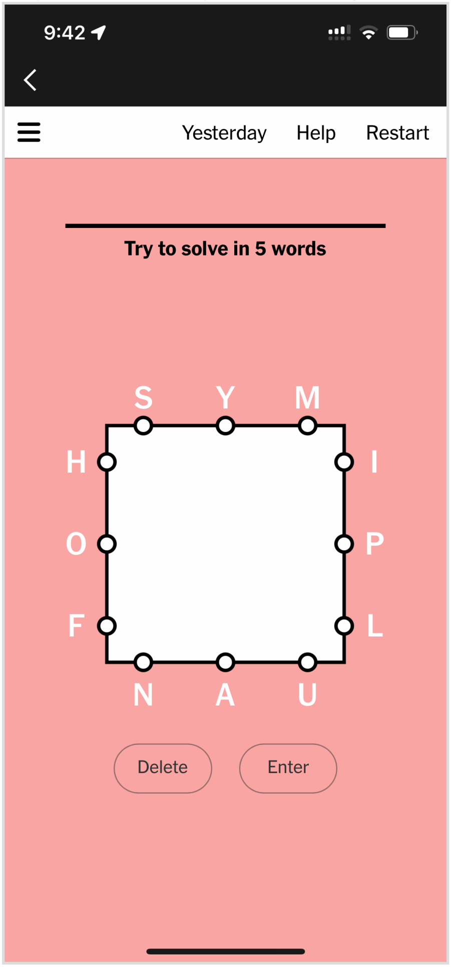
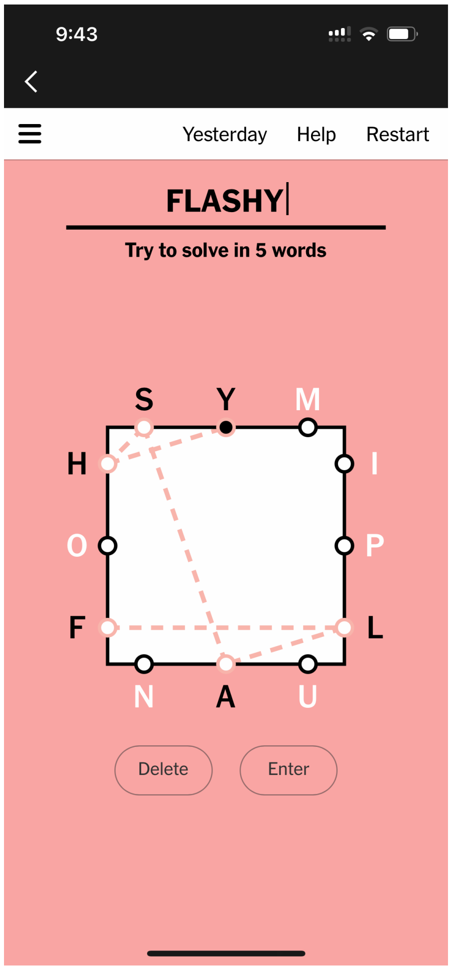
Background

- As word games go, Letterboxed from New York Times is one of the puzzles I go to the first thing in the morning. The rules can be summarized as follows:
- Starting From any of the 12 letters organized along 4 sides, each step takes you to a different side and by implication a different letter.
- A series of such steps can terminate completing a word.
- The next word is similarly completed starting from the last letter of the previous letter.
- Game concludes when all the letters have been visited. The puzzle challenges you to complete the game in a certain number of words.



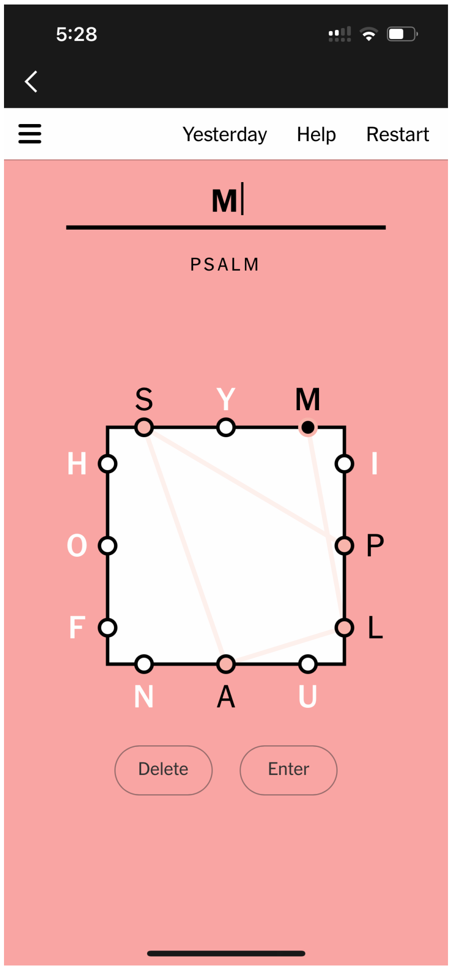
Starting view for the puzzle

Does this sound like a variation on the Traveling Salesman problem? The dictionary lookup poses a special twist and challenge. Consider the following first word in the above puzzle. The first word productive having covered 6 letters but the last letter Y is a bit problematic. There are not too many words with Y as the first character. I find myself tying myself in such knots and have to backtrack. In particular since the puzzle seems to feature letters such as Y too often - in fact way more often than can be anticipated from studying the dictionary.

