A theoretical note on learning languages

Virinchi Rallabhandi

January 2023

1 Introduction

What do we do if we don't understand the meaning of a word? While most people I pose this question to greet me with a stunned silence assuming I've asked them a trick question, it turns out this question has more to it than first meets the eye, without introducing any tricks. The most logical answer is of course that we consult a dictionary. But, if you're like me, then you're familiar with the disappointment of reading the definition, only to find it contains other words you also don't understand, which you now have to also look up. Thus, the process is recursive. Furthermore, it reveals that one needs to know something to learn something - much like money, it takes some to make some. After one such scenic tour of the dictionary, I was brought to consider the end point of this recursion. More precisely, what is the minimum number of words in the English language[1] one needs to already understand, so that the meaning of every other word could (theoretically) be understood with the aid of a dictionary? The remainder of this essay is devoted to answering this question.

2 Method

For the purposes of my analysis, the dictionary is very naturally represented as a directed graph. The vertices are the words defined in the dictionary and an edge points from one word to another if and only if the first word appears in the dictionary's definition of the second. With this representation, there are two very distinct components to the problem. The first is data processing - actually creating the graph, given a text file containing the entire dictionary. The second component is essentially a graph search problem. I have to find the minimum number of words whose meanings I assume are known, such that all further words can be understood. In terms of the graph, I'm effectively ignoring all edges leading into the known words/vertices and then trying to arrange the vertices into an order - listed, say, left to right visually - such that none of remaining edges points from right to left[2]. In computer science terminology, this is nothing but the well known problem of topological sorting. The complete code I wrote for this project and the dictionary file I used are available at the accompanying GitHub page - https://github.com/VirinchiRallabhandi/DictionaryAnalysis.

2.1 Data processing

Data processing was surprisingly more complicated than I anticipated at the outset of this project. As an example, consider the following three entries in the dictionary.

[1] not that this problem is specific to English over any other language

[2] If the graph is disconnected, then this discussion applies to each connected component.

Hundred adj. & n. (pl. Hundreds or (in sense 1) hundred) (in sing., prec. By a or one) 1 ten times ten. 2 symbol for this (100, c, c). 3 (in sing. or pl.) Colloq. A large number. 4 (in pl) The years of a specified century (the seventeen hundreds). 5 hist. Subdivision of a county or shire, having its own court. hundredfold adj. & adv. Hundredth adj. & n. [old english]

Thousand adj. & n. (pl. Thousands or (in sense 1) thousand) (in sing. prec. By a or one) 1 ten hundred. 2 symbol for this (1,000, m, m). 3 (in sing. or pl.) Colloq. Large number. thousandfold adj. & adv. Thousandth adj. & n. [old english]

Billion adj. & n. (pl. Same or (in sense 3)–s) 1 a thousand million. 2 (now less often) a million million. 3 (in pl.) Colloq. A very large number (billions of years). billionth adj. & n. [french]

I chose to represent the graph as a list of lists; the first word in each list is the word being defined and the subsequent words are the ones appearing in the definition. An adjacency matrix type approach would be unnecessarily space and time inefficient given the number of entries in the dictionary is far more than the number that appear in any definition. As these examples illustrate, the first word of each entry is the word being defined. But, from then on, it's more complicated.

To get a consistent graph, the later words in each list must also be vertices of the graph. A number of annoying characters get in the way of that - e.g. capitalisation and punctuation marks. I think it's fair to assume a dictionary user understands English grammar even if they don't understand the meaning of a specific word. Therefore, I start by simply deleting all punctuation marks and making all characters lowercase. The one exception is brackets, which require special treatment I've discussed below. Apostrophes also pose a problem, because removing them tends to change a word into plural form or simply create a list of characters that isn't a word at all.

When a word has multiple definitions, the different definitions are labelled by a number, e.g. "1 ten times ten. 2 symbol for ... ." For constructing the graph, these numbers are a nuisance and I've chosen to just removed them. Occasionally, the dictionary I used would list the different definitions in different entries. This was just annoying and required awkward, fiddly code to keep track of.

Each entry has a number of "words" that exist solely for the purpose of grammar, example and etymology. These are "words" such as "adj.," "n.," "[old english]" or "(billions of years)." I think it's fair to assume a dictionary user already understands the meaning of these "words" - e.g. in the case of grammar "words" like n. - or they don't need to understand the meaning - in the case of etymology or the word being used for exemplar purposes. As a consequence, it makes sense to remove both brackets and any characters appearing inside a pair of brackets.

