EE473—Introduction to Artificial Intelligence
Spring 2014
Problem Set #6
Due: Friday 4–April–2014

This problem set is a further exercise in *backtracking search* and *constraint satisfaction*. It is intended to teach you how to formulate different preteoretic problems within the formal framework of constraint-satisfaction problems. We also want you to learn how to judge which constraint-based techniques can improve the performance of a given pretheoretic problem.

You will implement several methods for solving crossword puzzles. Crossword puzzles are often used as benchmarks to help evaluate search heuristics. See for example, Ginsberg, Frank, Halpin, and Torrance (1990). For the first problem, you will implement a procedure for solving crossword puzzles using backtracking search and early failure detection. For the second problem you will solve the same crossword puzzles using a variety of constraint-satisfaction techniques, including *early failure detection (EFD)*, *forward checking (FC)*, *value propagation (VP)*, *generalized forward checking (GFC)*, and *arc consistency (AC)*. You will use the same constraint-satisfaction engine that you built for problem set 5. If you correctly solved problem set 5, you can use the code from that solution. Alternatively, you can use the solution provide on the course Web page. In any case, you *must* include a constraint-satisfaction engine with your handin solution to this problem set. I.e. your handin solution must be self-sufficient. We will *not* load either your or our solution to problem set 5 as part of the grading process. 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 `ee473.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 `ee473` program. It is just there in case you wish to look at it.

Problem 1: For this problem, you are given the crossword puzzles shown in figures 1, 2, and 3.[1] Unlike an ordinary crossword puzzle that is supplied with clues for each entry, you are instead given a list of words from which the entries may be filled. The task is to find words from the word list to fill the entries in a consistent fashion. Not all of the words in the word list need appear in the solved puzzle; the word list contains some "noise" words to make the problem more difficult. Additionally, some words in the word list may appear more than once in the solved crossword puzzle.

We have prepared three different lists of words that can be used to solve the puzzles. The first (called Scant) is a short list of 14 words that contains just the words that are used by Mackworth (1992) when solving Puzzle 1. The second (called Few) is a medium-length list of 147 words that contains just the union of the words needed to solve the three puzzles. The third (called Many) is a long list of over 25,000 words that contains all of the words in `/usr/dict/words`.

For this problem we want you to write a general-purpose crossword-puzzle solver that uses backtracking search and early failure detection. You should write the following nondeterministic procedure:

[1] The first puzzle was taken from Mackworth (1992). The second puzzle was constructed automatically by a Scrabble-playing program written by Carl DeMarcken. The third puzzle was taken from a crossword puzzle magazine.

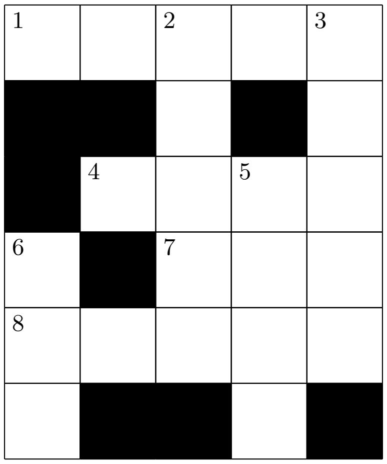

Figure 1: The crossword puzzle Puzzle 1. A representation of this crossword puzzle is contained in the variable `*puzzle1*` in `ee473.sc`.

