# Contents

# Emogic's Crossword Generation Algorithm

See "Emogics_Crossword_Algorithm.rtf" for a better formatted version of this text.

Note: My Javascript version is the most up to date. While porting my PERL code to JS, I have found many logical errors in my algorithms. I have tried to fix them. I believe my JS version to be the preferred version. Time permitting I may update my PERL code with the fixes.

Both versions are intended more as a tool to study and test a few algorithms that can generate crossword puzzles and the differences between them.
It is not intended to generate New York Times quality puzzles.

See: "Design and Implementation of Crossword Compilation Programs Using Serial Approaches" (CCP). It was at http://thesis.cambon.dk/. But that site no longer exists. The author has granted permission for me to post it on my github. See "thesis.cambon.dk.pdf". I have used many of the concepts mentioned in his thesis and in some cases expanded on them. If you want to create your own crossword generator script,it is a great place to start. The thesis has sample code in C.

It has been suggested that a letter by letter search is as good (or equivalent) to a word by word search. In a logical sense it is, but not in a practical sense. Yes, a letter search can mimic a word search. But letter searches take longer to discover dead ends in the search space, compared to full word searches. So unless your letter search is mimicking a word search, there will likely be many more CPU cycles burned on a letter search.
Update: I have noticed that the letter by letter search is very fast in JS and in many cases out performs the word searches. On PERL this is not the case. I believe this is due to PERL having highly optimized regexp code. My word searches rely on the regexp code as it provides amazing memory savings and makes the code much simpler. To make the word search routine as fast as a letter search routine, I would need more RAM than a standard PC has.
However as the puzzle sizes increase there is a point when the word search outperforms the letter search. As the puzzle size increases the search tree grows much faster for the letter search than for the word search. So even though the letter search algorithm is much faster, eventually a word search can outperform the letter search.

# The Code

My Crossword Source Code is at:
PERL: https://github.com/vpelss/crosswords
JS: https://github.com/vpelss/crosswords_js

You can download the code for my old British style Crossword Generator at:
https://www.emogic.com/store/free_crossword_script.
It is very simplistic.

My code is commented. However, I make no apologies if it makes no sense to you.

## Why the PERL version?

PERL is not known for its speed. It also has a higher memory overhead for variable management. But it does have a few benefits. Regular expressions are powerful and fast. Hash variables (associative array) allow for an easy way to eliminate duplicate values and make it easy to search and look-up values.

PERL is ubiquitous. PERL is elegant. PERL is, in my opinion, organic. I just like PERL

## What, a Javascript Version Too?

Why a JS version? Well it is ubiquitous also.

One benefit of the JS version is you can run and wait for the code to complete as long as you like. The perl version is likely to timeout and stall on long runs.

With the JS version, if you open up the browser's Inspect console, you will be able to set breakpoints and single-step through the code and see the variable values, etc.... I also have provided an option to turn on some descriptive output to the console, but you will want to turn that off if you are looking for speed.

I also wanted to compare the speed with my Perl version.

Converting my Perl code to JS also gave me a chance to revisit my code to see if it could be improved or modified.

JS associative arrays are not as easy to use and flexible as the Perl hash (for JS nested arrays you need to manually create each nest level) . So unless I had to use an associative array, I converted most of the data structures to simple arrays.

Note that the JS version downloads the database from my site, the files are not stored locally.

# Word Lists

The word lists are not included.
A wonderful one that I had found was at  www.otsys.com/clue (now: https://tiwwdty.com/clue/).
There was a plain text database there in the past, but it is no longer provided.
You could petition the author, Matt Ginsburg, if he is still a member at:
https://www.cruciverb.com/

Cruciverb is a great source for information also. The community is very friendly.

## Dictionary and Clues Database Structure

I am using flat file text databases for simplicity and speed. The dictionary data we need is loaded upfront and stored in ram for quick access. After the puzzle is created, we load the clues database files as required. Speed is not required for the few clue lookups.

For huge word lists (required for big grids) a combined dictionary and clue database list is just too large.

A single database of all words and clues is very large and takes too much time to load and separate the data logically.

If we want our crossword algorithm to have quickest access to the database must be loaded into RAM. But there are very few systems with that much RAM.

I settled on creating text files based on word lengths. 3.txt contains all 3 letter words in the dictionary. 7.txt contains all the 7 letter words.