Recognising which phrases are of a grammatical nature is quite hard for a computer though. I could only find one somewhat sensible solution. I scan the dictionary once to find out all the words listed. Then, I scan it again, to create the graph, and I only include words in my

lists if they appear as words defined in the dictionary. Phrases such as adj., n. or Colloq. are not going to be words defined in the dictionary and hence will not appear in the list of words appearing in the definition of any other word. Similarly, phrases that aren't actual words, created by deleting apostrophes, will also be automatically ignored.

This decision has a potentially drastic effect on another data processing problem. While grammatical information like adj. or n. could be recognised by enumeration - just like I identified punctuation marks - it's much harder to detect verb conjugations and plurals. Given I'm assuming a perfect knowledge of grammar, it's fair to assume that understanding a word means all variations of that words are also understood. But, English grammar and spelling are not consistent in any way. Without some serious natural language processing, it's impossible to know when a word is a conjugation of a known word and when it's in fact something new entirely. The effect of my earlier decision to simply remove words in a definition that are not themselves entries in the dictionary is to also remove all conjugated verbs and plurals. Effectively, I'm assuming the meaning of these words is known. In my opinion, this is the only potentially serious, and somewhat unquantifiable, limitation of my method. Many of the most commonly occurring verbs will likely be ones that a dictionary user will have to know anyway - i.e. they will be in the prior knowledge, not the words whose meaning is deduced using the dictionary - and the meaning of many can be deduced without encountering any of the conjugation or plural problems. So, I don't think the problem is as bad as it sounds, but it's disconcerting nonetheless.

Applying this method to the example, one gets the following lists.

```
hundred a century county court for its large number of or own
    shire symbol ten the this

thousand for hundred large number symbol then this

billion a large million number thousand very
```

Equivalently, in visual form, one gets figure 1.

2.2 Graph search

As aforementioned, once the graph has been created, the task reduces to a topological sort. For acyclic graphs, this is easy - most computer science students will know how to solve this problem using a straightforward application of depth first search. However, there's no a priori reason to expect the dictionary graph is acyclic. In fact, as I discuss later, it's actually highly cyclic. Thus, a different approach is necessary.

I truly spent long and hard thinking about this problem, but made no meaningful progress. Given that I made some non-trivial assumptions in the data processing, I decided it was not essential to get an exact solution to the graph search problem. Instead, a good upper bound for the minimum will suffice as a significant first step towards settling my dictionary search conundrum. Abandoning exactness opened the door to one my favourite optimisation techniques, genetic algorithms - a beautiful technique for finding great solutions to diabolically hard optimisation problems without requiring or using any intelligence whatsoever.

In my case, the implementation worked as follows. I started with a population of p initial solutions. Each of these solutions was simply just a list of all the words in the dictionary. Hence, each is the worst possible solution; it says one has to know the meaning of every word