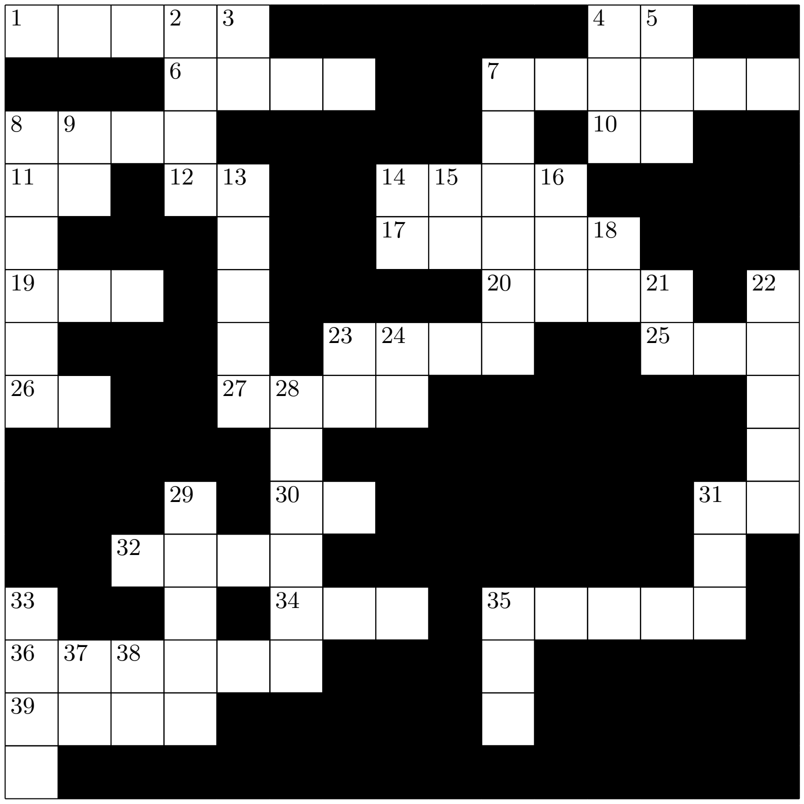

Figure 2: The crossword puzzle Puzzle 2. A representation of this crossword puzzle is contained in the variable `*puzzle2*` in `ee473.sc`.

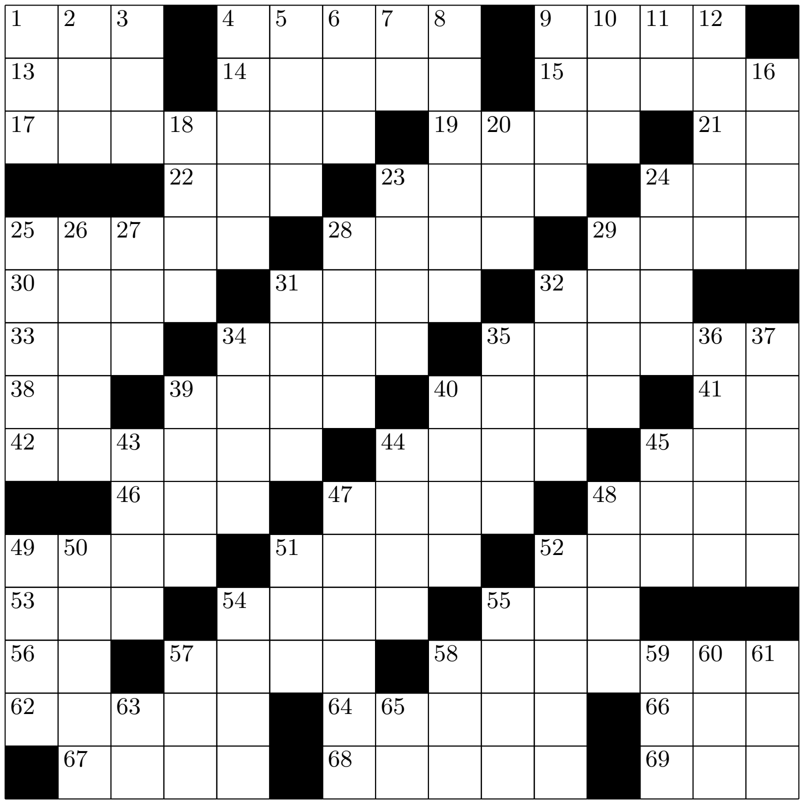

Figure 3: The crossword puzzle Puzzle 3. A representation of this crossword puzzle is contained in the variable `*puzzle3*` in `ee473.sc`.

`solve-crossword-puzzle-by-backtracking` *entries words* [*Procedure*]

Entries must be a list of entries, while *words* must be a list of strings. This procedure should implement a generate-and-test search strategy with early failure detection, *not* one that is constraint based. More specifically, it should repeatedly select the next entry to fill, nondeterministically choose a word of the appropriate length to fill that entry, and most importantly, incrementally check each new word as it is placed to make sure that it is consistent with the words with which it intersects. If you do not do such early failure detection, your program will take far too long since there are many possible ways to place words in a puzzle if one delays enforcing the requirement that intersecting words contain the same letter at their intersection until after all words are placed. (Hint: this procedure will be very similar to the `place-n-queens-by-backtracking` procedure.)

