

Tushar Koushik

Mark LeBlanc

AI/ML

Anagram Report

Table of Contents

1. Summary
 2. What I Studied
 3. Step-by-Step Trie Diagrams for "Cat"
 4. Identified Issues and Limitations in the Code
 5. Debugging Process and Key Findings
 6. Conclusion
 7. My Version of Anagram Finder Made with the Help of GPT-4
-

1. Summary

Richard Neal's anagram finder is a Python program that efficiently generates anagrams using a **Trie data structure**. The program reads a dictionary file, constructs a Trie for fast word lookups, and then recursively generates anagrams using backtracking. It allows users to specify required and excluded words, as well as set a maximum number of words per anagram. The core strengths of this approach are **efficient word storage**, **fast search**, and **recursive traversal** to generate valid anagrams dynamically.

2. What I Studied

While analyzing Richard's code, I focused on the following aspects:

- **Trie Data Structure:** How words are stored letter-by-letter for fast retrieval.
- **Recursive Anagram Generation:** Using backtracking to explore valid word combinations.
- **Efficient Dictionary Handling:** Filtering words based on user-defined include/exclude lists.
- **Code Optimization & Readability:** Understanding how the code was structured and improving its documentation.

The study involved analyzing how the program builds a Trie dynamically and how it traverses the Trie to generate valid anagrams. To solidify my understanding, I created **step-by-step visualizations** of how the Trie structure is built for different words.

Additionally, a heavily commented version of Richard Neal's original code was created to better document its logic, structure, and potential areas for improvement. This **commented version** has been included in a ZIP file attached to this report for reference. Readers can refer to it for a deeper understanding of the inner workings of the code.

3. Step-by-Step Trie Diagrams for my favourite anagram of "Cat"

Below are the **stepwise visualizations** of how the Trie structure is built for the word "cat" and its valid anagrams:

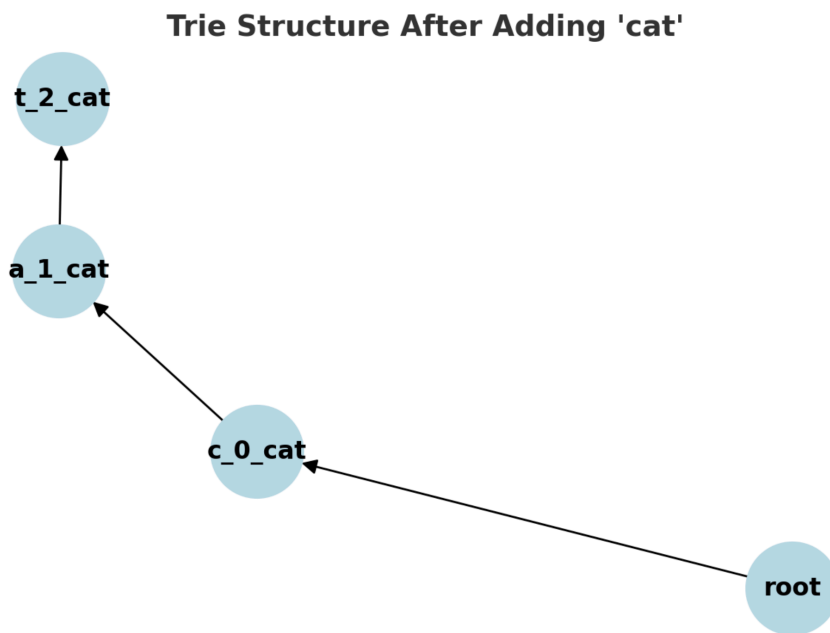


Diagram 1: Adding "cat" to the Trie

Diagram 1 represents the **initial insertion of "cat"** into the Trie. Each letter is added sequentially:

- 'c' is added as the first child node.
- 'a' is added as a child of 'c'.
- 't' is added as a child of 'a', marking the completion of the word "cat".
- The Trie structure now contains "cat" as a valid word.