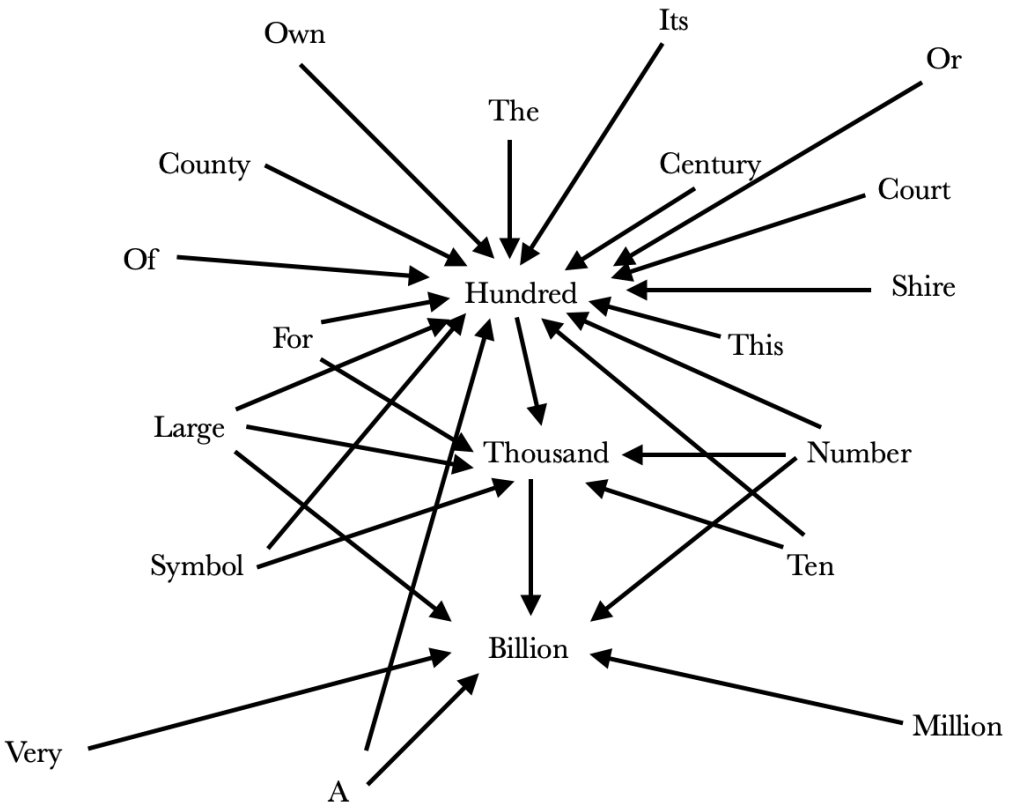

Figure 1: An excerpt of the dictionary graph, generated from the definitions of the words, "hundred," "thousand" and "billion."

in the dictionary to be able to work out the meanings of all the words - a tautology.

Next, for a given solution, I choose a word in that solution at random. If all the words in its definition appear in the solution, I remove the original word. One doesn't need to know the meaning of that word a priori; it can be worked out using the words I've assumed are known. I repeat this process n times, thereby potentially reducing the words that need to be known by up to n. Because I remove a word only if all its defining words are in the solution, I ensure the words left at the end of this process are sufficient to recover the meanings of all words in this dictionary. I apply this reduction method to all p solutions. Since the words are chosen from each solution at random, for some of the p, more words will be removed than for others.

Next, I impose the "selection pressure." In nature, the organisms best able to survive under the selection pressures of food, habitat, predators etc. are the ones that are most likely to live, reproduce and pass on their adaptations to their offspring. In my scenario, at the end of the reduction process, I rank the p solutions from shortest to longest and then cull the worst performing $p/2$ solutions. I replace these poorly performing solutions with "offspring" of the better performing solutions. I produce each of the $p/2$ offspring by randomly choosing two of the surviving solutions and defining their offspring to be the union of the set of words comprising each parent solution. Since the meaning of all words can be worked out from each of the two parent solutions, it can definitely be worked out from the union of those two solutions.

However, in nature, it isn't sufficient to simply have reproduction between the fitter organisms. Creating more sophisticated creatures requires mutations - chance changes to the DNA. If the mutation is unhelpful - like, say, cancer - then the individual dies and fails to pass on the

mutation to its offspring. If the mutation helps, then the individual is more likely to survive, pass on its wonderful new trait to its offspring, who also survive, reproduce, outlive inferior individuals without the mutation etc. I implemented mutations in my algorithm in the following way. I can't randomly remove any word, because for all I know that word might be essential to working out the meaning of some other word. But, I can randomly add words. The naive reader may believe this just makes the solution worse - after all, I'm trying to find the *minimum* number of words required to learn a language. But, I found adding random words can rescue an otherwise doomed solution. In the word removal process, I randomly choose a word and remove it if its defining words are in the solution. What if the first word I picked was "the?" Initially, since every word is in the solution, all of "the"'s defining words will be there. Hence, it gets removed. But, this is terrible for minimising the solution list - a simple word like "the" will be required to learn the meanings of many other words. Mutations thus serve to re-introduce such simple, essential words that were removed by chance. So, to each of the $p/2$ offspring solutions, I chose to add m words from the dictionary at random. As an aside, when I ran my algorithm without mutations, i.e. $m = 0$, the results were awful. The population got stuck on huge lists of words that couldn't be improved - in some sense terrible local minima. This was evidence that the dictionary graph is highly cyclic, with some very long loops.