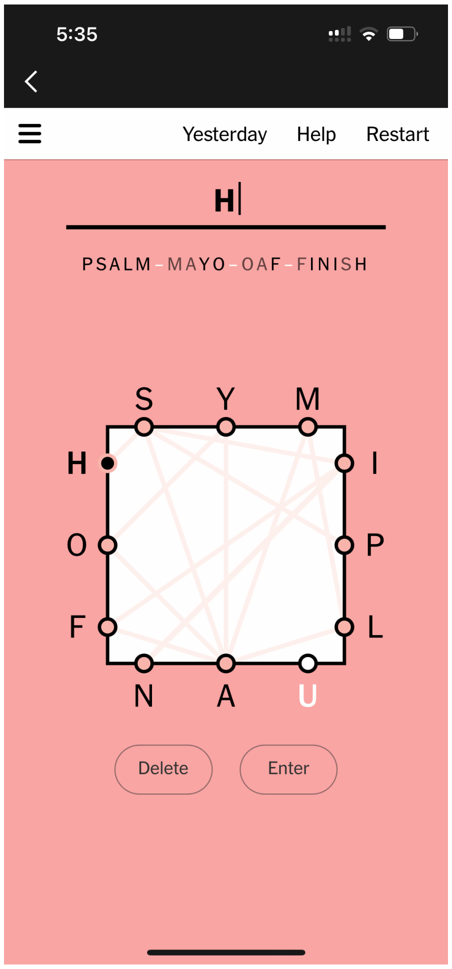


First Word

Starting over again, we can chose a different starting word ending up with M as the first character for the next word



Another attempt



Wrapping up, this is the situation after adding 4 words leaving one letter uncovered.

Automating the solution

After a few solutions, I was hooked. Every puzzle was interesting in its own way. Some days I saw difficult letters like Y or Z appeared. Other days the puzzle had a Q and thankfully an U. The frequency of appearance of such complications were puzzling but I kept at it. One particular puzzle was too complicated and I could not solve under the suggested number of words. A software tool was in order.

Initial Attempt

Initial attempt was basically to replicate the manual process namely start with a random letter and attempt to make a word and thereafter the next step and so on. Before too long I was tying myself into knots - trying to deal with all the backtracking, ensuring coverage of all the letters and keeping the solution under the specified number of words. Given a puzzle, it was not clear how many solutions exist.

Brute Force Method

The next step was to take a brute force "greedy" approach. Essentially this meant:

- For each starting letter - traverse the puzzle space enumerating valid words - thus resulting in 12 tables of words - one for each starting letter.
- For each word, build up a chain of words starting with the word - choosing the last letter of the word as the first letter to choose the next word
- For each such word chain, we can check whether all the letters have been visited (used in a word). Once a word chain is found with all the letters visited, we have a solution.
- It turned out the number of solutions is numerous. For example a puzzle featuring the letters: "paf uot mie xlr" has 84964 solutions of 4 words each. (This may also say something about the dictionary used.)

Data Structures used:

The Game

It is easy to see how to parse a game defined as a series of 12 characters located

```
type Side is
      ( left, top, right, bottom ) ;

LETTERS_PER_SIDE : constant := 3 ;
type Position is new integer range
1..LETTERS_PER_SIDE ;

type Game is array
(Side'Range , Position'Range ) of
Character ;

type Visited is array
(Side'Range , Position'Range) of boolean ;
```

each character in its own bucket in the **Game** data structure.

Enumeration of words

The first phase of the solution is to start with each position in the **Game** and travel to a different side, a letter in that side and checking if this sequence is actually a dictionary word. If it is a word, it gets added to a **Vector** of **Unbounded_String's**. While we do this, we build up a **GameSummary** with 2 tables. The table accessed as **w i**s a list of words for each starting letter in the puzzle. All the words for a particular starting character have the same first letter. While this is getting built up, another table accessed as **wi** is a table of words indexed by the first letter, ie all the words in the list of a letter **"x"** have **x** as the starting letter. This is a convenience to enable forming the word chains as needed.

```
package Words_Pkg is new
Ada.Containers.Vectors( Natural ,
   Ada.Strings.Unbounded.Unbounded_String );

type WordsType is array
(Side'Range , Position'Range ) of
Words_Pkg.Vector ;

type WordsIndexType is array
(character'Range) of Words_Pkg.Vector ;


type GameSummaryType is
     record
        w : WordsType ;
        wi : WordsIndexType ;
     end record ;
```

Once the **GameSummary** is prepared, it is just a matter of walking through the words, picking the next word based on the last character of a word and checking whether we have reached a solution or not.

Example runs

```
      38617 word lines read
Dictionary contains       38617 entries
      u    o    t
  p                 m
  a                 i
  f                 e
      x    l    r
Solution:           1 ; wordcount:          4
pupil
later
rumor
reflex
Solution:           2 ; wordcount:          4
pupil
later
romp
prefix
Solution:           3 ; wordcount:          4
pupil
later
romper
reflex
```

As can be seen, the brute force finds numerous solutions. Running through the entire list reveals that there are 84964 solutions with 4 words.

This was done in 11.10 seconds including the printouts on a MacBook Pro. While it is not too shabby, it could potentially be improved.

Potential Improvements

- The dictionary contains 38617 words. This could probably be reduced to 40% or so if we discard all words which do not start with one of the 12 characters in the puzzle. This will help reduce the searches tremendously.
- Words which have letters which don’t appear in the puzzle can also be eliminated - thought this additional cost of filtering may not be worth it.
- In any case since the dictionary is maintained as a hash table, the benefit of these optimizations may not be too great.

SAMPLE IMPLEMENTATION

The repository specified below contains a prototype solution in Ada:

https://codeberg.org/RajaSrinivasan/letters.git