# Graph-Based Word Ladder

Presented By:
- Waniya Badar (22k-4526)
- Nimil Zubair (22k-4617)
- Alishba Hassan (22k-4333)

# Contents

# Introduction

**Overview:**

- Implement a Word Ladder game solver using graph theory.
- Compare Python and MATLAB for performance and efficiency.

**Objectives:**

- Build a graph representation of words.
- Use BFS for shortest path and explore longest word ladders.
- Analyze results, performance, and provide insights.

# Implementation Plan

**Graph Construction:**

- Load word list and connect words differing by one letter.

**Algorithm:**

- Use BFS for shortest paths and backtracking for longest paths.

**Optimizations:**

- Minimize graph size by preprocessing.
- Explore alternatives for faster traversal.

# Working of Code

1. Mechanism of Word Transformation:
    - Starting from the start word, each word is modified by changing one letter at a time to form new words.
    - Example: From hit, modify one letter:
        - h**o**t, h**a**t, etc.
    - Only valid words from the dictionary (D) are considered for further exploration.
2. Shortest Path Function (shortestChainLenAndPath):
    - Uses Breadth-First Search (BFS) to find the shortest chain.
    - Tracks the level of each transformation and stops at the first valid target word.
3. Longest Path Function (longestChainLenAndPath):
    - Also uses BFS but explores all valid paths.
    - Keeps track of the longest valid transformation chain by continuing exploration even after reaching the target.
4. Visualization:
    - A directed graph is created with nodes as words and edges showing valid transformations.

# Python Code:

```
import time
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque

# Function to visualize the word graph with arrows and start/stop markers
def visualize_graph(paths, start, target):
    G = nx.DiGraph()  # Directed graph to show the direction of transformations

    # Add nodes and edges
    for path in paths:
        for i in range(len(path) - 1):
            G.add_edge(path[i], path[i + 1])

    # Draw the graph
    pos = nx.spring_layout(G)  # Positions for all nodes
    node_color = ['lightgreen' if node == start else 'lightcoral' if node == target else 'lightblue' for node in G.nodes()]
    edge_color = 'gray'

    # Draw nodes, edges, and labels
    nx.draw(G, pos, with_labels=True, node_color=node_color, edge_color=edge_color, font_size=10,
font_weight='bold', arrows=True)

    # Show arrows on the graph
    nx.draw_networkx_edge_labels(G, pos, edge_labels={(u, v): '->' for u, v in G.edges()}, font_size=8)

    # Show plot
    plt.show()
```

```
# Function to find the longest chain length and paths
def longestChainLenAndPath(start, target, D):
    if start == target:
        return 0, [start]

    if target not in D:
        return 0, []

    word_length = len(start)
    D = set(D)  # Convert D to a set for faster lookup

    # Queue to store paths (each item in the queue is a tuple: (current_word, path_so_far))
    Q = deque([(start, [start])])

    # Track the longest path found
    longest_path = []

    while Q:
        current_word, path = Q.popleft()
        word = list(current_word)

        # For every character of the word
        for pos in range(word_length):
            original_char = word[pos]

            # Try changing the character to every letter from 'a' to 'z'
            for c in range(ord('a'), ord('z') + 1):
                word[pos] = chr(c)
                new_word = "".join(word)
```

# Python Code:

```
 # If the new word is the target, check if the path is the longest
        if new_word == target:
          candidate_path = path + [new_word]
          if len(candidate_path) > len(longest_path):
            longest_path = candidate_path

        # If the new word is in the dictionary and hasn't been used in this path
        elif new_word in D and new_word not in path:
          Q.append((new_word, path + [new_word]))

      # Restore the original character
      word[pos] = original_char

  return len(longest_path), longest_path

# Function to find the shortest chain length and paths
def shortestChainLenAndPath(start, target, D):
  if start == target:
    return 0, [start]

  if target not in D:
    return 0, []

  level = 0
  word_length = len(start)

  # Queue to store the word and its path
  Q = deque()
  Q.append((start, [start]))

  # To store all shortest paths found
  shortest_paths = []
  min_len = float('inf')

  # While the queue is non-empty
  while Q:
    level += 1
    sizeofQ = len(Q)

    for _ in range(sizeofQ):
      current_word, path = Q.popleft()
      word = list(current_word)

      # For every character of the word
      for pos in range(word_length):
        original_char = word[pos]

        # Replace the current character with every possible alphabet
        for c in range(ord('a'), ord('z') + 1):
          word[pos] = chr(c)
          new_word = "".join(word)

          # If the new word is the target, return the result
          if new_word == target:
            candidate_path = path + [new_word]
            if level + 1 < min_len:
              shortest_paths = [candidate_path]
              min_len = level + 1
            elif level + 1 == min_len:
              shortest_paths.append(candidate_path)

          # If the new word exists in the dictionary and hasn't been visited
          if new_word in D:
            D.remove(new_word) # Mark word as visited
            Q.append((new_word, path + [new_word]))

        # Restore the original character
        word[pos] = original_char

  return min_len, shortest_paths
```

