There are no English double word squares of order 9

Ilkka Kokkarinen, Toronto Metropolitan University, Canada
ikokkari@torontomu.ca

Draft version September 11, 2024

Abstract

Word squares are a classic linguistic amusement where an n-by-n grid is filled with letters so that every row and column is a legal English word. In a more challenging variation of double word squares, no word may be repeated in the square. Double word squares of English words have been constructed up to the order $n = 8$. An exhaustive search through the possible combinations using backtracking with constraint propagation, along with a dictionary with a total of 53,402 nine-letter words, conclusively revealed that no double word squares of order $n = 9$, nor even more unlikely $n = 10$, exist in English.

Introduction

A word square is an n-by-n grid with one letter placed in each cell so that each row and column forms a legal word recognized by the dictionary being used. The Wikipedia page for word squares gives several examples of word squares, the most famous of them of course the medieval "Sator square"

```
S A T O R
A R E P O
T E N E T
O P E R A
R O T A S
```

later immortalized in fiction such as the 2020 science fiction blockbuster film titled after the center word. In such a basic word square, both the i:th row and column contain the exact same word, which makes the construction of such squares largely trivial for small n, at least until n becomes seven or eight or thereabouts.

However, this problem suddenly becomes massively more interesting and difficult once repetition of forbidden so that the words in all rows and columns are distinct. These so-called double word squares are significantly more difficult to construct; solutions in English have so far been found only up to order $n = 8$, and no double word squares of English words above this size have been discovered. The following example for $n = 8$ was discovered by Jeff Grant in [1]:

```
T R A T T L E D
H E M E R I N E
A P O T O M E S
M E T A P O R E
N A I L I N G S
A L O I S I A S
T E N T M A T E
A S S E S S E D
```

Despite its smaller vocabulary, the smaller alphabet of Latin makes it easier to construct larger double word squares. (We will resist making any puns about "Latin" squares.) Eric Tentarelli gives the following $n = 9$ double word square in Latin in [2].

```
A D A M P L I A S
D E M I R A N D O
P R I S A N T U R
L U C E S C E N T
O P E R I E N D I
S E M I N A T A E
U R I T A B A N T
R E N A T A N T I
A M I S I S T I S
```

Backtracking through possibilities

This article describes this author's search for an English double word square of the order $n = 9$ using an optimized backtracking algorithm. This hunt for the combinatorial white whale started out as an exercise for the course that the author taught in Chang School of Continuing Education, Toronto Metropolitan University, continued there as an idle hobby to be occasionally pondered, until some crucial insights led to the efficient algorithm used now to complete the exhaustive search of all possible double word squares of that size.

Finding the 9-by-9 double word square would first seem to be a constraint satisfaction problem with 81 variables, each variable having the range of values of 26 English language letters from *a* to *z*. The constraints would then simply say that each row and column must be a distinct word in the given wordlist. However, a far better formulation will be to define this problem as having 18 variables, nine for the rows and nine for the columns of the grid, each of these variables ranging over the set of all nine-letter words. The constraints then require that all 18 variables have distinct values, and that each row and column share the same letter in the position of their intersection.

We used the classic backtracking approach to solve this problem, with some constraint propagation for pruning some search tree branches that cannot possibly lead to a solution. The backtracking algorithm fills the grid alternating between rows and columns, proceeding downwards along the rows and rightwards along the columns. Both rows and columns are numbered starting from zero, as customary in computer science. Filling in the columns and rows alternates so that at even levels $k = 2m$ of recursion the algorithm tries to place the word in row m, and at odd levels $k = 2m + 1$, place the word in column *m.* Since any double word square remains a double word square after transposing, the word in the first column may as well be constrained to be lexicographically lower than the word in the first row, to cut the search space in half. Even under this constraint, there are millions of ways to choose these first two words for most of the fixed first letters in the top left corner.

After filling in the first couple of rows and columns, the remaining possibilities for the words for the later row or column are typically pretty heavily constrained by the prefix of characters from the words chosen in the previous levels of recursion. Since the fixed wordlist is given and kept in sorted order, simple binary search will quickly find the starting position to iterate through the words that start with the given prefix. As a small optimization, the starting positions of all one- and two-character prefixes in the wordlist are precomputed for quick lookup while reading in the wordlist.

As in all interesting constraint satisfaction problems, the backtracking search lives and dies on its ability to quickly recognize early partial branches of the search tree that are dead ends in that they cannot possibly be extended to a complete solution. To this end, our algorithm uses a simple but effective early pruning technique at levels two and three of recursive backtracking. At level two, after placing the word on the second row, we check that every two-character column prefix formed by the words in the first two rows can be extended to at least one legal word found somewhere in the wordlist, before advancing the level three to fill in the word in the second column. For example, in the following example situation for $n = 6$, the presence of the column prefix `wd` would cause the search to reject the word `ardoit` and move on to the next possible choice for the second row:

```
i s w a r a
a r d o i t
m . . . . .
b . . . . .
i . . . . .
c . . . . .
```

A symmetric check for the two-character prefixes of each row is done at level three of the backtracking recursion. Precomputing all two-letter prefixes found in the wordlist makes both these checks blazingly fast. Since a large number of two-character prefix-es (such as `qz`) are not prefixes of any word in the wordlist, and the presence of any such prefix in the first two rows allows the search to immediately abandon that branch, this check gifts us trillions of quick rejections during the run of the entire backtracking search for nine-letter words.

