BT3051 - ASSIGNMENT 6

BE23B016

K.VARUNKUMAR

28/10/2024

Contents

1 Problem 1

1.1 Requirements

The problem specifies the implementation of BFS (Breadth-First Search) and DFS (Depth-First Search) algorithms. Below are the requirements : :

- Implement BFS and DFS algorithms for graph traversal.
- Start the traversal from the specified starting node, which is **Germany**.
- Traverse the graph in alphabetical order of the nodes (A-Z).
- Construct the traversal tree for each algorithm:
  - The traversal tree should represent the order in which nodes are visited during the traversal.
  - Both the BFS and DFS traversal trees need to be constructed and returned as objects of the `NetworkX` module.
- Visualize both BFS and DFS traversal trees using appropriate methods from the `NetworkX` module.

1.2 Implementation

- **Queue Class:**
  Defines a simple queue data structure to be used in BFS.
  Used to perform FIFO [First in First Out] extraction of Data.
- **BFS (Breadth-First Search):**
  - Initializes `BFS_TT` as a directed graph to store the BFS traversal tree.
  - Traverses the graph using a queue, marking nodes as visited and recording edges in traversal order.
  - Returns a `parent` dictionary representing the traversal tree structure.
- **DFS (Depth-First Search):**
  - Recursively visits each unvisited neighbor, marking nodes and adding edges to `DFS_TT`.
  - Assigns weights to edges to indicate traversal order..

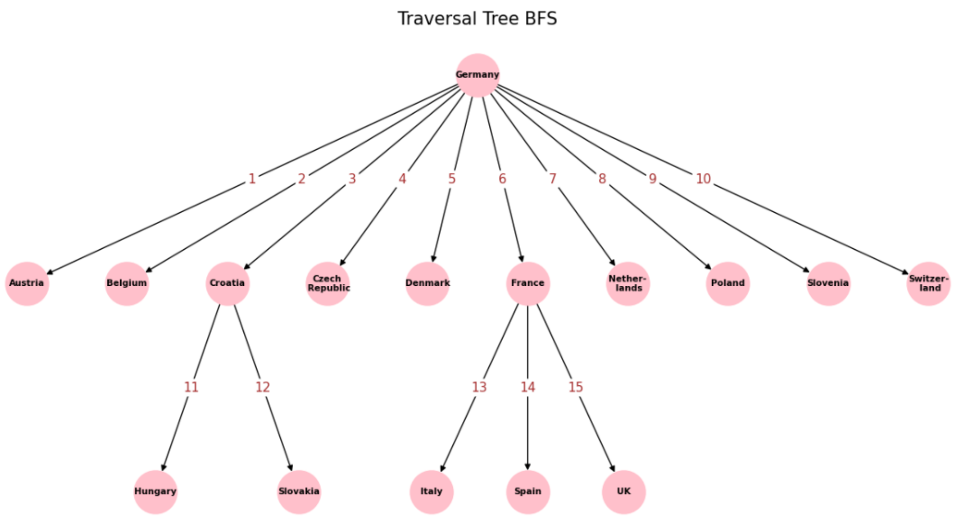
Time Complexity Analysis

- The time complexity of both BFS and DFS is $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph.
- BFS:
  - Each vertex is enqueued and dequeued exactly once, resulting in $O(n)$.
  - Each edge is examined once when checking neighbors, resulting in $O(m)$.

