EE570—Artificial Intelligence
Fall 2013
Problem Set #5
Due: Friday 25–October–2013

This problem set is an exercise in *backtracking search* and *constraint satisfaction*. It is intended to teach you how to solve search problems using nondeterministic programming. We also want you to learn when and how constraint-based techniques can improve the performance of search programs.

You will implement several methods for solving the N-Queens problem. First, you will implement a procedure that solves the N-Queens problem using backtracking search with early failure detection. Next, you will implement a procedure that solves the N-Queens problem by formulating it as a constraint-satisfaction problem. The course software includes a general constraint mechanism along with several inference procedures that implement *early failure detection (EFD)*, *forward checking (FC)*, and *value propagation (VP)*. You will implement inference procedures for *generalized forward checking (GFC)* and *arc consistency (AC)*. When solving this problem set, it may be useful to refer to the lecture slides for lectures 9, 10 and 11, available on the course Web site.

The problems in this problem set are difficult, but not overly so. They require some thought and planning, as well as fluency in nondeterministic search programming, but do not require more than a page of code for each problem. We have solved each of these problems ourselves before giving them to you. Although this problem set may be difficult, it is designed to be fun and we hope that you get hands on experience with constraint satisfaction techniques by working on this problem set. A word of advice: *Do not leave it to the last minute.*

We provide you with a substantial amount of code to be used as a basis for your solution to this problem set. In addition to a graphical user interface (GUI), we provide you with most of the uninteresting low-level data-structure–manipulation procedures. You will only have to write the high-level conceptual algorithms around the substrate that we provide. Furthermore, we give you all of the source code for the procedures that we provide. That code is in the files `QobiScheme.sc` and `ee570.sc` in the course-software distribution available on course Web page. You will not need to compile or load that code; the code has already been compiled and incorporated into the `ee570` program. It is just there in case you wish to look at it. (Hint: your procedures for generalized forward checking (GFC) and arc consistency (AC) will be very similar to the procedures for early failure detection (EFD), forward checking (FC), and value propagation (VP) that are included in `ee570.sc`.)

Problem 1: The N-Queens problem involves placing N queens on an $N \times N$ chessboard so that no two queens attack each other. We first want you to solve the N-Queens problem using backtracking search with early failure detection. We want you to implement the following procedure:

`place-n-queens-by-backtracking` N [*Procedure*]

N is a positive integer. Places N queens on an $N \times N$ chessboard so that no two queens attack each other. The solution is indicated by calling `(place-queen` i j`)` to place the N queens at appropriate positions. The return value of this procedure is ignored. This procedure should perform backtracking search with early failure detection.

For the GUI to interface correctly with your code, the code that you write must conform to the naming and calling

conventions (API) established for each problem set. There are two kinds of API specifications: procedures that you write that are called by the course software and procedures that are provided by the course software that your code must call. In problem sets 1 through 4, there were only API specifications of the first kind, i.e. procedures that you write that are called by the course software. The output of your code was passed back to the GUI by way of procedure return values. Starting in this problem set, we will have the second kind of API, i.e. procedures that are provided by the course software that your code must call. For this problem set, the course software provides the following API:

`place-queen` i j [*Procedure*]

i and j are integers between 0 and $N-1$ that indicate row and column positions respectively. Places a queen at position (i, j) on the chessboard. It updates the graphical display, drawing the queen at the specified position. The display update is done in such a fashion that the queen disappears upon backtracking. Just like local side effects are stored in the trail entry associated with the current choice point and are undone when backtracking selects the next alternative for that choice point, local graphic operations are stored in the trail entry associated with the current choice point and are also undone when backtracking selects the next alternative for that choice point.

To help debug and test your implementation, we have provided the GUI `(p5)`. After you issue the `(p5)` command, a window will be displayed with a 4×4 chessboard. You can use your code to solve the 4-Queens problem by clicking on Solve. You can change the size of the chessboard by clicking on +N or −N and then click again on the Solve button to solve the N-Queens problem for the resized chessboard. Note that you see the search process in action.

The N-Queens problem has a solution for all N except for $N = 2$ and $N = 3$. In general, finding a solution using backtracking search takes longer as N increases, though for some values of N the solution may appear more quickly. You should try solving the N-Queens problem for different values of N and watch the search process in action, so that you get a feel for how backtracking search works.

The N-Queens problem has multiple solutions for most N. Clicking on Solve will find the first solution. You can click on Next to find subsequent solutions. The GUI will inform you when there are no more solutions. When there are no more solutions, the message *No (more) solutions* will appear in the message pane at the bottom of the GUI. This message will appear if there are no solutions in the first place as well.