After the crossword grid is loaded, only the required text files are loaded into their own comma delimited string variable for word searches. I use regular expression searches to search for word patterns to get the words that will fit. For letter searches we build an associative array of masks as keys and next possible letters as values.

Once the words are all placed in the puzzle and we need the clues for those words. A quick way to load clues would be to have a word.txt file for each word containing all the possible clues.

jude.txt would contain:
Name in a Beatles song
Law of "Sleuth"
New Testament book
Saintly Thaddaeus
...

This would quickly fill up all the available disk space as 250,000+ small text files take up much more disk space than the data they contain.

To save disk space I create clue text files based on the words first two letters no matter what the word size.
This takes 250,000 files and converts them into 650 files. It takes a little longer to load the clues, but it is not noticeable.

_ae.txt contains:

```
AEA|Thespians' org.
AEA|Thespians' org.
AEA|Thespians gp.
AEACUS|Grandfather of Achilles
AEAEA|Circe's island
...
```

# Crossword Puzzle Grid Template Design

I chose a simple text design.

The grid templates are text files containing rows made up of 'o' and 'x'.
'x' = black squares/cells
'o' = empty squares/cells
When looking at the puzzle we are likely to think in terms of ‘squares’, and when talking of code we are more likely to use the term ‘cells’.
The x and o were chosen as they are the same size and this gives an easily visual representation of the actual grid in notepad.

In my code, the puzzle variable retains these x and o values
x = a back square or pad (I use the term pad in my code)
o = a empty white square or unoccupied
A capital letter is used when we fill a square.
The main puzzle array is in the form of puzzle[y][x] = ‘T’

eg:

```
xooooo
ooooxo
ooooxo
```

# Global Data Structures

Our puzzle data structures use both word storage structures and letter storage structures.
Both are used at the same time. Both can be beneficial in different situations.

var puzzle : letter centric
the puzzle with the words inserted.
puzzle[y][x] = letter

all_masks_on_board : word centric
All words on board whether complete or not. Letters not known are saved as ‘o’
all_masks_on_board [dir 0=across 1=down][word number]

letter_positions_of_word and this_square_belongs_to_word_number map word numbers to cell positions and vice versa.

letter_positions_of_word[dir][word_number] = [[x0,y0],[x1,y1]...]
Can be used to find all crossing words fast with this_square_belongs_to_word_number and using the opposite dir

this_square_belongs_to_word_number[dir][y][x] returns the word number this square belongs to
It can be used with letter_positions_of_word to find a words crossing words

position_in_word[dir][y][x] returns the position of letter in the word this square belongs to, starting at 0

words_that_are_inserted[word] = true
Used to prevent duplicates words on the board

most_frequent_letters
Not used: letter by letter search
used as most_frequent_letters[letter_pos][word_length] = [E, R , J , …]
contains an ordered list of the possible letters at a position in a word, and ordered from less frequent to most frequent

letter_frequency = (E,T,A,O,I,N,S,R,H,D,L,U,C,M,F,Y,W,G,P,B,V,K,X,Q,J,Z)
Not used: letter by letter search
A list of the most common letters first to last

var pad_char = 'x'

var unoccupied  = 'o'

Ordering of associative array keys:
In the associative arrays with multiple layers, the keys were set in the following order.
dir , y , x
dir , word_number
This order provides a better experience when troubleshooting and trying to examine the associative array’s values.

# Recursion

Initially I tried to create my algorithm using loops. As you can imagine the code got very confusing very quickly. Crossword algorithms are more suited to recursive routines.

The general recursion steps are:

```
RecursiveFunction(){
    are we at the end of our walk? if so return true (puzzle is full. we are done. recurse all the way back true)
    get next cell or word location in out walk
    get a list of possible letters or words that fit here
    success = false; //be a pessimist
    while(success == 0){
    pick a letter or word that from the list of possible letter words
    are we out of letters/words than can go at this location? if so backtrack : return false
    add it to our puzzle
    success = RecursiveFunction();
    //if success == false then we will do one while loop trying the next possible letter or word at this location
    if success == true : return true
    }
return true;
}
```

# Letter Searches

## Letter Walks

Walks: The order we try to fill in the cells.