# Python Code:

```python
# Driver code
if _name_ == "_main_":
  # Set of valid words
  D = {
    "hit", "hot", "dot", "dog", "cog", "lot", "log", "hip", "hop", "top", "lop", "bot", "pot", "cop", "cot"
  }
  start = "hit"
  target = "cog"

  # Timing the longest chain search
  start_time = time.time()
  longest_length, longest_paths = longestChainLenAndPath(start, target, D)
  end_time = time.time()
  longest_time = end_time - start_time

  # Timing the shortest chain search
  start_time = time.time()
  shortest_length, shortest_paths = shortestChainLenAndPath(start, target, D.copy())
  end_time = time.time()
  shortest_time = end_time - start_time

  # Visualize the graph for shortest paths
  if shortest_paths:
    print("\nVisualizing the shortest paths graph...")
    visualize_graph(shortest_paths, start, target)
```

```python
# Print results
  print("Length of shortest chain is:", shortest_length)
  if not shortest_paths:
     print("No shortest path found")
  else:
     print("Shortest paths are:")
     for path in shortest_paths:
        print(" -> ".join(path))
  print("Time taken for shortest path:", shortest_time, "seconds")

  print("\nLength of longest chain is:", longest_length)
  if not longest_paths:
     print("No longest path found")
  else:
     print("Longest paths are:")
     for path in longest_paths:
        if path == longest_paths[-1]:
           print(path)
        else:
           print(path, end=" -> ")
  print("Time taken for longest path:", longest_time, "seconds")
```

# MATLAB Code:

```
function visualizeGraph(paths, startWord, targetWord)
    % Create a directed graph (digraph) instead of undirected graph
    G = digraph();

    % Add nodes and edges to the graph based on the paths
    for i = 1:length(paths)
        path = paths{i};
        for j = 1:length(path)-1
            % Add directed edges from path{j} to path{j+1}
            G = addedge(G, path{j}, path{j+1});
        end
    end

    % Plot the graph with directed edges
    figure;
    h = plot(G, 'Layout', 'force', 'NodeLabel', G.Nodes.Name, 'ArrowSize', 10);

    % Color the start and target nodes differently
    highlightStart = findnode(G, startWord);
    highlightTarget = findnode(G, targetWord);
    h.NodeColor = repmat([0.7 0.7 0.7], size(G.Nodes, 1), 1); % Default node color
    h.NodeColor(highlightStart, :) = [0.1 0.8 0.1]; % Start node in green
    h.NodeColor(highlightTarget, :) = [0.8 0.1 0.1]; % Target node in red
end



% Function to find the longest chain length and paths
function [longestLength, longestPaths] = longestChainLenAndPath(startWord, targetWord, D)
    longestLength = 0;
    longestPaths = {};  % Initialize to avoid unused variable warning

    if strcmp(startWord, targetWord)
        longestLength = 0;
        longestPaths = {startWord};
        return;
    end

    if ~ismember(targetWord, D)
        longestLength = 0;
        longestPaths = {};
        return;
    end

    wordLength = length(startWord);
    D = unique(D); % Ensure dictionary is unique
```

```
% Queue to store paths (currentWord, pathSoFar)
    Q = {startWord, {startWord}};
    longestPaths = {};

    while ~isempty(Q)
        [currentWord, path] = Q{1, :}; % Dequeue the first element
        Q(1, :) = [];

        for pos = 1:wordLength
            originalChar = currentWord(pos);

            % Try changing the character at each position to every letter 'a' to 'z'
            for c = 'a':'z'
                newWord = currentWord;
                newWord(pos) = c;

                if strcmp(newWord, targetWord)
                    candidatePath = [path, newWord];
                    if length(candidatePath) > longestLength
                        longestLength = length(candidatePath);
                        longestPaths = {candidatePath};
                    end
                elseif ismember(newWord, D) && ~ismember(newWord, path)
                    Q = [Q; {newWord, [path, newWord]}];
                end
            end
        end
    end
end

% Function to find the shortest chain length and paths
function [shortestLength, shortestPaths] = shortestChainLenAndPath(startWord, targetWord, D)
    shortestLength = inf;  % Initialize to avoid unused variable warning
    shortestPaths = {};    % Initialize to avoid unused variable warning

    if strcmp(startWord, targetWord)
        shortestLength = 0;
        shortestPaths = {startWord};
        return;
    end

    if ~ismember(targetWord, D)
        shortestLength = 0;
        shortestPaths = {};
        return;
    end

    wordLength = length(startWord);
    Q = {startWord, {startWord}};
    visited = containers.Map();
```

