**Artificial Intelligence (CS 131)**
**Assignment 02: Constraint Satisfaction Solver (80–100 points)**

*Due Tuesday, 05 July, 11:59 PM*

---

[ **1** ] (*75 points*) This exercise involves coding a constraint satisfaction problem (CSP) solver. Your program needs to run from the command line and take **two strings** as input, each naming a text-file. This first file will describe a crossword puzzle grid; the second will contain a list of words. As an example, if written into a Python script called `solver.py`, we might execute it using:

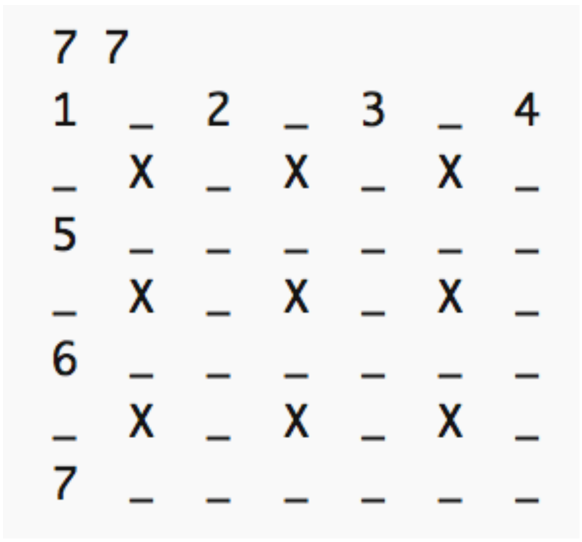
```
python solver.py xword00.txt dictionary_small.txt
```

Your program will use CSP techniques to attempt to find a combination of words from the dictionary that will fill in the crossword grid, where a complete solution satisfies the conditions:

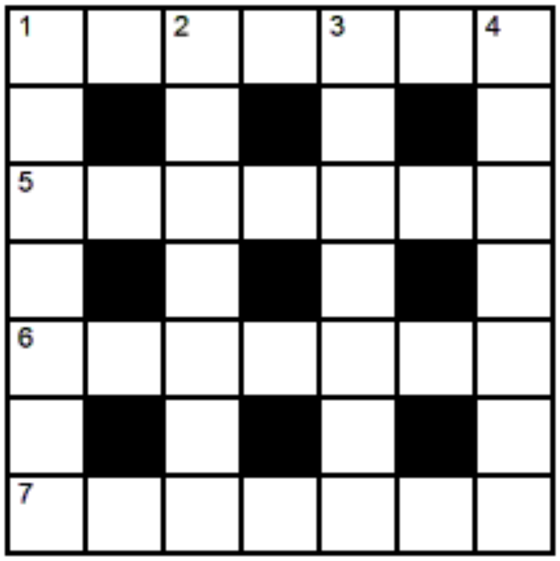
- **a.** Words begin at numbered squares, and fill up all blanks leading across or down from that number until encountering the edge of the puzzle or a black square.
- **b.** Across words always have a puzzle edge or a black square to the **left of their number**; down words always have a puzzle edge or a black square **above their number**.
- **c.** If two words intersect at a given blank square, their letters must match.
- **d.** Words *may* be repeated in a valid filling of the grid.

A puzzle specification file will begin with two integers specifying the number of rows and columns in the puzzle respectively. It will then have a corresponding grid of numbers (which mark the start of a word), as well as symbols for blank squares (the underscore, _) and black squares (the `X`).

For example, the puzzle file `xword01.txt` and the puzzle grid it represents look like:



```
7 7
1 _ 2 _ 3 _ 4
_ X _ X _ X _
5 _ _ _ _ _ _
_ X _ X _ X _
6 _ _ _ _ _ _
_ X _ X _ X _
7 _ _ _ _ _ _
```



When run, your program will perform **backtracking CSP search**, as given by the pseudo-code in Russell& Norvig, Figure 6.5, page 192) to solve the problem.

**a.** It will first read in the puzzle file and dictionary file.

**b.** Based on the puzzle file, it will compute the number of variables (blank words going across or down) in the problem, and print out this value.

**c.** It will compute the set of constraints on the variables—there will be one constraint for every position in the grid at which two words intersect.

**d.** For each word-variable, it will compute the possible domain of values (words of the right length) from the dictionary file. Initially, each variable will have no value. The program will print out a list of the variables (given in the form $m$`-across` or $n$`-down` for appropriate values of $m$, $n$), along with the size of that variable's domain.