See "thesis.cambon.dk.pdf" for most of the letter walks I used. All walks assume that words will be filled from the first letter of the word to the last letter in order.

Our walk order is stored in the global array next_letter_positions_on_board.

Data structure:
var next_letter_positions_on_board = [ [x0,y0] , [x1,y1] , [x2,y2] , … ]
An array of cell arrays containing (x , y) puzzle locations in the order we want to fill in the puzzle. It is a global variable so our recursive routine can quickly shift and unshift (on backtracks) cell locations.

## Letter Backtracks Naive

Naive backtracks are simpler to code. When you cannot place a letter in a cell, go back to the previous cell that sent you there.

## Letter Backtracks : Optimal

See "thesis.cambon.dk.pdf".

Optimal backtrack can be dependent on the walk we have chosen and the grid structures around the failed letter.

I have attempted to generalize all the different optimal backtrack rules into one set of rules that seem to work with most walks and grids. I believe that with this method, there are no illegal prunings of viable puzzle layouts.

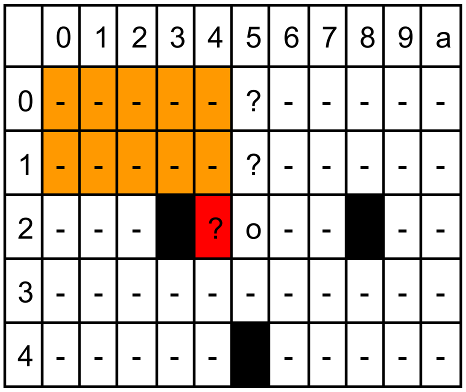
Letter Optimal Backtrack Rules:

1. For a square, all letters in the crossing horizontal and vertical words (up to the failed letter) can affect the failure of laying a letter, so mark them as an optimal backtrack target.

2. On the optimal backtrack, stop at the first optimum backtrack target that has possible letters to lay.
Important: If we stop at optimal backtrack targets that do not have letters to lay, then our script would treat it as a failed cell and start another optimal backtrack (which attempts to skip naive backtrack cells) , possibly pruning viable puzzle layouts.

3. On our optimal backtrack, if we hit a cell that is not an optimal backtrack target AND it is both left of, AND above the failed cell (indirect contributor), then we must revert to the naive backtrack mode.
We can ONLY optimum backtrack over cells that are to the right and below the failed cell. If we bactrack over cells to the left and above the failed cells, these cells contribute to the possible letters in the optimum backtrack targets. Ignoring them may prune valid puzzle layouts.

Letter search example:
scenario: trying to lay a letter at o fails
Rule 1: ? can directly affect placements of letters at o. So ? are our optimum bactrack targets.
Rule 2: If we backtrack to the red square and there are no possible letters left in that square, we must try to go to the next optimal backtrack target 5,1. If we did not, the next backtrack cycle might take us to 4,1 which will erase any viable options we had in cell 5,1.
Rule 3: Note that the orange squares can affect the letters at ? which can indirectly affect the letters we place at o.



|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | - | - | - | - | - | ? | - | - | - | - | - |
| 1 | - | - | - | - | - | ? | - | - | - | - | - |
| 2 | - | - | - | ■ | ? | o | - | - | ■ | - | - |
| 3 | - | - | - | - | - | - | - | - | - | - | - |
| 4 | - | - | - | - | - | ■ | - | - | - | - | - |

Data structure:
For a square ‘x_y_source’, I am using an associative array to store the optimal backtrack targets.
target_letters_for_backtrack[‘x_y_source’][x_y] == true for all x_y that are backtrack targets of ‘x_y_source’

## Letter Search Methods

Prefix / Linear Searches : Return the next possible letters after a given series of letters

Given the prefix letters for a word, and a word length, return the next possible letters.
eg: given 'boa??' found in the space for a 5 letter word, the next letter might be t or r.
This is useful only if you plan to build words from beginning to end. It will not allow for filling in random letter positions (eg: it cant solve for TO?TH)

Method 1.
For each word length I built a chain of hash references of the letters at each position in the word. This required that, for a search, our code had to follow the chain of hash references. The lookup code is more complex but it is fast and very memory efficient as it is a simple data tree.