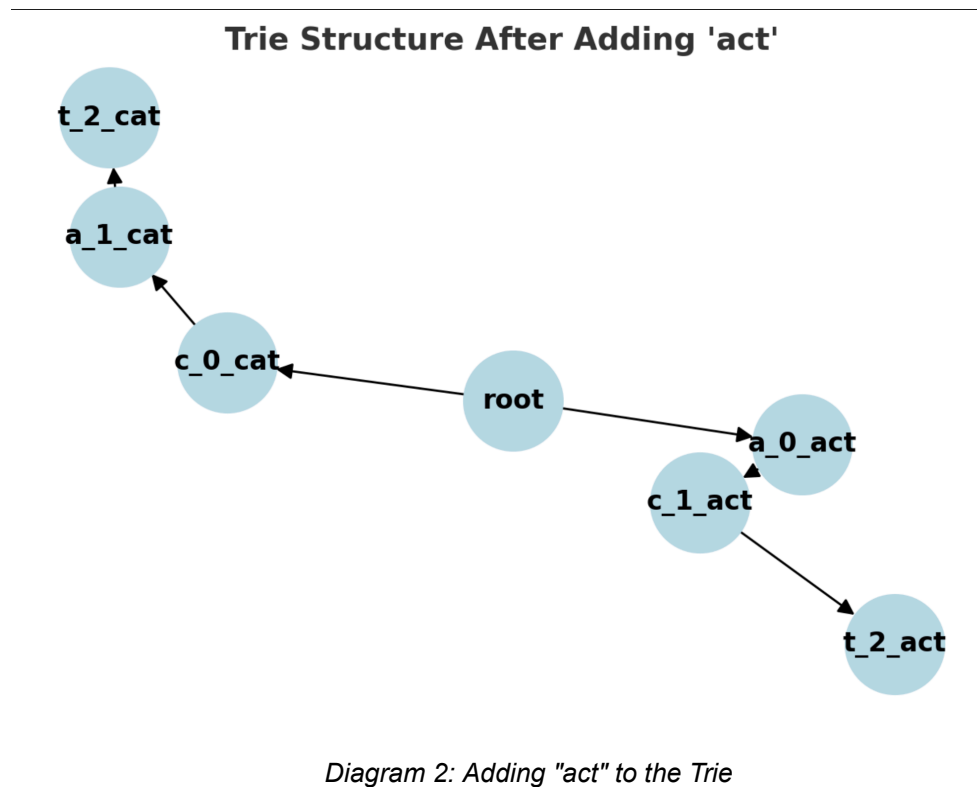


Diagram 2 shows the **insertion of "act"** into the existing Trie:

- The root already contains 'c' from "cat", but "act" starts with 'a', so a new branch is created from the root.
- 'a' is added as a new node.
- 'c' follows as a child of 'a'.
- 't' is added as a child of 'c', forming "act" as a valid word.
- The Trie now holds both "cat" and "act" as separate branches.

Trie Structure After Adding 'tac'

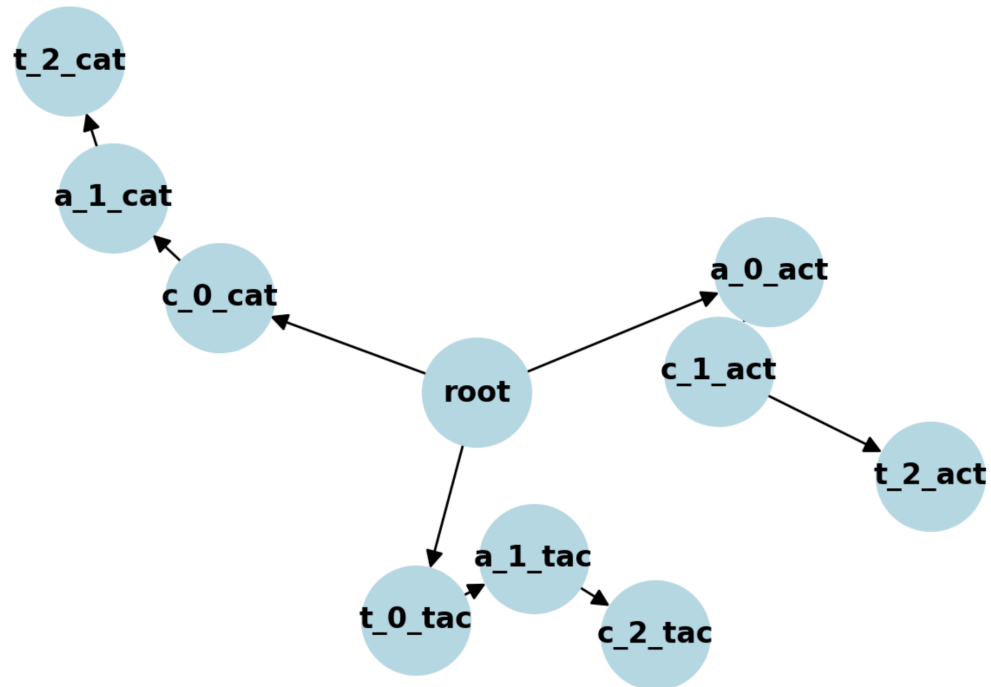


Diagram 3: Adding "tac" to the Trie

Diagram 3 represents the **insertion of "tac"**:

- Since "tac" starts with 't', a new branch is added from the root.
- 't' is added as a new node.
- 'a' is inserted as a child of 't'.
- 'c' is added as a child of 'a', completing "tac".
- The Trie now contains "cat", "act", and "tac", forming three unique branches.

Trie Structure After Adding 'tac'

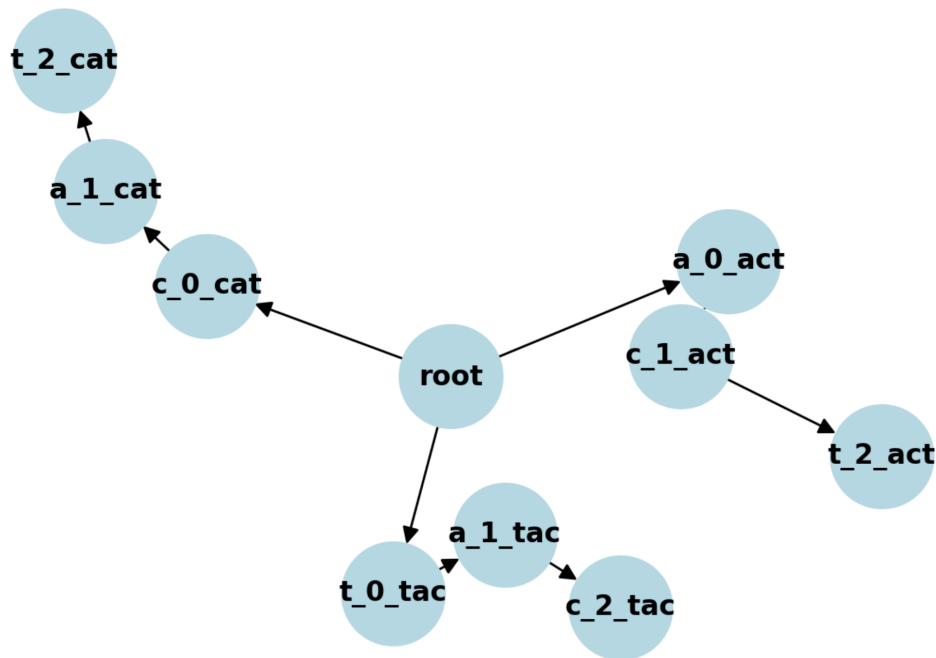


Diagram 4: Complete Trie for "cat" and its anagrams

This final Diagram 4 shows the **fully constructed Trie** containing all anagrams: "cat", "act", and "tac"; other anagrams such as "at c" are found at the path while going back.

- Each word is stored in a unique path within the Trie.
- The Trie structure efficiently allows searching for anagrams by exploring valid letter paths.
- With this structure, the program can recursively generate all possible valid anagrams of "cat" by traversing the Trie.

These descriptions **explain the evolution of the Trie step by step**, highlighting how each word gets inserted and how the structure facilitates anagram generation. Each word follows a **prefix-based insertion** in the Trie, allowing efficient lookups and recursion-based anagram formation.

4. Identified Issues and Limitations in the Code

While studying Richard's code, several areas for improvement were identified:

1. Lack of Type Hinting

- The code does not use **Python type hints**, making it harder to understand function inputs/outputs at a glance.
- **Fix:** Adding type hints would improve readability and maintainability.

2. Cannot Use External Word Libraries

- The program requires a dictionary file but does not allow users to use **NLTK, SpaCy, or other linguistic libraries** dynamically.
- **Fix:** Adding an option to load words from Python libraries would enhance flexibility.

3. No Handling for Empty or Invalid Inputs

- If the user enters an empty string or a word not in the dictionary, the program does not handle it gracefully.
- **Fix:** Adding better error handling and fallback options.