Like all of the programs you will write for this course, your program must inform the GUI of its progress. Your `solve-crossword-puzzle-by-backtracking` procedure should do so by calling `(fill-entry` *entry word*`)` each time it places a *word* in an *entry*. It would do so in a fashion analogous to the way `place-n-queens-by-backtracking` calls `(place-queen` *i j*`)`.

The file `ee473.sc` contains a number of procedures to help you write the procedure

`solve-crossword-puzzle-by-backtracking`:

```
(define (entries-intersect? entry1 entry2)
 (or (and (across-entry? entry1)
          (down-entry? entry2)
          (<= (across-entry-j entry1) (down-entry-j entry2))
          (>= (+ (across-entry-j entry1) (across-entry-length entry1) -1)
              (down-entry-j entry2))
          (<= (down-entry-i entry2) (across-entry-i entry1))
          (>= (+ (down-entry-i entry2) (down-entry-length entry2) -1)
              (across-entry-i entry1)))
     (and (down-entry? entry1)
          (across-entry? entry2)
          (<= (down-entry-i entry1) (across-entry-i entry2))
          (>= (+ (down-entry-i entry1) (down-entry-length entry1) -1)
              (across-entry-i entry2))
          (<= (across-entry-j entry2) (down-entry-j entry1))
          (>= (+ (across-entry-j entry2) (across-entry-length entry2) -1)
              (down-entry-j entry1)))))

(define (consistent-entries? entry1 entry2 word1 word2)
 (or (not (entries-intersect? entry1 entry2))
     (if (across-entry? entry1)
         (char-ci=?
          (string-ref word1 (- (down-entry-j entry2) (across-entry-j entry1)))
          (string-ref word2 (- (across-entry-i entry1) (down-entry-i entry2))))
         (char-ci=?
          (string-ref word1 (- (across-entry-i entry2) (down-entry-i entry1)))
          (string-ref
           word2 (- (down-entry-j entry1) (across-entry-j entry2)))))))
```

The procedure `entries-intersect?` determines whether two entries intersect. The procedure `consistent-entries?` determines whether `word1` can fill `entry1` at the same time that `word2` fills `entry2`.

In order to effectively solve crossword puzzles using backtracking search within a reasonable amount of computer time, one must be clever in the order one chooses to fill the entries. It makes no sense to randomly choose entries to fill since this will thwart early failure detection. Thus we suggest that you pre-sort the list of entries that is passed to `solve-crossword-puzzle-by-backtracking` so that, to the extent possible, each entry intersects some prior entry in the list. This static pre-computed ordering is often quite effective in guiding the search to solve the puzzle.

Once you have written the procedure `solve-crossword-puzzle-by-backtracking`, you can start the GUI for problem set 3 by evaluating `(p3)`. You can select a puzzle by clicking on Puzzle 1, Puzzle 2, or Puzzle 3. (We suggest that you start with the simpler puzzles first and only try the more complex ones once your program is debugged.) You can select either a word list by clicking on Scant, Few, or Many. Note that it may take a minute or so to load the Many word list from `/usr/dict/words`. (Again, we suggest that you start with the shorter word lists and only try the longer word lists once your program is debugged.) For now, leave the search strategy on Backtrack. In the next problem, you will implement the procedures necessary to solve crossword puzzles using constraint-based techniques.

Puzzle 1 has solutions with all of the word lists. You should try to solve Puzzle 1 with the Scant word list using backtracking search. This should take a few seconds. You should then try to solve Puzzle 1 with the Few word list using backtracking search. This should take less than a minute. Like before, you can click on Pause? to single-step through the solution process. Some of the puzzles have multiple solutions with some of the word lists. Like before, you can click on Next to find subsequent solutions, if there are any. Unless you are particularly clever, you will not be able to solve Puzzle 1 with the Many word list using backtracking search. There are no solution to Puzzles 2 and 3 with the Scant word list. You should be able to determine this with backtracking search. Puzzles 2 and 3 have solutions with both the Few and Many word lists. Unless you are particularly clever, you will not be able to find such solutions, using backtracking search.