Constraint propagation

Immediately after crossing the hurdle of the column and row prefix checks at levels two and three, a more advanced constraint propagation check is applied to recognize many partially filled dead end grids that cannot possibly be completed into a working solution. For each empty cell in the grid, we compute the set of remaining letters that are still in possible for that cell. As soon as the set of possible letters for any cell of the grid becomes empty, completing the solution is obviously impossible, and the search can move on to try the next available word for the second column.

These sets are computed iteratively only at this level of recursion in the following fashion. Initially, the set of possible letters for each empty cell contains all 26 charac-ters from *a* to *z*. The sets of possible letters for each empty cell are encoded in 32-bit unsigned integers for compactness and efficiency of set theoretic bitwise operations.

Initially, each row and column of the grid after the first two rows and columns is marked as needing checking. Of all the rows and columns that still need checking, choose the row or column with the tightest two-character prefix, that is, the prefix for which the wordlist contains the fewest words that start with that prefix. (This count, of course, has been precomputed for all two-character prefixes during reading in the wordlist.) Loop through all words that start with that prefix, but skip all words where some letter after the second position is not in the set of remaining letters of the cell that letter would fall into.

Using only words that survived the previous filtering, compute the new set of possible letters for each cell in that row or column as the union of the letters that the surviving words contain in the position of that cell. If the new set of possible letters for some cell is smaller than it was before this step, the corresponding row (if currently processing a column) or the column (if currently processing a row) that contains that cell is marked as still needing checking, even if that row or column had already been processed earlier.

In practice, this constraint propagation yields a massive speedup by immediately rejecting the vast majority of partially filled grids without having to loop through all the possible words for the third row and column only to see all such possibilities eventually leading to failure. Since the remain sets computed at level three are kept in memory for later levels of recursion, they allow the search to skip over any word whose prefix matches the current prefix of that row or column, but some later letter is not in the remain set of the cell that it would fall into.

To speed up the backtracking algorithm, the general purpose technique of undo stack is used to downdate the data structures to undo either having placed a character in a cell, or having updated the set of remaining characters for an unfilled position.

Results

After all this work, it was bit of a disappointment that the exhaustive search performed over the period of four days using a current off-the-shelf Intel laptop running Ubuntu Linux, the program implementing the algorithm compiled with `gcc` under the `-O3` option, did not discover any double word squares of order 9, leading us to the unfortunate conclusion that such squares simply do not exist in English.

Having the algorithm output the deepest level reached during the recursion along with the prefix and remain cutoff counts shows that the grid can sometimes be filled up to level 13 or 14, that is, filling in the first seven rows and the first six or seven columns. However, the remaining small empty rectangle at the lower right corner stubbornly refuses to be completed simultaneously in both directions. Close, but no cigar. One early example of such partially filled square would be

```
a b a c i s c u s
b i s o n t i n e
a n y s t i d a e
c o m p e n s e r
i m m e r g e n t
n i e c e l e s s
a n t i s e r u m
t a r e s s . . .
e l y s e s . . .
```

After completing the run to search for a 9-by-9 double word squares, we attempted the moon shot of finding an 10-by-10 double word square. This search finished in about the day, despite the dictionary containing almost as many ten-letter words as it does nine-letter words. This speedup to the massively increased proportion of two-character prefix cutoffs at levels two and three that decreased the number of expensive remain set cutoffs considerably. Unsurprisingly, the search for 10-by-10 double word squares also failed to discover any such squares.

Conclusions and future work

The backtracking search implementation `wordsquare.c` (released under GNU Public License v3) and the wordlist file `words_sorted.txt` (adapted from another GitHub repository [4] where the wordlist was released into the public domain) that contains 53,402 nine-letter words are freely available in author's GitHub repository https://github.com/ikokkari/Wordsquare.

After cleaning out the bugs and implementing a host of other small optimizations not described here, the algorithm spits out double word squares for smaller grid orders *n* quickly and correctly. For *n* = 6 and lower orders, the set of all double word squares is so dense that each word that appears in the first row of one double word square can usually be completed into a double word square in a gargantuan number of ways. Since our program can be given the first and last word to try for the first row as command line arguments when launched, the search can be easily parallelized over different parts of the wordlist.

All is not lost, since a faint hope remains in that perhaps this exhaustive search can be repeated using a larger wordlist of acceptable words, perhaps stretching the rules a tad bit by allowing proper geographical and other names as words. We will also perform an exhaustive search for double word squares of orders $n = 10$ onwards. Even if the hope of finding a double word square of such enormous size is somewhere in the order of winning the lottery, surely there is no harm in making sure, computing power being essentially free for citizen scientists in this day and age. Furthermore, our program can be used to generate all double word squares of order $n = 8$, the majority of which must surely be yet unknown at this time.

Speakers of other languages may also be interested in adapting the program to look for word squares in other languages than English. The only change needed is encoding the remain sets characters into unsigned integers, perhaps using a 64-bit integer type for larger alphabets with more than 26 characters in them.

References

[1] Grant, Jeff (1992) "Double Word Squares," Word Ways: Vol. 25 : Iss. 1 , Article 3. Retrieved from: https://digitalcommons.butler.edu/wordways/vol25/iss1/3

[2] Tentarelli, Eric (2020) "Large Word Squares in Latin," Word Ways: Vol. 53 : Iss. 4 , Article 9. Retrieved from: https://digitalcommons.butler.edu/wordways/vol53/iss4/9

[3] Wikipedia: "Word Square". Retrieved from https://en.wikipedia.org/wiki/Word_square

[4] GitHub repository english-words. Retrieved from https://github.com/dwyl/english-words