At the end of this reproduction, I once again have p solutions - a 2nd "generation." I can now repeat the whole rigmarole again. I did that g times. After each of the g generations, the population of p solutions should get better and better on average and approach the optimum solution.

Any genetic algorithm has a number of hyperparameters - in my case, p, n, m and g - which can't be fixed in any canonical way. Someone blessed with more time than me would systematically search the parameter space and optimise those choices. However, in my experience, I find guess and check - the oldest and noblest of problem solving strategies, as Captain Jack Sparrow might call it - works sufficiently well. After some experimentation, I settled on $p = 1000$, $n = 1.5 \times \text{currentSolutionSize}$, $m = 130$ and $g = 60$. For g, the bigger the better, because the solutions improve over time; it's just a matter of choosing a g large enough that the improvement is negligible because the population is very close to the optimum solution.

3 Results

The complete dictionary had 29997 words. Figures 2, 3 and 4, show the size of the optimum solution after a given number of generations. Figure 2 contains the complete data. Because the trajectory is one of asymptotic decay, the data points become very cramped in later generations. Figures 3 and 4 plot only a subset of the data points to remedy this issue. In both figure 2 and 4, the curve appears to be reaching an asymptote. Therefore, my $g = 60$ choice is justified. The optimum solution is changing so little that it is unlikely letting the algorithm go on for many more generations would yield a significantly better outcome. A genetic algorithm, just like evolution in nature, can follow a patter of "punctuated equilibrium," so I can never be sure that there isn't a great improvement that would have been unearthed had I gone to $g > 60$. However, I have to draw the line somewhere and based on the data, I'm comfortable with my chosen cut-off.

The best solution I achieved was 2914 words. It was achieved after 58 generations. This finally answers my initial question - there are approximately 2914 words in the English language one has to already know the meanings of, to theoretically understand every other word