The GUI for problem set 3 allows you to create your own crossword puzzles. You can adjust the size of the puzzle by clicking on +M, −M, +N, or −N. By clicking on a crossword-puzzle square you can change it from white to black and vice versa. So if you ever wish to create a crossword puzzle, say to submit to *The New York Times*, you can design the layout and use this program to select words from a large dictionary to fill the layout. Then, of course, you must write the clues for the entries, since people need such clues to solve crossword puzzles.

Problem 2: After you have successfully written the `solve-crossword-puzzle-by-backtracking` procedure (which should not be constraint based) we would like you to solve the problem a second time using GFC and again using AC. To do so you will use the general purpose GFC and AC procedures that you wrote for problem set 5. You should write the following procedure:

`solve-crossword-puzzle-by-constraints` *entries words* [*Procedure*]

Entries must be a list of entries, while *words* must be a list of strings. It should perform the same function as `solve-crossword-puzzle-by-backtracking` except that is should use constraint-based techniques. (Hint: this procedure will be very similar to the `place-n-queens-by-constraints` procedure.) This procedure should first create a domain variable for each entry. The domain of each domain variable should be the subset of the *words* of the appropriate length. The procedure should then assert the appropriate constraints between the domain variables to insure that the entries take on consistent values. 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 `fill-entry` whenever the domain variable is bound. This can be accomplished with code analogous to the following:

```
(attach-after-demon!
 (lambda ()
  (when (bound? domain-variable) (fill-entry entry (binding domain-variable))))
 domain-variable)
```

Your code should be able to solve each of the three crossword puzzles with the Scant and Few word lists using both GFC and AC. Each puzzle should take no longer than a few minutes to solve with either word list. It is possible to solve all three puzzles with the Many word list using GFC though we do not require you to do so. Puzzles 1 and 2 take a few minutes to solve with the Many word list using GFC, while Puzzle 3 takes a few hours.

Good luck and have fun!

Documentation for procedures in `ee473.sc`

`crossword-puzzle-square` [*Structure*]

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

(`make-crossword-puzzle-square` *i j contents*)

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

(`crossword-puzzle-square-i` *x*)
(`crossword-puzzle-square-j` *x*)
(`crossword-puzzle-square-contents` *x*)

access the appropriate slots of *x*, which must be a crossword-puzzle square. The procedures

(`set-crossword-puzzle-square-i!` *x e*)
(`set-crossword-puzzle-square-j!` *x e*)
(`set-crossword-puzzle-square-contents!` *x e*)

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

(`local-set-crossword-puzzle-square-i!` *x e*)
(`local-set-crossword-puzzle-square-j!` *x e*)
(`local-set-crossword-puzzle-square-contents!` *x e*)

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

`across-entry` [*Structure*]

A structure representing a crossword-puzzle across entry. Instances of this structure have three slots. The slots `i` and `j` give the row and column positions of the first character of the entry while the slot `length` gives the length of the entry. The upper left crossword-puzzle square is given the coordinates $(0, 0)$. The procedure

```
(make-across-entry i j length)
```

will make a new across entry with the slots initialized from the corresponding arguments. The procedure

```
(across-entry? x)
```

returns `#t` if x is an across entry and `#f` otherwise. The procedures

```
(across-entry-i x)
(across-entry-j x)
(across-entry-length x)
```

access the appropriate slots of x, which must be an across entry. The procedures

```
(set-across-entry-i! x e)
(set-across-entry-j! x e)
(set-across-entry-length! x e)
```

modify the appropriate slots of x, which must be an across entry, to contain the new value e. The procedures

```
(local-set-across-entry-i! x e)
(local-set-across-entry-j! x e)
(local-set-across-entry-length! x e)
```

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

`down-entry` [*Structure*]

A structure representing a crossword-puzzle down entry. Instances of this structure have three slots. The slots `i` and `j` give the row and column positions of the first character of the entry while the slot `length` gives the length of the entry. The upper left crossword-puzzle square is given the coordinates $(0, 0)$. The procedure

```
(make-down-entry i j length)
```

will make a new down entry with the slots initialized from the corresponding arguments. The procedure

`(down-entry?` *x*`)`

returns `#t` if *x* is a down entry and `#f` otherwise. The procedures

```
(down-entry-i x)
(down-entry-j x)
(down-entry-length x)
```

access the appropriate slots of *x*, which must be a down entry. The procedures

```
(set-down-entry-i! x e)
(set-down-entry-j! x e)
(set-down-entry-length! x e)
```

modify the appropriate slots of *x*, which must be a down entry, to contain the new value *e*. The procedures

```
(local-set-down-entry-i! x e)
(local-set-down-entry-j! x e)
(local-set-down-entry-length! x e)
```

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

`*m*` [*Variable*]

The number of rows in `*crossword-puzzle*`. Click on +M or −M to change this variable.