4. Recursive Backtracking Can Be Slow for Large Inputs

- The recursive function explores **all** possible word combinations, which can be inefficient for long input words.
- **Fix:** Implementing **memoization** or **dynamic programming** techniques to optimize redundant calls.

5. Debugging Process and Key Findings

A thorough debugging process was conducted to understand the code's execution flow, identify inefficiencies, and analyze how different components interact. The primary purpose of debugging was to track how words are inserted into the Trie, how recursive calls function during anagram generation, and how the program handles various edge cases.

Approach to Debugging

The debugging process involved setting breakpoints and stepping through the code to examine key operations:

- **Trie Insertion Debugging:** A breakpoint was set in the `insert()` function of the `Trie` class to track how words are added letter by letter.
- **Recursive Anagram Generation Debugging:** Debugging focused on verifying how the function `anagramization()` iterates through letters and forms valid anagrams.
- **Variable State Inspection:** The debugger was used to observe changes in variables such as `children`, `leafiness`, and `progress` to confirm proper data flow.
- **Error Handling Debugging:** Edge cases such as invalid inputs and missing dictionary files were tested to examine how the program responds.

Through this debugging process, it was possible to visualize the exact steps the program follows when generating anagrams, uncover inefficiencies in recursion, and validate the correctness of the Trie construction.

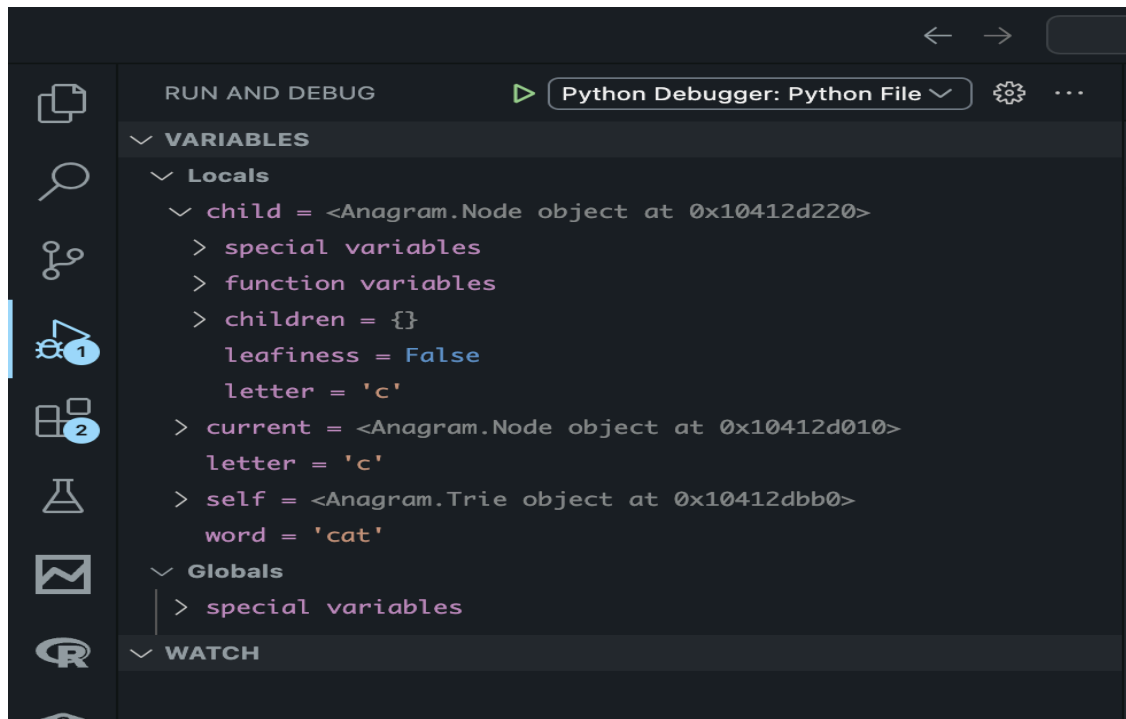


Diagram 5: Debugging Trie Insertion - Showing variable states during insertion process

Diagram 5 captures a **debugging session in VS Code** while stepping through the `insert()` function of the `Trie` class. The debugger stopped inside the function, showing the **state of key variables** during word insertion.