# MATLAB Code:

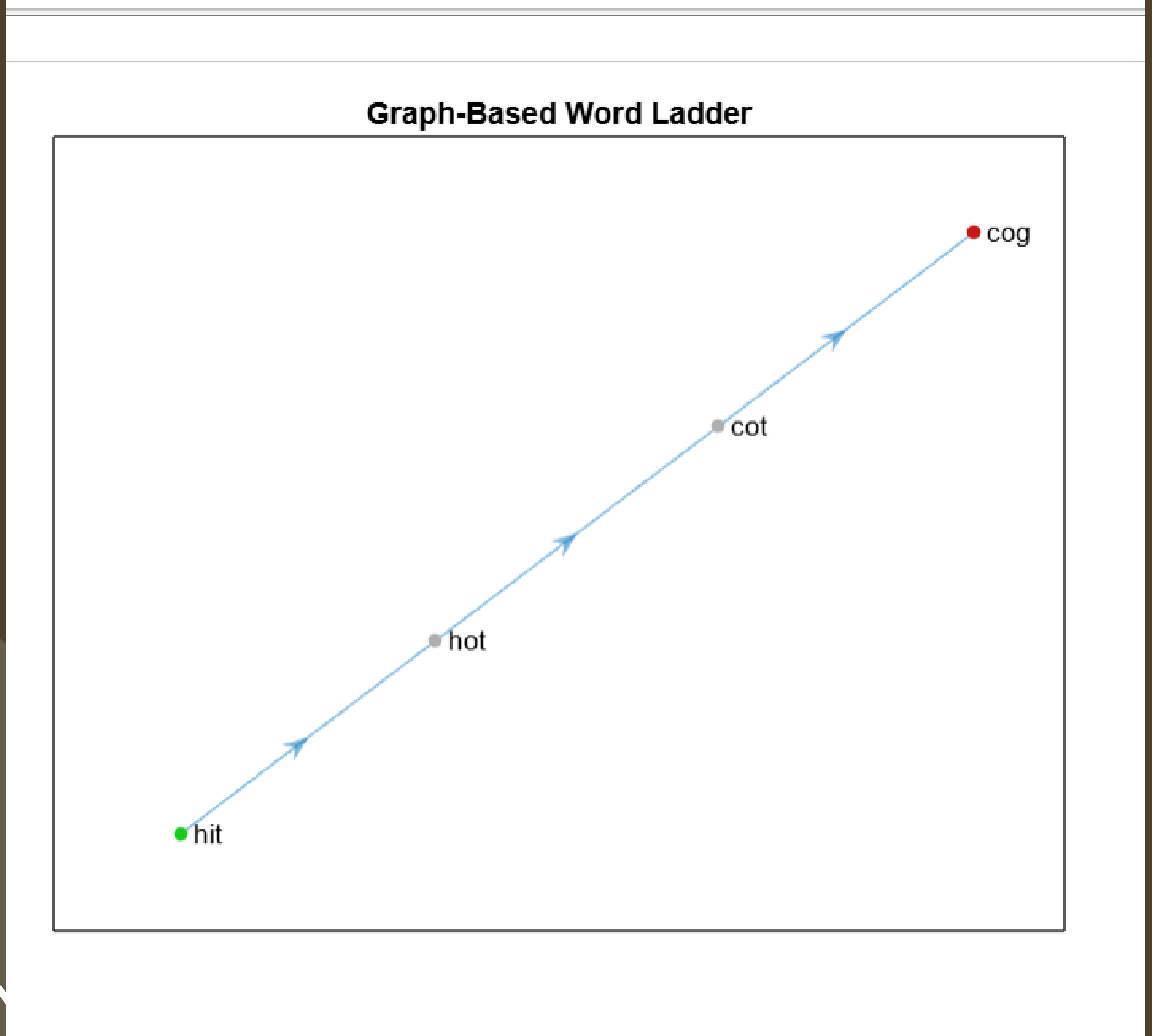
```
while ~isempty(Q)
    level = size(Q, 1);

    for i = 1:level
      [currentWord, path] = Q{1, :};
      Q(1, :) = []; % Dequeue the first element

      for pos = 1:wordLength
        originalChar = currentWord(pos);

        % Try changing the character to every letter 'a' to 'z'
        for c = 'a':'z'
          newWord = currentWord;
          newWord(pos) = c;

          if strcmp(newWord, targetWord)
            candidatePath = [path, newWord];
            if length(candidatePath) < shortestLength
              shortestLength = length(candidatePath);
              shortestPaths = {candidatePath};  % Wrap in a cell
            elseif length(candidatePath) == shortestLength
              shortestPaths{end+1} = candidatePath;  % Append to cell array
            end
          elseif ismember(newWord, D) && ~isKey(visited, newWord)
            visited(newWord) = true;
            Q = [Q; {newWord, [path, newWord]}];
          end
        end
      end
    end
  end
end

% Main function to run the example and compare shortest/longest paths
function wordTransformationComparison()
 % Dictionary of words
 D = {'hit', 'hot', 'dot', 'dog', 'cog', 'lot', 'log', 'hip', 'hop', 'top', 'lop', 'bot', 'pot', 'cop', 'cot'};
 startWord = 'hit';
 targetWord = 'cog';

  % Measure time for longest chain calculation
  tic; % Start timer
  [longestLength, longestPaths] = longestChainLenAndPath(startWord, targetWord, D);
  longestTime = toc; % Stop timer

  % Measure time for shortest chain calculation
  tic; % Start timer
  [shortestLength, shortestPaths] = shortestChainLenAndPath(startWord, targetWord, D);
  shortestTime = toc; % Stop timer

  % Display Results
  fprintf('\n--- Comparative Analysis ---\n');
  fprintf('\nExecution Time (MATLAB):\n');
  fprintf('Time for finding longest chain: %.6f seconds\n', longestTime);
  fprintf('Time for finding shortest chain: %.6f seconds\n', shortestTime);

  % Visualize the graph for the shortest paths
  if ~isempty(shortestPaths)
      disp('Visualizing the shortest paths graph...');
      visualizeGraph(shortestPaths, startWord, targetWord);
  end

  % Print results for shortest and longest chains
  fprintf('\nLength of shortest chain is: %d\n', shortestLength);
  if isempty(shortestPaths)
      disp('No shortest path found');
  else
      disp('Shortest paths are:');
      for i = 1:length(shortestPaths)
          disp(strjoin(shortestPaths{i}, ' -> '));
      end
  end

  fprintf('\nLength of longest chain is: %d\n', longestLength);
  if isempty(longestPaths)
      disp('No longest path found');
  else
      disp('Longest paths are:');
      for i = 1:length(longestPaths)
          disp(strjoin(longestPaths{i}, ' -> '));
      end
  end
end

% Call the main function to run the transformation and perform comparative analysis
wordTransformationComparison();
```