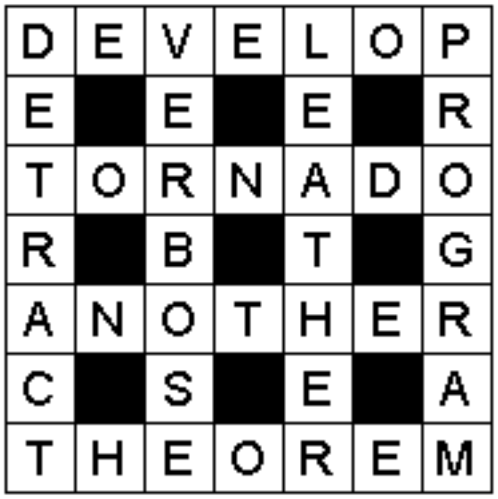
**e.** It will employ recursive backtracking search:

- When choosing a variable (Select-Unassigned-Variable), use the ***Most-Constrained Variable*** heuristic; break ties using the ***Most Constraining Variable*** heuristic, as described in lecture 07, slides 5–6. Variable values can be ordered however you see fit.
- The Inference step will be left out of the algorithm, making it a little simpler.

**f.** Search will terminate when either (1) a variable assignment is found that satisfies all constraints or (2) no such solution is found, and search fails.

- When search fails, the program will indicate that no solution was found, and print out the number of distinct calls to the recursive search method that were made.
- If search is successful, then it will also print out how many recursive calls were required, and then print out the solution found (using spaces for black squares in the grid).

For example, if we ran the CSP solver on the `xword01.txt` file given above, along with the file `dictionary_small.txt`, a possible solution to the problem is:



In generating the solution just given, our program output will look like the following:

```
%python solver.py xword01.txt dictionary_small.txt

8 words
16 constraints

Initial assignment and domain sizes:
1-across = NO_VALUE (8 values possible)
1-down = NO_VALUE (8 values possible)
2-down = NO_VALUE (8 values possible)
3-down = NO_VALUE (8 values possible)
4-down = NO_VALUE (8 values possible)
5-across = NO_VALUE (8 values possible)
6-across = NO_VALUE (8 values possible)
7-across = NO_VALUE (8 values possible)

SUCCESS! Solution found after 648 recursive calls to search.

DEVELOP
E E E R
TORNADO
R B T G
ANOTHER
C S E A
THEOREM
```

When you have finished encoding the solution, your code should be able to solve the smallest puzzles in most cases; run tests of this as follows:

**a.** Test the simplest puzzle grid (`xword00`) against all three of the dictionaries. A proper implementation will solve the puzzle in all three cases very quickly.

**b.** Test the second-simplest puzzle grid (`xword01`) against the `small` and `medium` dictionaries. A proper implementation will solve the puzzle in the `small` case very quickly, while the `medium` case may take several seconds.

***Optional***: Test this puzzle grid against the `large` dictionary. This can be solved, but it may take a few minutes.

**c.** Test the more complex puzzle grid (`xword02`) against the `small` dictionary. The algorithm should very quickly determine that the problem can't be solved, as the dictionaries lack the words necessary to fill the grid. (Trying to solve this puzzle on either of the larger dictionaries is unlikely to succeed in any reasonable amount of time.)

[ **2** ] (*20 points*) ***Required of any Tufts graduate students; optional for others.***

As described in the text and in class, we can use the method of **arc-consistency** as a pre-processor for solving a CSP, to reduce the number of actual values allowed for the variables in the problem. You will implement the `AC-3` algorithm given in the text (Figure 6.3, page 187), and run it on the CSP problem (after step **??**, page **??** of this document) before doing search.

When you do so, you will amend your code so that it can run both the version without pre-processing (as before) and a version where pre-processing is performed. The code should run from the command line as before, and the first two arguments will be the same as before (i.e., the two file-names). If a third argument is provided, of any sort, then pre-processing should be done; that is, ***any*** of the following commands would invoke the pre-processor:

```
python solver.py xword01.txt dictionary_small.txt true
python solver.py xword01.txt dictionary_small.txt 1
python solver.py xword01.txt dictionary_small.txt bananagram
```

In addition, if the pre-processing option is chosen, then the code should print out the size of each variable's domain both *before and after* running the arc-consistency check, and then proceed to do the search on the reduced domain. (See the sample runs included with the assignment specification for an example of how this could look.)

If properly implemented, the search should then proceed very quickly to a solution or failure state, for any combination of puzzle and dictionary files. Test this.

---

You will hand in source-code for the program. The instructor will compile it and test it by giving it different input files. All sample files are provided, and you can create your own for testing, but you need to use the same format as given above. I prefer that you code the solution in Python or C++, but if you wish to use another language, ensure that it is something that course staff will easily be able to compile (if necessary) and run. When doing your coding, follow these conventions:

1. Provide a short `README` file with your code, explaining exactly what commands or procedures are needed to make it compile and run properly.

2. Follow basic software engineering principles; that is, your code should be clean and well-structured, with comments explaining each method or function, and the usual format conventions to make it readable.