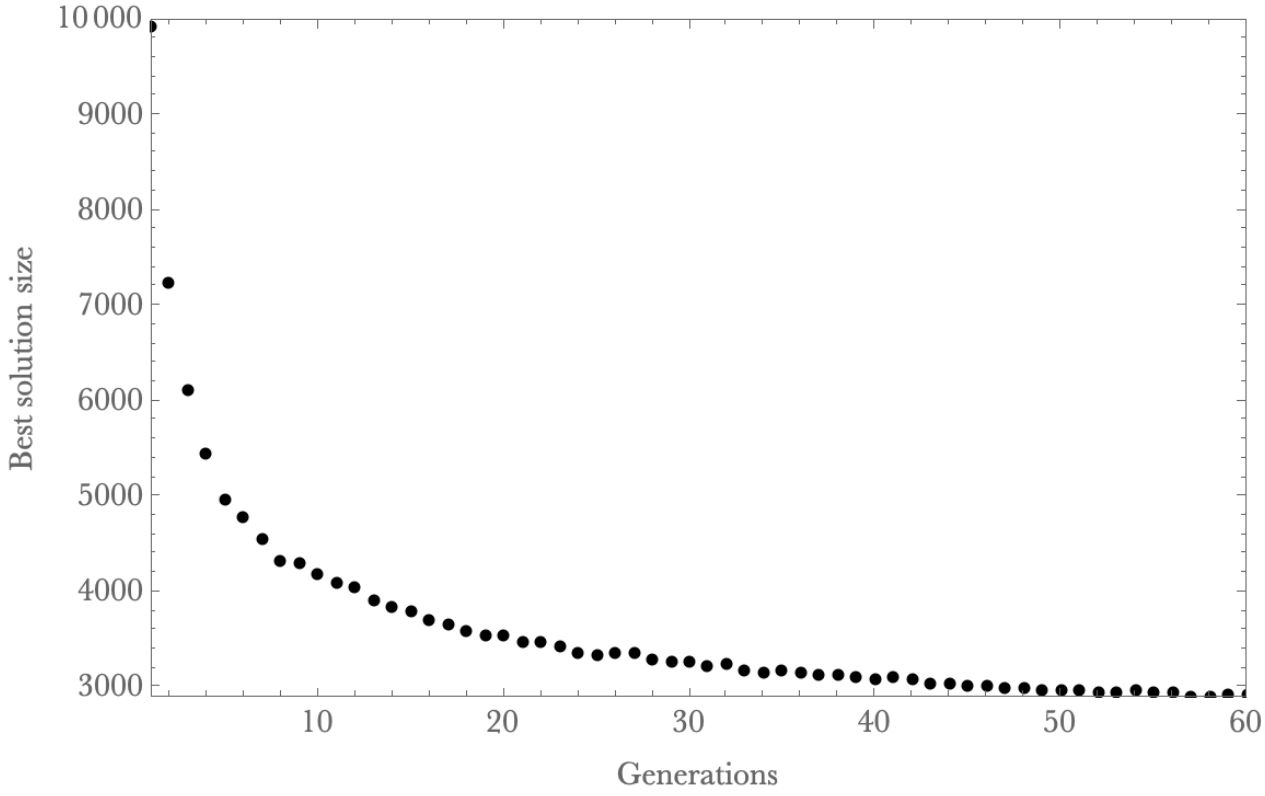

Figure 2: Best solution (i.e. least number of words required to be known) after a given number of generations - complete data

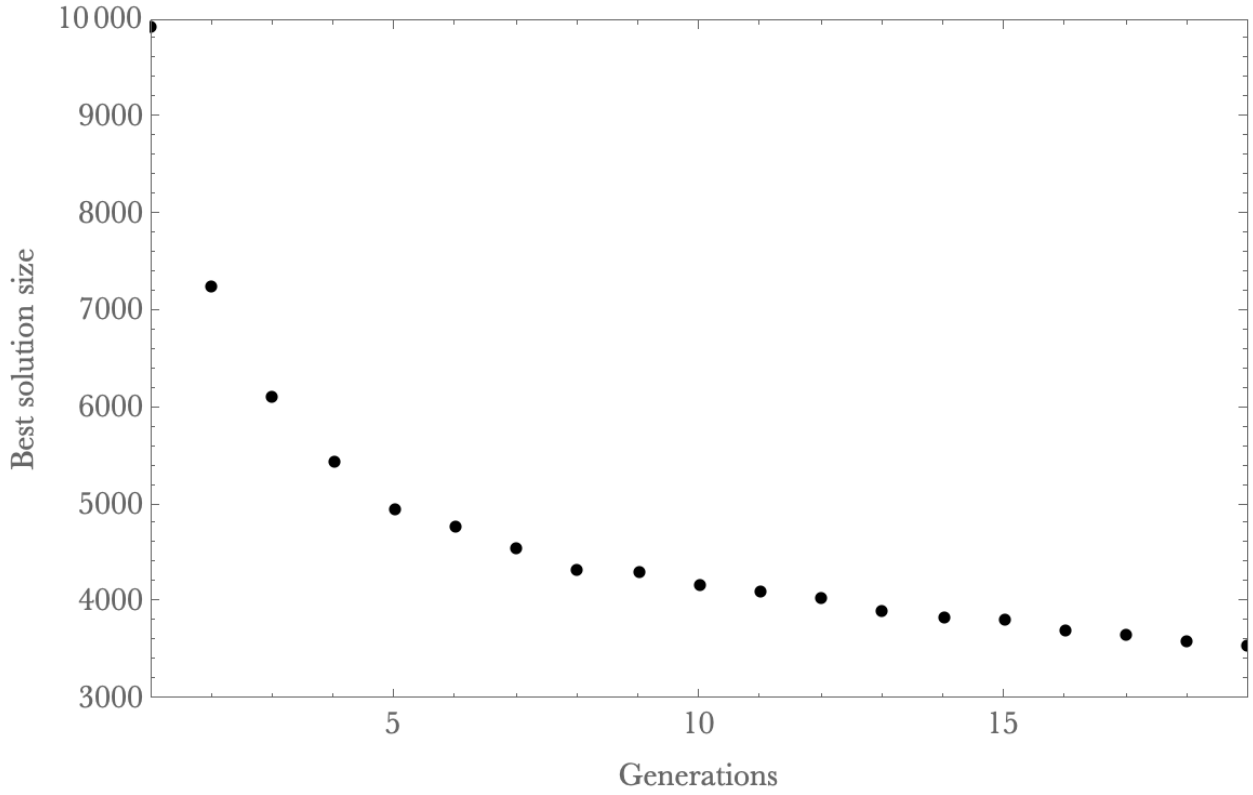

Figure 3: Best solution (i.e. least number of words required to be known) after a given number of generations - generations 1 to 20

