Declarative Programming

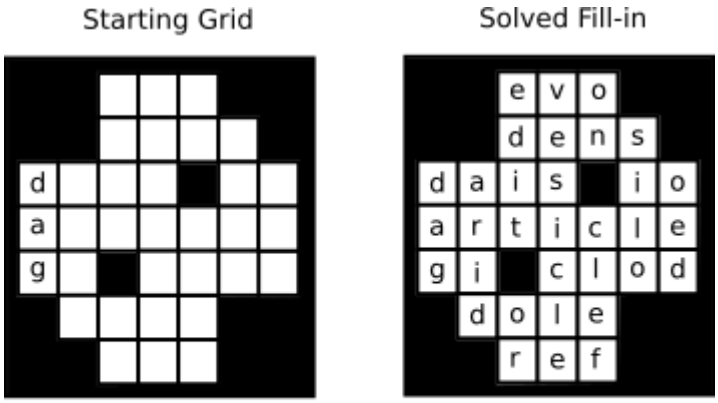
Project 1: Fill in puzzles

A fill-in puzzle (sometimes called a fill-it-in) is like a crossword puzzle, except that instead of being given obscure clues telling us which words go where, you are given a list of all the words to place in the puzzle, but not told where they go.

The puzzle consists of a grid of squares, most of which are empty, into which letters or digits are to be written, but some of which are filled in solid, and are not to be written in. You are also given a list of words to place in the puzzle.

You must place each word in the word list exactly once in the puzzle, either left-to-right or top-to-bottom, filling a maximal sequence of empty squares. Also, every maximal sequence of non-solid squares that is more than one square long must have one word from the word list written in it. Many words cross one another, so many of the letters in a horizontal word will also be a letter in a vertical word. For a properly constructed fill-in puzzle, there will be only one way to fill in the words (in some cases, the puzzle is symmetrical around a diagonal axis, in which case there will be two symmetrical solutions).

Here is an example 7 by 7 fill-in puzzle, together with its solution, taken from https://en.wikipedia.org/wiki/Fill-In_(puzzle) . In this example, one word is already written into the puzzle, but this is not required.



Word List: gi, io, on, or, dag, evo, oed, ref, arid, clef, clod, dais, dens, dole, edit, silo, article, vesicle

You will write Prolog code to solve fill-in puzzles. Your program should supply a predicate **puzzle_solution(Puzzle, WordList)** that holds when **Puzzle** is the representation of a solved fill-in puzzle for the given list of words, **WordList**.

A fill-in puzzle will be represented as a list of lists, each of the same length and each representing a single row of the puzzle. Each element in each of these lists is either a: '#', denoting a solid, unfillable square; an underscore (_), representing a fillable square; or a single, lower case letter (e.g., h), denoting a pre-filled square.

For example, suppose you have a 3 by 3 puzzle with the four corners filled in solid and one pre-filled letter. This would be represented by the Puzzle argument:

?- Puzzle = [['#',h,'#'],[_,_,_],['#',_,'#']]

A word list will be represented as a list of lists. Each list is a list of characters, spelling a word. For the above puzzle, the accompanying word list may be:

?- WordList = [[h,a,t], [b,a,g]]

You can assume that when your **puzzle_solution/2** predicate is called, both arguments will be a proper list of proper lists, and its second argument will be ground. **You may assume your code will only be tested with proper puzzles, which have at most one solution.** Of course, if the puzzle is not solvable, the predicate should fail, and it should never succeed with a puzzle argument that is not a valid solution. For example, your program would solve the above puzzle as below:

?- Puzzle = [['#',h,'#'],[_,_,_],['#',_,'#']], WordList = [[h,a,t], [b,a,g]], puzzle_solution(Puzzle, WordList).

Puzzle = [[#, h, #], [b, a, g], [#, t, #]],

WordList = [[h, a, t], [b, a, g]] ;

false.

Your **puzzle_solution/2** predicate, and all supporting code, should be written in the file **proj2.pl**. You may also use Prolog library modules supported by SWI Prolog as installed on the server, which is version 7.2.3.

Hints

1. The basic strategy for solving these puzzles is to select a *slot* (*i.e.*, a maximal horizontal or vertical sequence of fill-able and pre-filled squares), and a word of the same length from the word list, fill the word into the slot in the puzzle and remove the word from the word list. It is important that the word selected should agree with any letters that are already filled into the slot. Repeat this process until the puzzle is completely solved and the word list is empty.

2. To avoid writing much code to handle filling slots vertically, you can just transpose the puzzle, use operations to handle horizontal puzzle slots, and then transpose the puzzle back. It is easier to detect and handle horizontal slots than vertical ones.