`*n*` [*Variable*]

The number of columns in `*crossword-puzzle*`. Click on +N or −N to change this variable.

`*how-many-words*` [*Variable*]

Either the symbol `scant`, indicating that `*words*` contains the Scant word list, `few`, indicating that `*words*` contains the Few word list, or the symbol `many`, indicating that `*words*` contains the Many word list. This variable is used only by the GUI to control which button is highlighted.

`*words*` [*Variable*]

The list of the words to use when solving a crossword puzzle. Click on Scant, Few, or Many to set this variable to the indicated word list.

`*crossword-puzzle*` [*Variable*]

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

`*puzzle1*` [*Variable*]

A representation of Puzzle 1 from figure 1.

`*puzzle2*` [*Variable*]

A representation of Puzzle 2 from figure 2.

`*puzzle3*` [*Variable*]

A representation of Puzzle 3 from figure 3.

`scant-words` [*Procedure*]

Sets `*words*` to the Scant word list.

`few-words` [*Procedure*]

Sets `*words*` to the Few word list.

`many-words` [*Procedure*]

Sets `*words*` to the Many word list.

`redraw-crossword-puzzle-square` *crossword-puzzle-square* [*Procedure*]

Crossword-puzzle-square must be a crossword-puzzle square. Redraws the *crossword-puzzle-square* on the GUI display.

`crossword-puzzle-square-region` *crossword-puzzle-square* [*Procedure*]

Crossword-puzzle-square must be a crossword-puzzle square. Enables a mouse-sensitive region for the *crossword-puzzle-square* on the GUI display, so that clicking on the crossword-puzzle square toggles it from white to black and from black to white.

`place-letter` *i j c* [*Procedure*]

I and *j* must be positive integers, while *c* must be a character. Informs the GUI that *c* has been placed at row *i* column *j*. User programs should not call this procedure directly but should rather call `fill-entry` to inform the GUI when an entry has been filled.

`fill-entry` *entry word* [*Procedure*]

Entry must be an entry, while *word* must be a string. Informs the GUI that the string *word* has been placed in *entry*. User programs must call this procedure whenever an entry is filled for the GUI to properly reflect that placement.

`make-crossword-puzzle` [*Procedure*]

Initializes `*crossword-puzzle*` to contain an `*m*` by `*n*` array of empty crossword-puzzle squares and redraws the GUI display.

`make-standard-crossword-puzzle` *puzzle* [*Procedure*]

Puzzle must be a representation of a crossword puzzle in the format typified by `*puzzle1*`. Sets `*m*` and and `*n*` to the dimensions of the *puzzle*, sets `*crossword-puzzle*` to contain the appropriate crossword-puzzle squares with the appropriate empty or black contents, and redraws the GUI display.

`across-entries` [*Procedure*]

Returns a list of all of the across entries in `*crossword-puzzle*`.

`down-entries` [*Procedure*]

Returns a list of all of the down entries in `*crossword-puzzle*`.

`entries-intersect?` *entry1 entry2* [*Procedure*]

Entry1 and *entry2* must be entries. Returns `#t` if the two entries intersect and `#f` otherwise.

`consistent-entries?` *entry1 entry2 word1 word2* [*Procedure*]

Entry1 and *entry2* must be entries, while *word1* and *word2* must be strings. Returns `#t` if *word1* and *word2* can consistently fill *entry1* and *entry2* respectively. Otherwise returns `#f`.

`entry-length` *entry* [*Procedure*]

Entry must be an entry. Returns the length of *entry*.

References

Ginsberg, M. L., Frank, M., Halpin, M. P., & Torrance, M. S. (1990). Search Lessons Learned from Crossword Puzzles. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pp. 210–5, Boston, MA.

Mackworth, A. K. (1992). Constraint Satisfaction. In S. C. Shapiro (Ed.), *Encyclopedia of Artificial Intelligence* (second edition)., pp. 285–93. New York, NY: John Wiley & Sons, Inc.