The GUI also provides a way to single-step through the search process. Clicking on Pause? will toggle pause mode. Pause mode is initially disabled. When pause mode is enabled, the Pause? button will be highlighted. When you start a search (by clicking on Solve or Next) in pause mode, the GUI will pause after each queen is placed and indicate this by displaying *Pause* in the status pane at the lower left-hand corner of the GUI window. When *Pause* is displayed, you can press one of the following keys:

space to continue the search and pause again after placing the next queen,

`r` to disable pause mode and continue the search until a solution is found or the search space is exhausted, or

`q` to abort the current search.

Clicking on Quit exits the GUI.

Note: don't click on the EFD, FC, VP, GFC, or AC buttons at this point. These attempt to solve the N-Queens

problem using different search strategies that you have not implemented. You will implement these in the next part of this problem set.

Problem 2: For this problem you will implement generalized forward checking (GFC) and arc consistency (AC) and use these techniques to solve the N Queens problem. First, you will need to implement the following procedures: (Note that through out this handout, and all remaining problem set handouts, the term *predicate* refers to a procedure that returns either `#t` or `#f`. Also note that throughout this handout, and all remaining problem set handouts, the terms *unary*, *binary*, and *ternary* refer to procedures that take one, two, and three arguments respectively.)

`assert-unary-constraint-gfc!` *constraint x* [*Procedure*]

Constraint must be a unary predicate, while x must be a domain variable. Asserts a unary *constraint* on x that performs generalized forward checking. Your procedure should simply restrict the domain of x to those elements that satisfy the *constraint*. This procedure should be modeled after `assert-unary-constraint-efd!` except that, unlike that procedure, it is not necessary to attach an after-demon to x. This is because once the domain of x is so restricted, the *constraint* will necessarily be satisfied and need not be checked when it is further restricted.

`assert-binary-constraint-gfc!` *constraint x y* [*Procedure*]

Constraint must be a binary predicate, while x and y must be domain variables. Asserts a binary *constraint* on x and y that performs generalized forward checking. Your procedure should attach an after-demon to x that fires only when x is bound that restricts the domain of y to those elements consistent with the binding for x. It should also attach an after-demon to y that fires only when y is bound that restricts the domain of x to those elements consistent with the binding for y. This procedure should be modeled after `assert-binary-constraint-efd!` except that, unlike that procedure, the after-demons attached to x and y are different from each other.

`assert-unary-constraint-ac!` *constraint x* [*Procedure*]

Constraint must be a unary predicate, while x must be a domain variable. Asserts a unary *constraint* on x that performs arc consistency. Your procedure should simply restrict the domain of x to those elements that satisfy the *constraint*. This procedure should be modeled after `assert-unary-constraint-efd!` except that, unlike that procedure, it is not necessary to attach an after-demon to x. This is because once the domain of x is so restricted, the *constraint* will necessarily be satisfied and need not be checked when it is further restricted.

`assert-binary-constraint-ac!` *constraint x y* [*Procedure*]

Constraint must be a binary predicate, while x and y must be domain variables. Asserts a binary *constraint* on x and y that performs arc consistency. Your procedure should attach an after-demon to x that restricts the domain of y to those elements consistent with some element in the domain of x. It should also attach an after-demon to y that restricts the domain of x to those elements consistent with some element in the domain of y. This procedure should be modeled after `assert-binary-constraint-efd!` except that, unlike that procedure, the after-demons attached to x and y are different from each other.

(Hint: your procedures for GFC and AC will be very similar to the procedures for early failure detection (EFD), forward checking (FC), and value propagation (VP) that are in `ee570.sc`.) Note that only unary and binary versions are needed for this problem set. You will write the ternary versions for the next problem set. You must create procedures with precisely the above names in order for your code to interface correctly with the code that we provide. Each procedure must take precisely the same arguments as stated above, in the same order, although you may change the names of the arguments if you choose to do so.

Having done this, you will need to implement the following procedure:

`place-n-queens-by-constraints` n [*Procedure*]

This procedure should create n domain variables, each of which have the domain `(0`$\ldots n-1$`)` representing the column positions of queens on each of the n rows. It should then assert the appropriate constraints to prevent queens from attacking each other. Such constraints should be asserted using `assert-constraint!` so that the constraint solving strategy can be selected by the GUI. (See the documentation at the end of this handout.) Finally, the procedure `csp-solution` should be called to solve the CSP just created. (Again, see the documentation at the end of this handout.)

