Automated crossword puzzle generation in C
INTRODUCTION

The goal of this project was to create an efficient, automated crossword generator in C. While commercial software already exists for this task, the purpose of this project was not only to investigate the problem for my own sake, but also to produce a program that runs very fast and that produces results in a reasonable amount of time on challenging grids (grids with few black squares).

More precisely, the program takes two items as input. The first one is a square grid of size at least 3 by 3, with #'s to represent the black squares, -’s to represent empty squares, and capital letters for initially filled-in squares. The second is a list of files containing the words that can appear in the crossword. The output is the same grid, filled in with words from the input files, each word appearing only once in the final grid.

BRIEF SUMMARY OF THE ALGORITHM

Most algorithms for the automated generation of crossword grids rely on some form of depth-first search and backtracking, which is what has been done in this project. Here is a very rough outline of the algorithm (see below for a more detailed specification): given a slot in the grid, generate all the words in the dictionary that could fit there; then, for each word, write it into the grid and repeat the procedure on the next slot, until all slots have been successfully completed.
DETERMINANTS OF RUNNING TIME

There are several aspects in the problem of crossword generation that determine the running time to a significant extent. The first one is the choice of the filling path, that is, the order in which the word slots will be filled in. Second is the time it takes to generate the list of possible words that could fit a certain partial specification. Third is the order in which the candidate words for a given slot are evaluated. Fourth is the size and nature of the dictionary.

1. Filling path

The choice of the filling path determines the running time of crossword generation to a significant extent. For example, a filling path consisting first of all the across slots followed by all the down slots would fail miserably. That method proceeds blindly through the grid, without regard for the constraints added to the grid each time a new word is inserted. Therefore, a simple, intuitive approach to the filling path problem is to always next consider the word slot in the grid that can accommodate the fewest number of words. The dynamic nature of this approach causes the algorithm to take into account the state of the grid at all times. As a result, for a given depth in the filling process, the slot considered may not always be the same.

2. Generating all the words that fit a certain partial specification

When considering a slot for word insertion given what letters are already there, the problem of retrieving all the words that could fit there arises. The naïve approach is to simply scan the entire dictionary, comparing each word to the partial specification and
seeing if there is a match. With large dictionaries, however, this is prohibitively time-

consuming. The main idea in solving this lexical lookup problem is to partition the
dictionary beforehand, in a way that will produce smaller sets of words, each of which
can be reached in constant time. In the implementation for this project, the dictionary was
partitioned in a three-dimensional array, where each entry consisted of a linked list of
words of length i, with j being the letter in the kth position, so that array[5][B][0] for
example would have a linked list of all the words matching the partial specification
“B - - - - ”. A separate one-dimensional array was maintained that contained all words of
length i. Both word arrays had a companion array that contained an integer value for the
number of words in each linked list.

This is a solution only to the extent that the list of words being scanned is smaller
than the entire dictionary. One significant improvement was the introduction of caching.
A hash table was maintained, where each entry in the table was a linked list of nodes of
previously-encountered partial specifications. Each node contained the partial
specification, a linked list of words that could fit that partial, the associated count and a
pointer to the next node in the hash chain. Caching is particularly important in light of the
fact that all slots in the grid are being considered for next most constrained. Once a slot
has been chosen as most constrained, a word is written into it, and the state of most of the
other slots hasn’t changed, and so neither has the list of words that can fit there.

There is a space-time tradeoff when caching previously encountered partial
specifications. With time, the number of word lists that are saved grows to be quite large,
and if all solutions for an input grid/dictionary pair are wanted, it is important to
periodically free the storage allocated for these word lists so that the program can run for
a long time. However, it should be noted that some of the word lists saved in the hash table may be in use by the backtracking function, and therefore, the storage allocated for them cannot be freed immediately. To address this concern, each word list in the hash table has an “in_use” counter, that is incremented each time it constitutes the list of words that will fit in the slot that is able to accommodate the fewest number of words. It is then decremented when the backtracking function has tried on the grid all the words in the list.

3. Order of evaluation for candidate words

   The order in which candidate words for a given slot are tried significantly influences the running time of the algorithm. The approach used in this project was to sort the candidate words in decreasing order of convenience, where a word’s convenience for a given slot is defined as the product of the numbers of words that could fit in the slots connected to that slot, if the word in question were filled in. The advantage of using the product is that a word’s convenience attribute is automatically 0 if its insertion into the grid will cause an impossible partial specification. Because these convenience attributes can get quite large (on the order of $10^{35}$ at times), the type “double” was used in the implementation.

   For example, in the following grid, to calculate the convenience attribute of a word that would be inserted in the shaded area (across slot), the word is temporarily entered into the grid, then the product of the numbers of words that match the new partial specification of each down slot connected to the slot in question (marked with a vertical line) is the word’s convenience attribute.
Figure 1 – Calculating convenience attributes

Note that this one-level look-ahead heuristic is a greedy optimization that is not necessarily optimal, as a certain word may be good one level down the filling path but may cause trouble two or more levels down. However, n-level look-ahead still offers the same problems. It would be less short-sighted than the one-level look-ahead, but would not know what goes on at the \((n+1)\)th level, and would take a longer time to execute because of the exponentially-growing complexity of calculating convenience attributes with each increasing depth level. Moreover, a full look-ahead heuristic would simply reduce to solving the problem we are already trying to solve in the first place.

One aspect of the program that reflects the advantage of this heuristic is that, at least on all tested grids, the program finds a solution grid without ever backtracking when the input dictionary is a list of words that are a known solution to the grid.