Data structure:
For example, to find the first potential letters of a 5 letter word we would get the keys of next_letter[5].
If we decided to use the letter T as a first letter we would then return the keys for next_letter[5][T] to get the second 5 letter words starting with T. etc...

Method 2.
I decided that it was simpler to implement, and almost as fast, to just use an associative array using masks (with letter prefixes) as keys. That would eliminate the need to supply the word length. Also no code is required to look up the next letters.

Data structure:
Returning the keys for:
linear_word_search[‘CAo’] might return the potential next letters of:
['T' , ‘R' , 'B' , 'N' ]
Saving potential letters as keys, and not simply in an array, ensures there are no duplicate letters in the list.
Note: the mask must be a prefix mask only. ie: HIGoooooo

# Word Searches

## Word Walks

Two of my word walks are taken from the equivalent named letter walk, and when we find the first letter is a word, we push it on the word walk. Sik Sak and Slalom,

The Numerical walk simply follows the numerical clue numbers.

The Across and Down takes the Across words first then the Down words.

Random Words is just that.

With the Crossing Words walk I started with the very first word in the top right (although any start position could be used). Then I add all the crossing words of this starting word. Then I find and add all the crossing words to those words, etc. I only add a word once. This tends to backtrack very efficiently even with naive backtracking.

Data Structure:
next_word_positions_on_board is an array of direction and word number arrays in order
next_word_positions_on_board = [ [dir0,wordnumber0] , [dir1,wordnumber1] , [dir2,wordnumber2] …   ]

## Word Backtrack : Optimum

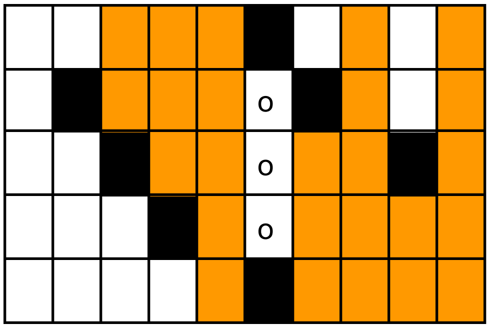
If a word attempt fails, the backtrack targets will be all of the crossing words ,and all of their crossing words as they immediately can block the placing of a word in the current location.

This is similar to a letter search. All cells in both the horizontal word, and vertical word, of the cell,  can affect the ability to place a letter in that cell.

I backtrack until I hit one of the bactrack targets..
You must stop on the first one encountered with possible words remaining or you risk losing possible puzzle solutions.

Grids that do not seem to benefit are the Double Word Grids with 100% interlock. This makes sense as all or most crossing words affect the word you are trying to solve for.

Word Search: Word marked by ooo can be blocked by horizontal and vertical words in orange



Data structure:
For a word word_backtrack_source, I am using an associative array to store the optimal backtrack targets.
target_words_for_word_backtrack[‘word_backtrack_source’][d_w] == true for all d_w that are backtrack targets of ‘word_backtrack_source’. d_w is a string if direction, underscore and word number.

## Word Search Methods

Mask Searches :
Return a list of potential words for a given mask.
words = wordsFromMask( ‘GOoD’)

Method 1.
For a given mask (eg GOoD), cycle through all the letters given. For each letter in the mask, return a list of all words, the same length as the mask, that have that letter in that position. Then return the intersecting values of each array.
The code is complex, and was not very fast as we had to repeatedly compare potentially large lists of words.

Its memory use was average to high as a word was stored multiple times, at least as many times as letters in the word. Eg: DOG was stored 3 times, once for each letter.

*Method 2.
Words of Length String. For each word length we have created a large, comma delimited string consisting of all the words of that length.
We use regular expressions to search that string for the mask and return the list of words.
The code is simple.
The memory storage is as efficient as one could hope for. eg DOG is only stored once in the comma delimited words of 3 letter string.
PERL regex is surprisingly fast for searching long strings. JS Regex, not so much…

Data Structure:

```
var pattern = new RegExp(unoccupied, 'g');
mask = mask.replace(pattern, '\\w'); //make a mask of 'GO$unoccupiedT' into 'GO.T' for the regexp
pattern = new RegExp(mask, 'g'); // /${mask}/g;
var possible_words_array = words_of_length_string[word_length].match(pattern);
```