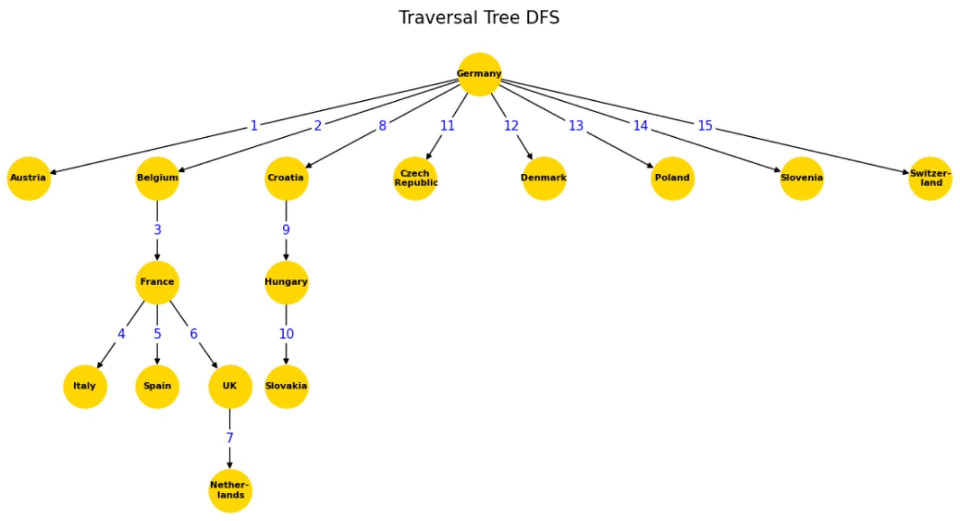
- DFS:
  - Each vertex is visited once, resulting in $O(n)$.
  - Each edge is traversed once during the recursive calls, resulting in $O(m)$.

1.3 Running Test Cases

```
The BFS Graph is as displayed below
```



```
The DFS Graph is as displayed below
```



2 Problem 2

2.1 Requirements

- Implement an algorithm to find the connected components of a graph using a modified form of DFS or BFS. (Here I have choosen to implement BFS).
- For each connected component, ensure that every node is reachable from every other node within the component by following edges.
- The algorithm should determine and print the total number of connected components in the graph.
- Return a dictionary where each component ID is a key, and the corresponding value is a list of nodes in that component.

2.2 Implementation

- **Initialization:**
  - Creates a list `next_component` of all nodes and initializes a dictionary `component` to store each component with a unique ID.
  - Initializes `visited` dictionary to track whether each node has been visited and starts the first component with the initial node.
  - Sets up a queue `q` to manage nodes for traversal.
- **Traversal and Component Detection:**
  - Uses BFS traversal to explore all reachable nodes in the current component.
  - If the queue is empty but unvisited nodes remain, identifies the next unvisited node as the start of a new component.
  - Updates the component counter (`cc`) and continues BFS on each new component.
- **Output:**
  - Prints the number of connected components found.
  - Returns the `component` dictionary, where each key is a component ID, and each value is a list of nodes in that component.

Time Complexity Analysis

- The time complexity of the above solution is similar to that of BFS, that is $O(n + m)$, where n is the number of vertices and m is the number of edges in the graph.
  - Each vertex is visited exactly once, resulting in $O(n)$.
  - Each edge is examined at most once when checking neighbors, resulting in $O(m)$.

2.3 Running Test Cases

```
The Adjacency Matrix is as shown below :
A  :
['B', 'C']
=======================================================
B  :
['A', 'C']
=======================================================
C  :
['B', 'H']
=======================================================
H  :
['C']
=======================================================
D  :
['F', 'G', 'E']
=======================================================
F  :
['E', 'D']
=======================================================
E  :
['F', 'D']
=======================================================
G  :
['D', 'I']
=======================================================
I  :
['G']
=======================================================
K  :
['L']
=======================================================
L  :
['K']
=======================================================
J  :
[]
=======================================================
The number of disjoint sets/components is : 4
=======================================================
{1: ['A', 'B', 'C', 'H'], 2: ['D', 'F', 'G', 'E', 'I'], 3: ['K', 'L'],
4: ['J']}
```

3 Problem 3

3.1 Requirements

- Represent the city's layout as an undirected weighted graph with intersections as nodes and roads as edges.
- Each road (edge) has a travel time (weight) that needs to be minimized during path calculation.
- Find the shortest path from a designated emergency command center, **Node F**, to all other intersections in the city.
- The algorithm should output a nested list in the format `[Source, Target, Duration]` for each shortest path..