A first version of the program randomized the order of evaluation of candidate words. This performed about as badly as, if not worse than, an alphabetical evaluation
order. The one-level look-ahead heuristic clearly outperformed both in terms of running time.

4. Size and nature of the dictionary

The main dictionary used for this project was a former version of /usr/share/dict/words, with other additions including inflected forms, short idioms, expressions, abbreviations, acronyms and various prefixes and suffixes. It contained about 55,000 words. Intuitively, the larger the dictionary, the more likely a solution will be found, though this also depends on what kinds of words are additionally available.

TECHNICAL DETAILS

1. Backtracking function

Among other arguments, the backtracking function called `makeIt` takes a pointer to a `struct word_slot`, containing information about the slot to be filled-in (coordinates and either across or down), and a pointer to a linked list of the words that can fit the slot in its current state. It then generates a linked list of all the slots that share a square with the slot to be filled-in (these are all mutually parallel, while each is perpendicular to the slot to be filled-in). Then, the product of the number of words that could fit in each of these slots is computed for each word in the word list, if that word were actually written into the grid. As noted above, this product will be 0 for a word whose insertion into the grid causes the partial word specification of a slot to match no words in the dictionary. Caching is also very important here since the same partial
specifications may be encountered over and over again for the candidate words that share
the same letter in the same position in the word.

The list of words is then put into an array, which is sorted using bottom-up merge
sort, in decreasing order of convenience attribute. These words are then tried into the grid
in that order. After a word is written into the grid, among all slots that don’t have a word
written into them, the one that can accommodate the fewest number of words is selected
as the next word slot to try. The backtracking function is then called recursively on this
new slot. If the new slot can’t accommodate any words, then the recursive call returns
immediately, the letters written into the previous slot are removed and the next word in
the list of words to try is written into the grid. If there are no more words left to try, then
makeIt returns at once.

2. Lexical lookup

Before the backtracking algorithm is executed, some dictionary preprocessing
takes place, with the goal of partitioning all the words into smaller sets so that word
search takes a minimal amount of time. As noted above, given a letter, its position in a
word, and a word length, a linked list of all the words satisfying those constraints and the
length of the linked list can be retrieved in constant time thanks to the dictionary
preprocessing. Moreover, all words of a given length can be retrieved in constant time.

Note that at any time during the execution of the program, there is only one live
copy of each word in the dictionary; all linked lists of words only contain pointers to the
individual storage for each word.
To get the list of words that match a partial word specification with more than one letter, the relevant array with the fewest number of words is chosen to find all the words that match the partial. For example, if a slot is “- - - W - - Z”, the shortest of the list of words matching “- - - W - - -” and the list of words matching “- - - - - - Z” is scanned in order to find all words matching “- - - W - - Z”.

RESULTS

The results produced by the final version of the program were, overall, very satisfactory. On grid geometries of typical 15 by 15 crosswords, the program consistently returned with a solution in about 0.3 seconds or less. On more challenging grid geometries, results varied between 1 and 10 seconds approximately. However, on some other fairly challenging grids, the program did not terminate even after about 300 million backtracks, i.e., the number of times a word was removed from the grid after having been unsuccessfully tried. This is probably due to the filling approach that was implemented in the program.

In the implementation, the slot to be filled in next is determined by the number of words that could fit in each slot that has yet to be completed (whether it is partially filled-in or empty). The one with the smallest number—the most constrained one—is then chosen. This is a slot-by-slot approach that considers the grid as a collection of individual, independent word slots. As a result, the following problem can arise. In figure 3, the slot that was filled in last was “SSN”, the down slot in the lower left corner. The next most constrained slot was found to be “- - I - T - O - -”, the down slot in the upper right corner. It turns out that while “- - I - T - O - -” and “S A I - - - -” can both accommodate words,

\[^1\] On an “Intel(R) Xeon(TM) CPU 2.80GHz” machine
no two words from each set can coexist in the grid, making that section of the grid temporarily impossible. Rather than trying to alter that part of the grid, the implementation here simply backtracks to the point of last success, “SSN”. Any changes made in the lower left corner, however, will not help to solve the problems in the upper right corner.

A solution to this problem would probably involve segmenting the grid into pieces or grid areas more meaningful than just a series of individual word slots. The idea would be to partition the grid into areas where this kind of problem could not occur. Within these partitions, the standard backtracking applied in the current implementation would then be adopted. It is not clear to me at this point how one would approach the task of making the partition. This deserves further investigation.

The following is an example result, where the input grid only had the ‘X’ of “XYLOPHONE”. The program returned after 277239 backtracks and 3.6 seconds.

Figure 2 – Backtracking problem
CONCLUDING REMARKS

The main strategies adopted here were: always prioritize next the incomplete slot that can accommodate the fewest number of words; sort the candidate words for a slot in decreasing order of one-level look-ahead convenience; keep a hash table of previously encountered partial word specifications and the associated word lists for fast lexical lookup.

While the problem of automated crossword generation may appear simple at first, as grids become more and more challenging (i.e., have fewer black squares), efficiency concerns and storage considerations grow in importance, as the amount of backtracking required increases considerably. Moreover, finding a proper filling approach is crucial in these cases in order to avoid making fruitless backtracks (as in the unlikely yet fundamental problem discussed above). Further improvements on the approaches described here likely lie in partitioning the grid into disconnected sections to be worked on independently, such that in each section, the next slot to be filled in is connected to the last as often as possible.