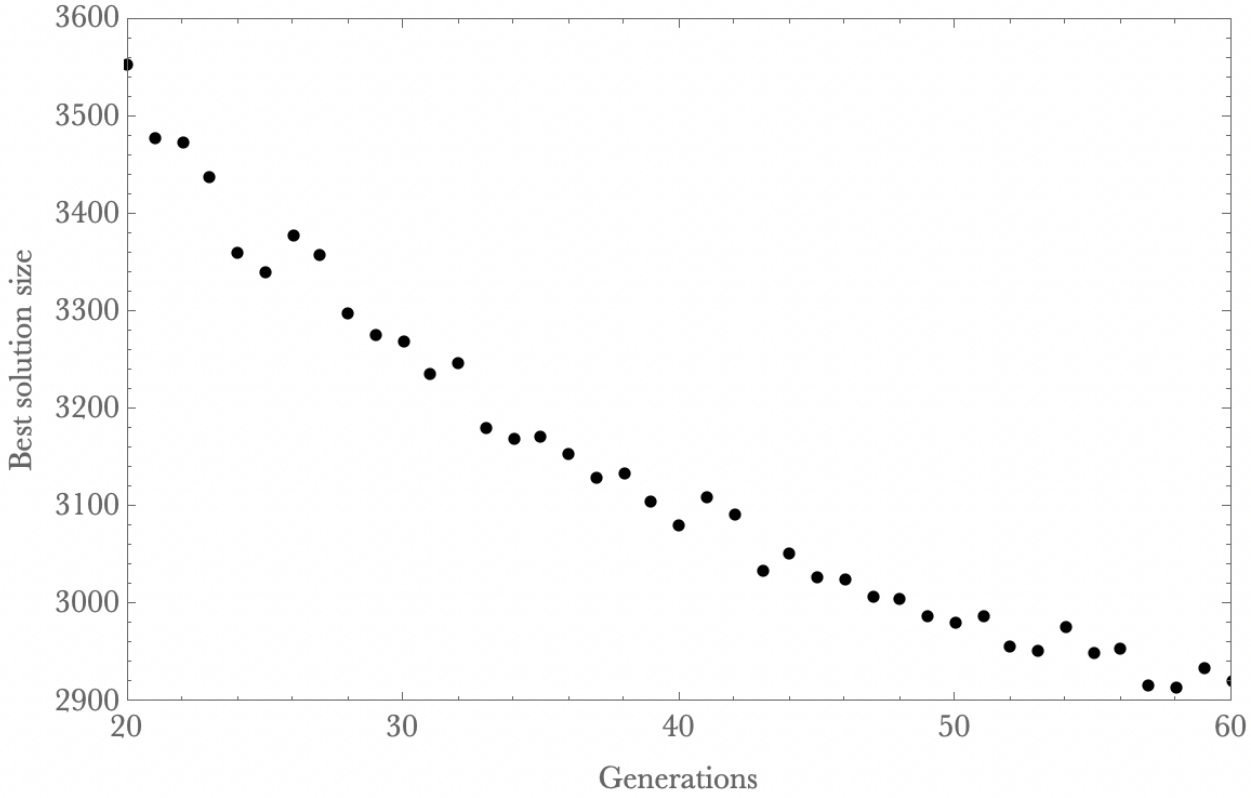

Figure 4: Best solution (i.e. least number of words required to be known) after a given number of generations - generations 21 to 60

with the aid of a dictionary. The complete list of 2914 words is available at https://github.com/VirinchiRallabhandi/DictionaryAnalysis, under the file name, minimalWords.txt.

4 Summary

The headline is that 2914 words is approximately all one needs to know to subsequently completely learn the English language. However, there are some caveats I'd like to recapitulate. Most importantly, my data processing threw out conjugated forms of verbs and plurals from definitions. As explained earlier, the effect may not be as dire as it sounds, but it is an unquantifiable error nonetheless. Correcting for this can only push the number, 2914, up. The other approximation comes from the genetic algorithm. A genetic algorithm continually improves a solution, but has no way of telling if the optimum has been found. While unlikely based on the data, it is possible running the algorithm for further generations would push the number, 2914, significantly down. Overall, my work produces an interesting, but approximate, answer. More sophisticated methods will be needed to find a more rigorous solution.