In order for your procedure to communicate the progress of finding a solution to the GUI, it should attach an after-demon to each domain variable that it creates. This after-demon should call `place-queen` whenever the domain variable is bound. This can be accomplished with code analogous to the following:

```
(attach-after-demon!
 (lambda ()
  (when (bound? domain-variable) (place-queen i (binding domain-variable))))
 domain-variable)
```

When solving N-Queens by constraint satisfaction, the GUI provides the same functionality as when solving N-Queens by backtracking. The buttons +N, −N, Solve, Next, and Pause operate the same irrespective of whether you are using backtracking or constraint satisfaction. You inform the GUI of the search strategy to use by clicking on one of the buttons Backtracing, EFD, FC, VP, GFC, or AC. Backtracking is the initial default strategy and uses your implementation of `place-n-queens-by-backtracking`. The remainder use your implementation

of `place-n-queens-by-constraints`. The course software contains implementations of EFD, FC, and VP. If you have written `place-n-queens-by-constraints` correctly you will be able to use those strategies. Once you have written `assert-unary-constraint-gfc!` and `assert-binary-constraint-gfc!` you can select the strategy GFC and use it to solve the N Queens problem. Once you have written `assert-unary-constraint-ac!` and `assert-binary-constraint-ac!` you can select the strategy AC and use it to solve the N Queens problem.

You should try to solve the N Queens problem for different values of N using each of the available strategies to see how they perform. You should verify that your implementations of GFC and AC are indeed correct. They should admit only valid solutions and they should operate more quickly, in most cases, than the other techniques.

Good luck and have fun!

Documentation for procedures in `ee570.sc`

`domain-variable` [*Structure*]

A domain variable is a structure with three slots: `domain`, `before-demons`, and `after-demons`. The procedure

`(make-domain-variable` *domain before-demons after-demons*`)`

will make a new domain variable with the slots initialized from the corresponding arguments. The procedure `(domain-variable?` x`)` returns `#t` if x is a domain variable and `#f` otherwise. The procedures

`(domain-variable-domain` x`)`
`(domain-variable-before-demons` x`)`
`(domain-variable-after-demons` x`)`

access the appropriate slots of x, which must be a domain variable. The procedures

`(set-domain-variable-domain!` x e`)`
`(set-domain-variable-before-demons!` x e`)`
`(set-domain-variable-after-demons!` x e`)`

modify the appropriate slots of x, which must be a domain variable, to contain the new value e. The procedures

`(local-set-domain-variable-domain!` x e`)`
`(local-set-domain-variable-before-demons!` x e`)`
`(local-set-domain-variable-after-demons!` x e`)`

are like the previous procedures except that they perform a local side effect, one that is undone upon backtracking.

`*strategy*` [*Variable*]

This variable controls which constraint-processing strategy is used by the procedure `assert-constraint!`. It must be bound to one of the symbols `efd` (early failure detection), `fc` (forward checking), `vp` (value propagation), `gfc` (generalized forward checking), or `ac` (arc consistency) when calling `assert-constraint!`. The GUI will set this variable to `backtrack` to specify that a problem is to be solved using backtracking search instead of constraint-based techniques. This variable can be set to each of its allowed values by clicking on Backtrack, EFD, FC, VP, GFC, or AC in the GUI.

`create-domain-variable` *domain* [*Procedure*]

Domain must be a list. Creates a new domain variable whose domain slot is initialized to *domain*. The before-demons and after-demons slots are initialized to the empty list. Fails if *domain* is the empty list.

`attach-before-demon!` *demon x* [*Procedure*]

Demon must be a procedure of no arguments, while *x* must be a domain variable. Attaches *demon* as a before-demon to *x* by local side effect. Also runs the *demon*.

`attach-after-demon!` *demon x* [*Procedure*]

Demon must be a procedure of no arguments, while *x* must be a domain variable. Attaches *demon* as an after-demon to *x* by local side effect. Also runs the *demon*.

`restrict-domain!` *x domain* [*Procedure*]

X must be a domain variable, while *domain* must be a list that is a subset of the domain of *x*. Fails if *domain* is empty. Otherwise, if *domain* is a proper subset of the domain of *x*, the before-demons of *x* are run, the domain of *x* is set to *domain* by local side effect, and the after-demons of *x* are run. All constraint-processing procedures should use this procedure to update the domains of domain variables.

`bound?` x [*Procedure*]

X must be a domain variable. Returns `#t` if the domain of x has a single entry and `#f` otherwise.

`binding` x [*Procedure*]

X must be a bound domain variable. Returns the value of x.