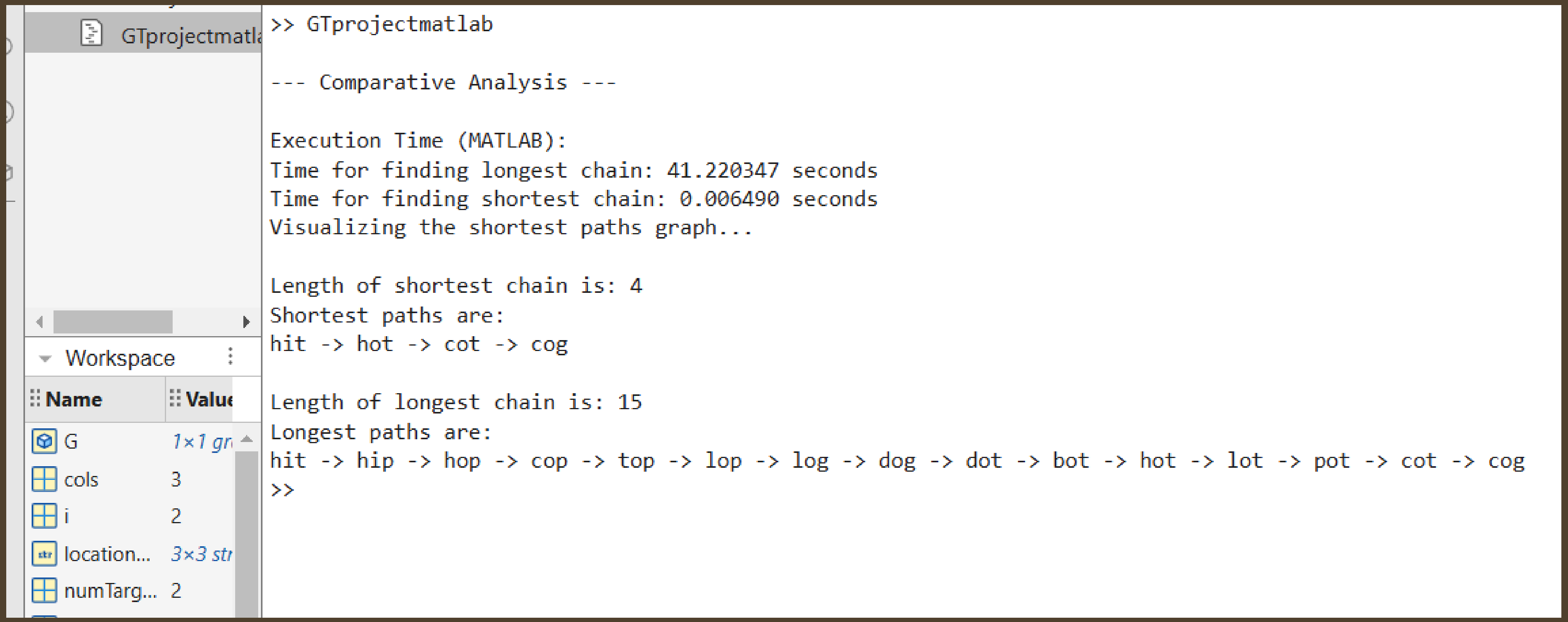
# Test Cases and Results

## 1. Valid Test Case

- **Start Word:** hit
- **Target Word:** cog
- **Word Set:** {'hit', 'hot', 'dot', 'dog', 'cog', 'lot', 'log', 'hip', 'hop', 'top', 'lop', 'bot', 'pot', 'cop', 'cot'}