The SWI Prolog library provides two different, incompatible**transpose/2** predicates, and unfortunately autoloads the wrong one by default. If you wish to use the correct one, you should put the line:

:- ensure_loaded(library(clpfd))

in your source file. This defines the predicate **transpose(Matrix0, Matrix)** that holds when **Matrix0** and **Matrix** are lists of lists, and the *columns* of each are the *rows* of the other.

3. One of Prolog's features, known as the *logical variable* can make this job easier. You can construct a list of *slots* where each slot is a list of Prolog variables representing squares in the puzzle. If you make sure that the same variable is used for the same square, whether it appears in a vertical or horizontal slot, then Prolog will ensure that the letter placed in the box for a horizontal word is the same as for an intersecting vertical word. For example, suppose you have a 3 by 3 puzzle with the four corners filled in solid. This would be represented as the puzzle:

Puzzle = [['#',_,'#'],[_,_,_],['#',_,'#']]

expressed diagrammatically as: # _ #

_ _ _

_

4. The slots in this puzzle could be represented as [[A,B,C], [X,B,Z]], in which case the filled-in puzzle will be Filled = [['#',X,'#'], [A,B,C], ['#',Z,'#']]. Then unifying [X,B,Z] with [c,a,t] will bind B to *a* in both lists, ensuring that the first list is only unified with a word that has an *a* as second letter.

Note that after the second list is unified with [c,a,t], Filled will become [['#',c,'#'], [A,a,C], ['#',t,'#']]. Once the first list is unified, binding A and C, this will be the final solved puzzle.

With this approach, the list of slots is simply a permutation of the list of words. For small puzzles, it is sufficient to backtrack over permutations of the wordlist until it can be unified with the slot list.

5. As an alternative to Hint 4, you may instead approach the problem by repeatedly transforming the puzzle by filling in slots in the puzzle. With this approach, the 3 by 3 puzzle discussed above would be represented as:

[['#',_,'#'], [_,_,_], ['#',_,'#']]

Filling in *cat* in the vertical slot would leave:
[['#',c,'#'], [_,a,_], ['#',t,'#']]

(again, it is easiest to work with horizontal slots; a vertical slot can be handled by transposing the puzzle, handling a horizontal slot, and then transposing again). As words are filled in, some of the letters of some slots will be filled in. Care must be taken when filling partially-filled slots to ensure the word being placed matches the letters already placed in that slot. Care must also be taken that a slot that has had all its letters filled by filling perpendicular words is in fact a word, and that word is removed from the word list.
This approach is simpler to understand than that of Hint 4 but requires considerably more code and more work.

6. The number of permutations of a list of length *n* is *n!*. Even a 7 by 7 puzzle may have 18 words, which has *18! > 10^15* permutations. Careful use of Prolog can substantially reduce this search space through the use of fast failure. That is, as each word is selected for its slot, it should be unified immediately with the slot. If this fails, then all further permutations involving that word in that slot can be discarded (Prolog will automatically do that).

7. Hint 6 will allow medium-size puzzles to be solved. For larger puzzles, the power of the factorial is too great. To solve those, it is necessary to avoid more of the search space. This can be done by carefully choosing the order in which to place words in slots. For example, if there is only one 6-letter word, then place that first. Once that word is placed, some letters, in other words, will be filled in; perhaps one of them will have only one matching word, and so on. Each time a word is to be placed, you should count the number of words that match each slot, and select (one of) the slot(s) with the fewest matching words to fill. This minimizes the search space.

8. For some puzzles, as they are being filled in, there is always at least one slot that has only one word that can possibly fit. These puzzles are easier to solve because no search is actually required. 30% of the test cases will be like this.

Assessment

Your project will be assessed on the following criteria:

30% Quality of your code and documentation;

30% The ability of your program to correctly solve fill-in puzzles of varying sizes (up to 15 by 15) that can be solved without search (see Hint 8);

20% The ability of your program to correctly solve small to medium size puzzles (up to 10 by 10) that do require search; and

20% The ability of your program to correctly solve larger puzzles (up to 32 by 20) that do require search.

Note that timeouts will be imposed on all tests. You will have at least 20 seconds to solve each puzzle, regardless of difficulty. Executions taking longer than that will be unceremoniously terminated, leading to that test being assessed as failing. Twenty seconds should be ample with careful implementation.

See the Project Coding Guidelines on the LMS for detailed suggestions for coding style. These guidelines will form the basis of the quality assessment of your code and documentation.

Late submissions will incur a penalty of 0.5\% per hour late, including evening and weekend hours. This means that a perfect project that is much more than 4 days late will receive less than half the marks for the project. If you have a medical or similar compelling reason for being late, you should contact the lecturer as early as possible to ask for an extension (preferably before the due date).