is simple to fix. That is simply that propagate constraints checks every arc to see if some domain will change. However, when used in backtrack, the only change to any domain since the last time propagate-constraints was called was in setting the value of one variable. So, only arcs coming into that variable need to be examined, which is what forward-check does. However, there is a more fundamental problem, which is illustrated by the trace we did above with the cube figure. If you have domains with multiple values, one tends to get little constraint propagation and one ends up still with domains having multiple values. So, generally, it is not worth doing propagation unless one starts off with an unambiguous domain, one with a single legal value. This is precisely what forward-checking always does. One could generalize forward-check a little bit and propagate further if one of the neighboring domains gets to have a single value, but we have not bothered.

Problem 4

Part A

The search performed by this call:

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

is essentially identical to the search carried out by `(backtrack-fc)`. That is, it is backtracking with forward checking using the (arbitrary) order of variables and values chosen by `initialize`.

Part B

The search carried out by:

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

versus the one by executing

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

orders the variables and the values dynamically by examining the current domains of the variables as determined by `forward-check`. The variables are ordered each time so that the variable with the fewest legal values in the domain is looked at first (most constrained variables). Note that this reduces the effective branching factor of the search tree. The values of the variables are sorted so that the values that eliminate the fewest values of the neighbors are looked at first (least-constraining values). This tends to give you the best chance of success later in the tree because it keeps your future options open.

Part C

Sorting the variables and values as above is aimed at (a) reducing the branching factor and (b) arranging the order of the links so that the likely winning paths are looked at first. Note that the ordering of values only makes sense if we are going to stop when we get the first legal answer. If we want all legal answers, we have to look at all the values anyway, so there is no point in sorting. Ordering the variables is always worthwhile, since if a branch is going to fail we'd like to find that out as early as possible.

Problem 5

A couple of observations can be seen from the somewhat incomplete results quoted in the problem set:

- Ordering variables matters – note the impact of reversing the course-scheduling problem. This affects the order in which the courses are considered in backtracking. The fact that the order the courses were typed in had the courses that needed to be taken earlier near the top made the simple backtrack methods work well. Reversing the order of the courses destroyed this property.
- Dynamic variable and value ordering can have a drastic beneficial impact. In fact, generally this is a good default strategy when a guaranteed method is required. Note that the bug in the first version of the code where it was using the least-constrained variable illustrated the negative impact of bad variable ordering.
- Some of the problems are a lot easier than others, partly as a function of how much local constraint there is. Line labeling has a lot of local constraint, but not map-coloring or course-scheduling. One way of looking at this is – how likely are we to pick the right value of variables that we look at early in the search.