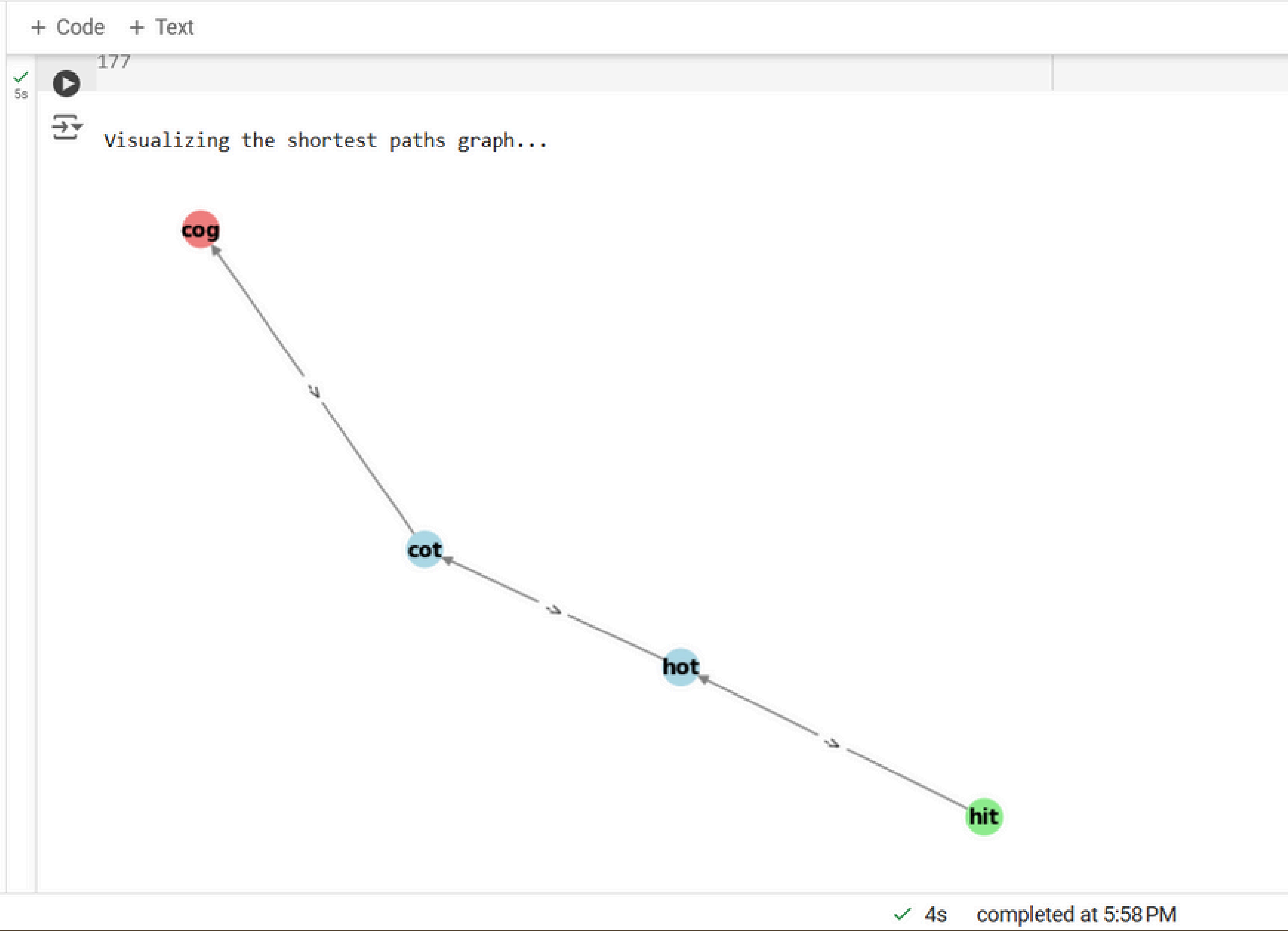
## MATLAB Results:



# Test Cases and Results

## Python Results:



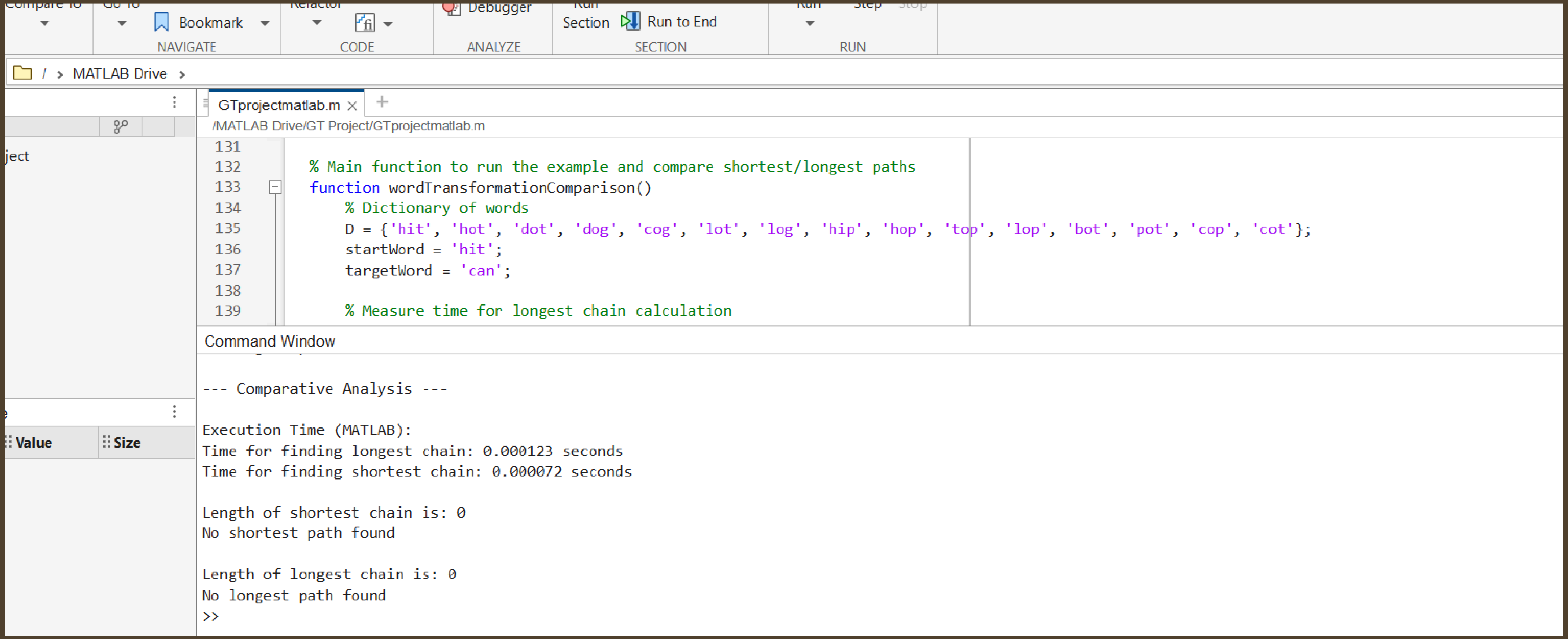

```
Length of shortest chain is: 4
Shortest paths are:
hit -> hot -> cot -> cog
Time taken for shortest path: 0.00039958953857421875 seconds

Length of longest chain is: 15
Longest paths are:
hit -> hip -> hop -> cop -> top -> lop -> log -> dog -> dot -> bot -> hot -> lot -> pot -> cot -> cog
Time taken for longest path: 0.8962106704711914 seconds
```