3.2 Implementation

- A custom implementation of the `MinHeap` data structure is used to increase the efficiency of **Dijkstra's algorithm**. The min-heap data structure that supports the key operations to:
  - **Extract the minimum element** (root) efficiently, which is the node with the smallest distance.
  - **Insert or update elements** and re-establish the min-heap property using `heapify_up` and `heapify_down`.
  - **Maintain positions** of nodes for constant-time access and updates, enabling faster operations.
  - **Build the heap efficiently** from a list of nodes and distances.
- The algorithm used for finding the shortest path is **Dijkstra's algorithm**, which calculates the shortest paths from a given source node to all other nodes in a weighted graph.
  - Initializes each node with a distance of infinity, except for the source node, which is set to 0.
  - Continuously extracts the node with the minimum distance from the source and updates distances for its unvisited neighbors.
  - Stops when all connected nodes to the source have been processed (nodes disjoint to the source set, are considered to have not been visited).
- **Output format:** The function returns a nested list of shortest paths in the format `[Source, Target, Duration]` for each node.

Time Complexity Analysis

- **Initialization:** Setting up initial distances and building the heap requires $O(n)$, where n is the number of nodes.

- **Heap operations:** In each iteration, the algorithm performs a heap extraction and updates for each node's neighbors.
- **Overall Complexity:** The time complexity is $O((n+m)\log n)$, where n is the number of nodes, and m is the number of edges, due to heap operations for each node and edge.

3.3 Running Test Cases

```
The Adjacency Matrix is as shown below :
A  :
[('B', 4), ('E', 8)]
========================================================
B  :
[('E', 11), ('C', 8), ('A', 4)]
========================================================
C  :
[('F', 2), ('H', 4), ('B', 8), ('D', 7)]
========================================================
D  :
[('C', 7), ('H', 14), ('I', 9)]
========================================================
E  :
[('A', 8), ('B', 11), ('F', 7), ('G', 1)]
========================================================
F  :
[('G', 6), ('E', 7), ('C', 2)]
========================================================
G  :
[('E', 1), ('H', 2), ('F', 6)]
========================================================
H  :
[('G', 2), ('C', 4), ('D', 14), ('I', 10)]
========================================================
I  :
[('H', 10), ('D', 9)]
========================================================
The output is as shown below for the above Adjancency List :
[['F', 'A', 14],
 ['F', 'B', 10],
 ['F', 'C', 2],
 ['F', 'D', 9],
 ['F', 'E', 7],
 ['F', 'F', 0],
 ['F', 'G', 6],
 ['F', 'H', 6],
 ['F', 'I', 16]]
========================================================
```

4 Problem 4

4.1 Requirements

- Extract sentences spoken by Bart from the provided text file (`Q4.txt`) using regular expressions.
- Construct a suffix trie where each word in Bart's sentences represents a node in the trie.
- Implement a function to check if a given sentence was spoken by Bart by searching in the suffix trie.
- Develop a function to autocomplete a partially entered sentence by returning potential completions from Bart's sentences stored in the trie.

NOTE : From the test cases provided, it is understood that punctuation marks at the end of sentences need not be considered while constructing the suffix trie.

4.2 Implementation

- The `SuffixTrie` class is initialized with an empty root dictionary that serves as the base of the trie.
- The `update` method takes a string as input, processes it by marking the start (`*`) and end (`$`) of the sentence, then splits it into words. It constructs branches in the trie by iteratively adding nodes for each suffix in the sentence.
- The `read` function recursively collects all possible paths from a given node, which enables the trie to generate completions for partially entered sentences.

Search Function

- The `search` function takes a sentence as input, formats it by appending start (`*`) and end (`$`) markers, and splits it into words.
- It traverses the trie using each word in the sentence as a key; if any word is missing in the current path, it returns `False`.
- If it successfully traverses all words and reaches a terminal node (`$`), it confirms that the sentence exists and returns `True`.

Autocomplete Function

- The `completion` function takes a partially entered sentence and splits it into words.
- It traverses the trie with the words from the input string to find the last node of the partial sentence. If any word is not found, it terminates with a "No matches" message.
- Upon reaching the end of the input words, it calls the `read` function, which generates all possible completions from the current node.
- The function then concatenates the entered sentence with each completion and prints a list of potential completions.

Time Complexity Analysis

Search Function

- The time complexity of the `search` function is $O(p)$, where p is the number of words in the input sentence.
- This complexity arises because the function traverses the trie once for each word in the sentence.
  If each word is present in the trie, it proceeds to the next word until reaching the terminal node.

Autocomplete Function

- The time complexity of the `completion` function can be broken down into two parts:
  - Let the number of sentences in the input data set be ”n”, and the length of the longest spoken sentence be ”m” words.
  - In the worst case the autocomplete function has to iterate through all the sentences of length m.
  - As such the time complexity of the autocomplete function is $O(mn)$ in the worst case.