Key Observations from the Debugger:

- `word = 'cat'`: The program is currently inserting the word "cat" into the Trie.
- `letter = 'c'`: The debugger is handling the **first letter** of "cat".
- `current = <Anagram.Node object>`: The current Trie node represents 'c'.
- `child = <Anagram.Node object>`: A **new node is being created** for 'c'.
- `children = {}`: At this moment, the Trie node **does not yet have any children**, meaning 'c' is being added for the first time.
- `leafiness = False`: The node is **not yet marked as a leaf**, meaning the word is still being processed.

Debugging Insights

- This confirms that Trie nodes are being **added letter by letter**.

- The *children* dictionary starts empty and gets updated dynamically.
 - The process successfully **creates new nodes for each unique letter**.
-

6. Conclusion

Richard Neal's code is a **well-structured** and **efficient** implementation of an anagram finder using a Trie. By studying its **Trie-based word storage**, **recursive anagram formation**, and **optimization strategies**, deeper insights were gained into **data structures and algorithm design**. While the code is effective, improvements such as **adding type hints**, **allowing external libraries**, **improving error handling**, and **optimizing recursion** could make it even more robust and user-friendly.

This study helped in understanding both **theoretical and practical applications** of Tries and recursion in natural language processing tasks.

7. My Version of Anagram Finder Made with the Help of GPT-4

This version of the Anagram Finder is developed with significant improvements to enhance efficiency, flexibility, and user-friendliness. The updated approach incorporates modern best practices to ensure optimal performance and usability.

Key Improvements in This Version

- **Type Hinting for Readability:** Uses Python type hints to make the code clearer and easier to maintain.
- **Support for External Word Libraries:** Allows users to load words dynamically from the NLTK words corpus or a custom dictionary file.
- **Optimized Recursive Search:** Implements memoization to avoid redundant computations and improve performance.
- **Prevents Infinite Recursion:** Ensures that recursive calls only happen when the problem size is actually reducing, preventing infinite loops.
- **Better Error Handling:** Gracefully handles invalid inputs and missing dictionary files to ensure a smooth user experience.
- **Interactive Command-Line Interface:** The script prompts users for input, making it easy to use without requiring manual code modifications.

This version uses Python's built-in data structures like `set` and `Counter` to efficiently store and validate words, improving lookup speed while maintaining the ability to dynamically generate valid anagrams. The program also supports user-defined dictionaries, making it adaptable for different linguistic applications.

By addressing the limitations of the previous implementation, this enhanced version serves as a reliable and efficient anagram generator.

Additional Documentation

A detailed README is included in the project, explaining:

- Installation steps
- Usage instructions
- File descriptions
- Code improvements

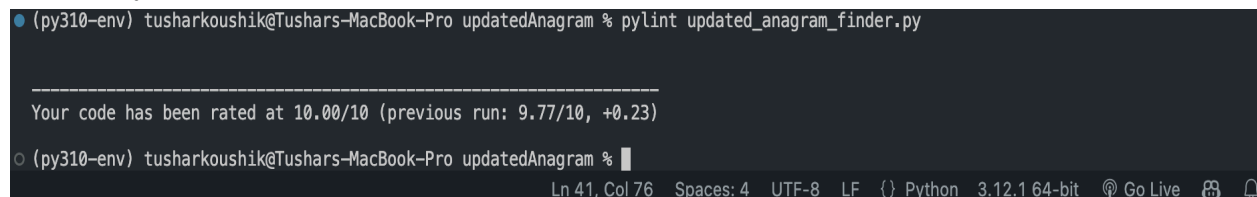
This version of the Anagram Finder is designed to be efficient, easy to use, and adaptable for various applications.

PyLint Score

Final Score: 10.00/10

No errors or warnings.

Follows Python best practices (PEP 8).



```
(py310-env) tusharkoushik@Tushars-MacBook-Pro updatedAnagram % pylint updated_anagram_finder.py

-----
Your code has been rated at 10.00/10 (previous run: 9.77/10, +0.23)
(py310-env) tusharkoushik@Tushars-MacBook-Pro updatedAnagram %
```

Ln 41, Col 76 Spaces: 4 UTF-8 LF {} Python 3.12.1 64-bit Go Live

*Citation : OpenAI. (2024). ChatGPT-4o - AI Language Model. Retrieved from <https://openai.com/>
Developed with the support of OpenAI's ChatGPT-4o for debugging and optimizations.*