# Test Cases and Results

## 2. Invalid Test Case

- **Start Word:** hit
- **Target Word:** can
- **Word Set:**{'hit', 'hot', 'dot', 'dog', 'cog', 'lot', 'log', 'hip', 'hop', 'top', 'lop', 'bot', 'pot', 'cop', 'cot'}

## MATLAB Results:



# Test Cases and Results

## Python Results:

```python
        print("No longest path found")
    else:
        print("Longest paths are:")
        for path in longest_paths:
            if path == longest_paths[-1]:
                print(path)
            else:
                print(path, end=" -> ")
    print("Time taken for longest path:", longest_time, "seconds")
```

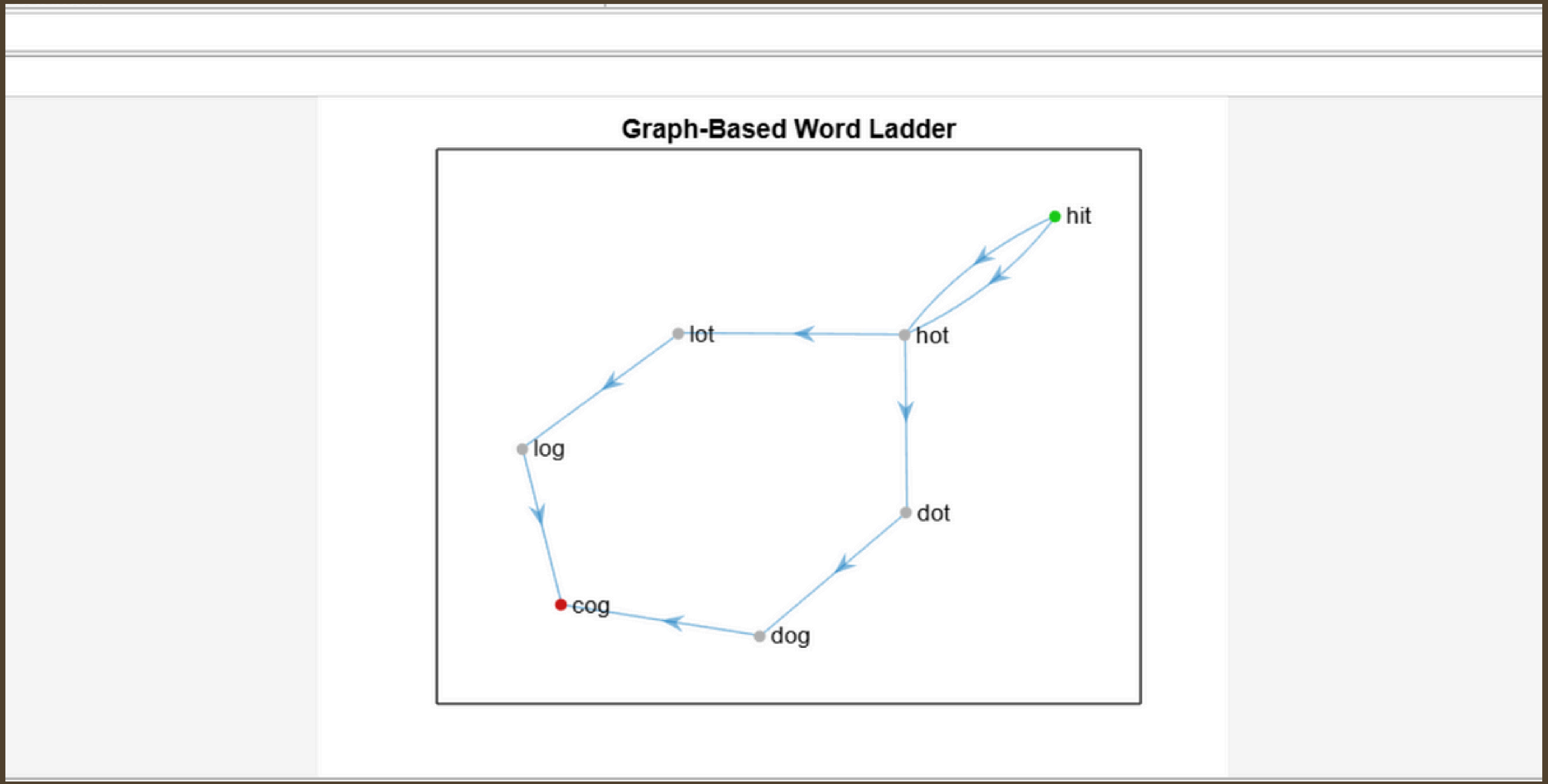
```
Length of shortest chain is: 0
No shortest path found
Time taken for shortest path: 3.0994415283203125e-06 seconds

Length of longest chain is: 0
No longest path found
Time taken for longest path: 2.384185791015625e-06 seconds
```