Method 3. Binary Mask. For each word we build every possible mask. Then for each mask we create a list of words that belong to that mask.
Eg: CAT -> CAT , CAo , CoT , Coo , oAT , etc
In theory it should be very fast. But it is only slightly faster than the Words of Length String word search ‘in some cases’.
Copying or accessing the list takes the majority of the time and therefore is not much faster for cases where the list returned will be large. Small returned lists are faster as the Method 2 still needs to search the complete comma delimited string.
So smaller returned word lists will show a speed improvement with this method.
However, this method uses large amounts of memory as longer words cause exponential memory growth.
It also takes a long time to build the database as we need to create many (grows exponentially as the word is longer) binary masks (and associated word lists) for every word in the dictionary.

Method 4. Possible words from letter lists Search (meta search) : choosing words that fit based on the crossing words. Note that it still needs to use one of the searches above for individual words.

This routine takes a list of possible letters (based on all the crossing word masks) for each position in a word and will output a list of words that can be made with said letters.

```
words = wordsFromLetterLists(['C','D','F','T','Z'] , ['E','R','T','Y','O'] , ['T','R','E','W','Q','Z']);
```

Eg: We want to find possible words at 5 down. We find all the masks that cross this word. We find all the possible letters that can exist in our word's cells (5 down). Then we calculate what words we can make with these sets of letters.

It takes a lot of processing time. We must first search for possible letters in our word based on all the crossing words. Then we must search all the combinations of those letter sets for words in our dictionary.

| | | | | |
|---|---|---|---|---|
| - | - | o | - | - |
| - | - | o | - | - |
| - | - | o | - | - |

It can take up to 250 times longer than a simple mask word search ), and 833 times slower than the letter search on some grids. However, it is faster on many types of grids as it looks at blocks of letters, not just crossing words. Therefore it notices errors sooner (less recursions) and avoids a lot of inefficient blind recursions that can occur with the other search methods.

Note: It really shines if your walking/search path ensures that each following word crosses the previous one(s).

It will also outperform letter searches on very large grids as its search area grows much slower than a letter search.

# Benchmarks

Based on the section on Double Word Squares at: http://en.wikipedia.org/wiki/Word_square
I feel my program is running well. The article states that 8 x 8 is around the largest order Double Word Square to be found using dictionary words. Considering my limited dictionary, my program can frequently generate a 6 x 6 crossword in around 3 seconds.

# Some things I have learned

1. The bigger the word database the better.

2. Single letter searches are not the answer (for some grids) as there are too many combinations. For each lay of a letter you only check at most two words with the letter in common. So even with the fast calculation time for each letter, all the recursion adds up. Complete word searches (Laying a word and simultaneously checking that all the crossing words will have a possible word) although much slower, has fewer recursive calls. On most of my test scenarios, laying words was faster. Another way of stating the benefit of laying words is that a single random word replacement searches the puzzle space quicker. For example, when

laying the first letter, if that letter will never result in a valid puzzle, we may not know until millions of recursions have passed. We discover mistakes faster by laying out whole words.

3. Each time you lay a word, you must ensure that all its crossing word positions will be able to fit a word. If not, you will perform too many recursions. The code is more complex, but it is worth it.

4. Your path algorithm for laying words (the order in which you lay words) should ensure that each following word in your path crosses the previous one. This helps reduce wasted recursions. A case of an inefficient recursion path would be trying to lay all across words, then checking if the down words are valid.

5. Optimal backtracks (see CCP) help a lot (x100 speed increases or more in some cases).

6. Recursively designed routines, although not required, seem more logical and suited for this sort of program.

7. The puzzle space is immense. The puzzle solution space is minuscule. Therefore simple recursive and random attempts are unlikely to work in a timely manner on their own. We need to prune the search space using choice forward paths optimal backtracks.

# Possible Explanations

# Ideas/Questions

How can we create an efficient walk generator and optimal backtrack for any puzzle grid?

Could we place the backtrack and walk tasks in their own sub routines and still use recursion? Assuming we can find suitable data structures, maybe the overhead will slow it down?

Are dynamic walks (where the next walk location is chosen based on the current puzzle/grid fill) possible without missing valid puzzle combinations?

Can we further increase generation time by choosing more suitable or likely words first?

I need to write a better clue selection routine to allow for different styles.

http://en.wikipedia.org/wiki/Word_square