`csp-solution` *domain-variables select* [*Nondeterministic Procedure*]

Domain-variables must be a list of domain variables. Finds values for all of the *domain-variables* that satisfy all of the constraints asserted between those variables. Fails if there is no solution. Otherwise, each nondeterministic solution consists of a list of values, in order, one for each of the *domain-variables*. *Select* must be a unary procedure that when applied to a list of domain variables returns one of them. *Select* is used to select the next variable to force during the backtracking search procedure and thus implements the search-ordering strategy. A typical default value for *select* would be the procedure `first` which would implement static ordering.

`assert-unary-constraint-efd!` *constraint x* [*Procedure*]

Constraint must be a unary predicate, while x must be a domain variable. Asserts a unary *constraint* on x that performs early failure detection. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-binary-constraint-efd!` *constraint x y* [*Procedure*]

Constraint must be a binary predicate, while x and y must be domain variables. Asserts a binary *constraint* on x and y that performs early failure detection. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-ternary-constraint-efd!` *constraint x y z* [*Procedure*]

Constraint must be a ternary predicate, while x, y, and z must be domain variables. Asserts a ternary *constraint* on x, y, and z that performs early failure detection. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-unary-constraint-fc!` *constraint x* [*Procedure*]

Constraint must be a unary predicate, while x must be a domain variable. Asserts a unary *constraint* on x that performs forward checking. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-binary-constraint-fc!` *constraint x y* [*Procedure*]

Constraint must be a binary predicate, while x and y must be domain variables. Asserts a binary *constraint* on x and y that performs forward checking. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-ternary-constraint-fc!` *constraint x y z* [*Procedure*]

Asserts a ternary *constraint* on x, y, and z that performs forward checking. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-unary-constraint-vp!` *constraint x* [*Procedure*]

Constraint must be a unary predicate, while x must be a domain variable. Asserts a unary *constraint* on x that performs value propagation. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-binary-constraint-vp!` *constraint x y* [*Procedure*]

Constraint must be a binary predicate, while x and y must be domain variables. Asserts a binary *constraint* on x and y that performs value propagation. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-ternary-constraint-vp!` *constraint x y z* [*Procedure*]

Asserts a ternary *constraint* on x, y, and z that performs value propagation. Users should never call this procedure directly. Call `assert-constraint!` instead.

`assert-constraint!` *constraint domain-variables* [*Procedure*]

This is the generic interface to the constraint-assertion procedures. The type of constraint is determined by the variable `*strategy*`. Asserts a *constraint* between the *domain-variables* that uses the selected strategy. The *constraint* must be a predicate whose arity equals the number of *domain-variables*. Currently, *domain-variables* must be a list of one, two, or three domain variables.

`chessboard-square` [*Structure*]

A structure representing a chessboard square. Instances of this structure have three slots. The slots `i` and `j` give the row and column positions of the chessboard square while the slot `contents` must be either the symbol `queen` or the symbol `empty`. The upper left chessboard square is given the coordinates $(0, 0)$. The procedure

```
(make-chessboard-square i j contents)
```

will make a new chessboard square with the slots initialized from the corresponding arguments. The procedure `(chessboard-square?` x`)` returns `#t` if x is a chessboard square and `#f` otherwise. The procedures

```
(chessboard-square-i x)
(chessboard-square-j x)
(chessboard-square-contents x)
```

access the appropriate slots of x, which must be a chessboard square. The procedures

```
(set-chessboard-square-i! x e)
(set-chessboard-square-j! x e)
(set-chessboard-square-contents! x e)
```

modify the appropriate slots of x, which must be a chessboard square, to contain the new value e. The procedures

```
(local-set-chessboard-square-i! x e)
(local-set-chessboard-square-j! x e)
(local-set-chessboard-square-contents! x e)
```

are like the previous procedures except that they perform a local side effect, one that is undone upon backtracking.

`*n*` [*Variable*]

The size of `*chessboard*`. Click on +N or −N to change this variable.

`*chessboard*` [*Variable*]

A vector containing the chessboard squares for the current chessboard, in row major order. This variable is set automatically when starting the GUI and whenever the size of the chessboard changes.

`redraw-chessboard-square` *chessboard-square* [*Procedure*]

Chessboard-square must be a chessboard square. Redraws the *chessboard-square* on the GUI display.

`place-queen` *i j* [*Procedure*]

I and *j* must be nonnegative integers. Informs the GUI that a queen has been placed at row *i* column *j*. The upper left chessboard square is given the coordinates $(0, 0)$. User programs must call this procedure whenever a queen is placed for the GUI to properly reflect that placement.

`make-chessboard` [*Procedure*]

Initializes `*chessboard*` to contain an `*n*` by `*n*` array of empty chessboard squares and redraws the GUI display.