# Test Cases and Results

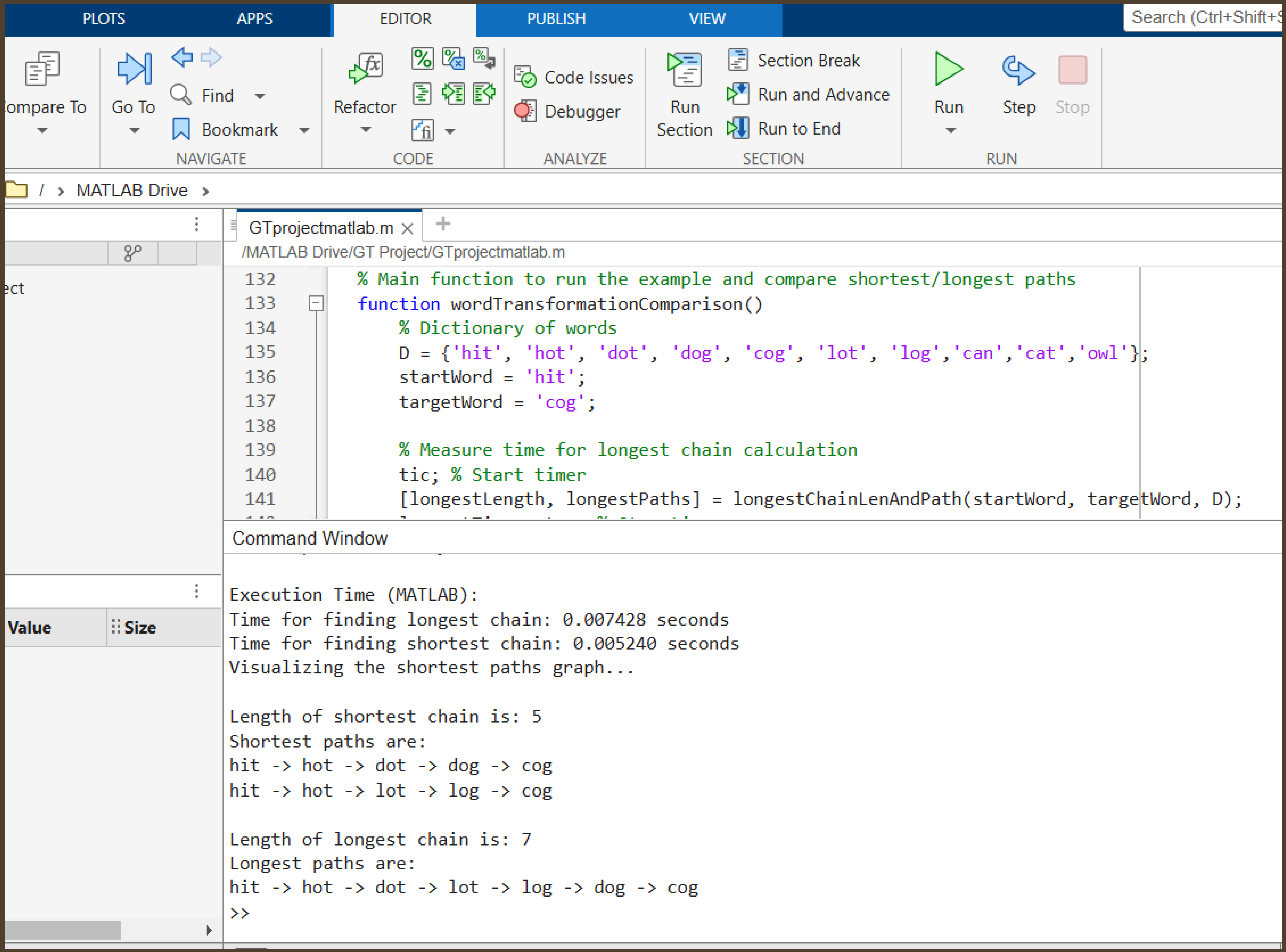
**3. Multiple Possible Paths Test Case**

- **Start Word: hit**
- **Target Word:** cog
- **Word Set:** {'hit', 'hot', 'dot', 'dog', 'cog', 'lot', 'log', 'can', 'cat', 'owl'}

**MATLAB Results:**

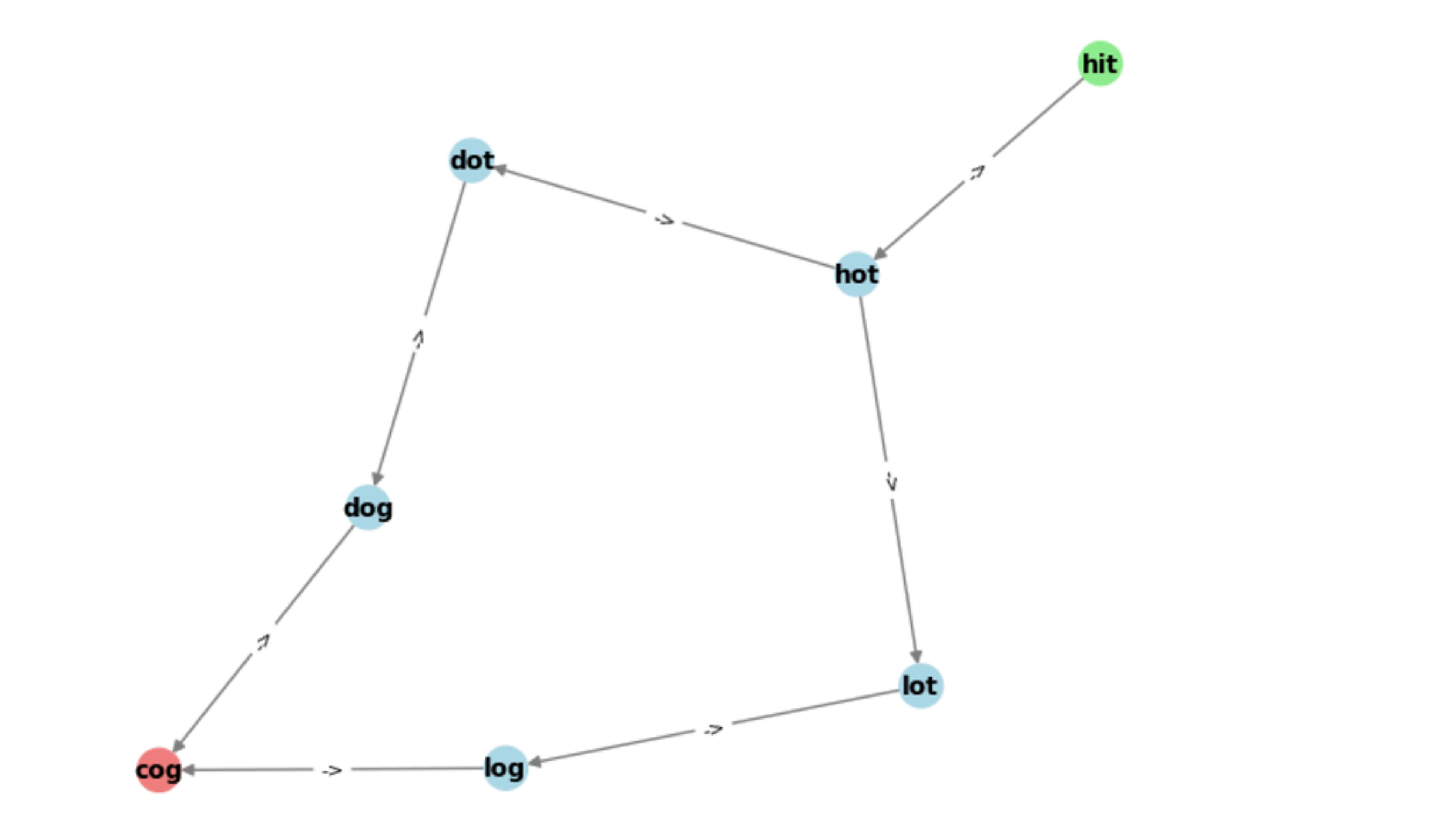


# Test Cases and Results



# Test Cases and Results

## Python Results:




```
Length of shortest chain is: 5
Shortest paths are:
hit -> hot -> dot -> dog -> cog
hit -> hot -> lot -> log -> cog
Time taken for shortest path: 0.00019216537475585938 seconds

Length of longest chain is: 7
Longest paths are:
hit -> hot -> dot -> lot -> log -> dog -> cog
Time taken for longest path: 0.000392913818359375 seconds
```

# Performance Analysis

| Metric | MATLAB | Python |
| --- | --- | --- |
| **Shortest Path** | Accurate and efficient. Faster for smaller graphs. | Accurate and faster for all scenarios. |
| **Longest Path** | Accurate but slow for larger graphs due to exhaustive search. | More optimized in comparison to MATLAB. |
| **Execution Time** | Higher due to inherent overhead in graph traversal. | Generally faster with better handling of BFS and DFS. |
| **Memory Usage** | Efficient with smaller datasets. | Slightly higher memory usage due to Python's graph representation. |
| **Language Strengths** | Easy visualization of graphs. | Better suited for rapid prototyping and scalability. |

# Comparative & Efficiency Analysis

## Strengths

- MATLAB: Better graph visualization.
- Python: Faster execution for word processing.

## Weaknesses

- MATLAB: Slower for text-heavy tasks.
- Python: Visualization requires external libraries.

## Time & Space Complexity

- Graph Construction: O(V^2).
- BFS for Shortest Path: O(V+E).
- Space Complexity: O(V+E)

## Visualization

- MATLAB excels with built-in graph tools.
- Python relies on libraries like NetworkX.

# Conclusion

## Restate the Objective

This project aimed to efficiently solve the Word Ladder problem using Python and MATLAB, comparing their strengths in computation and visualization.

## Key Findings

- Python is faster for BFS and backtracking.
- MATLAB is more intuitive for graph visualization.

## Comparison

- Python excels in performance-heavy tasks.
- MATLAB is ideal for presentation and exploratory research.

## Additional Insights

- Python handles larger datasets efficiently.
- MATLAB provides seamless integration for mathematical modeling.

# Thank you!