However it is to be noted that the time complexity of generating the suffix trie itself is $O(m^2 \cdot n)$

4.3 Running Test Cases

```
Suffix_trie.search("My name is Lance Ambu")
Suffix_trie.search("My name is Bart Simpson")
Suffix_trie.search("My name is Bart")
Suffix_trie.search("I do not want to read Hamlet")
Suffix_trie.search("I do not want to read Shakesphere")
print("="*100)
========================================================================
True
False
True
True
False
========================================================
Suffix_trie.completion("My")
Suffix_trie.completion("I do")
Suffix_trie.completion("name")
========================================================================
['My whole life is a lie',
 'My name is Lance Ambu',
 'My name is Bart',
 'My haircut is awesome']
========================================================
['I do not want to read Hamlet', 'I do not have it']
========================================================
['name is Lance Ambu', 'name is Bart']
========================================================
```

5 Problem 5

5.1 Requirements

Part (a): Tokenizing SMILES Strings

- Develop a single regular expression to parse SMILES (Simplified Molecular Input Line Entry System) strings.
- The SMILES strings should be split into tokens representing atoms, bonds, etc .
- Terms within square brackets, such as complex atom groups or ions, should be treated as a single token without further splitting.
- Expected tokens include individual atoms (e.g., C, N, O), bonds (e.g., =, #, .), brackets for branching, and any content within square brackets (e.g., [Fe], [OH]).
- The regex should capture all tokens in the correct order, reflecting the SMILES syntax.

Part (b): Validating SMILES Strings

- Write a function to check if a given SMILES string is valid according to specified rules.
- Only certain symbols for atoms (C, N, O, F, P, S, Cl, Br, I and lowercase equivalents) are allowed.
- Bonds are represented by specific characters: `=`, `#`, and `.`.
- Branches, represented by parentheses `()` in the SMILES string, should be balanced (each `(` should have a matching `)`).
- Square brackets `[]` are permitted, however upon clarification it was state All atoms, radical signs, bonds and @ symbols are indeed allowed inside [ ] brackets.
- All other character outside the defined ones should be considered invalid.

5.2 Implementation

Tokenize Function:

- The pattern shown below is used to tokenize the input SMILE sequences

```
'Cl|Br|[CNOFPSI]|[=#\.]|[\(\)]|\[[^\[^\]]*\]|\d'
```

- C,N,O.F,P,S,I,Cl,Br and their lower case equivalents are the only allowed atoms (Stated in the rules and thus these are the only atoms considered). This assumption is made to prevent cases such as CN etc from being flagged as 2 tokens in alternative regex methods.
- The second entry is to consider all types of bonds.
- The third entry is to consider "(",")" characters. This is followed by the consideration for complex character in square brackets
- The last entry is to consider every induvidual digit from 0-9 as it's own token.

Validation Function:

- The `search_pattern`. shown below is used to check inside [ ], for validation.

  `'[^A-Z=#\.\(\)\d\+\-@]'`

- Loops over each token:
  - For `(` and `)`, it checks balanced parentheses by using a stack.
  - Rejects any tokens that are digits as per the rules stated for 5)b).
  - Checks square-bracketed tokens for any invalid characters using `search_pattern`.
- Reconstructs the tokens and compares the result to the original SMILES string. If they match, the SMILES string is valid.

5.3 Running Test Cases

```
SMILES : N[C@@H](C)C(=O)O
Tokens: ['N', '[C@@H]', '(', 'C', ')', 'C', '(', '=', 'O', ')', 'O']
Is the sequence valid : True
========================================================
SMILES : N#N
Tokens: ['N', '#', 'N']
Is the sequence valid : True
========================================================
SMILES : [Cu+2].[O-]S(=O)(=O)[O-]
Tokens: ['[Cu+2]', '.', '[O-]', 'S', '(', '=', 'O', ')', '(', '=', 'O',
')', '[O-]']
Is the sequence valid : True
========================================================
SMILES : O1C=C[C@H]
Tokens: ['O', '1', 'C', '=', 'C', '[C@H]']
Is the sequence valid : False
========================================================
SMILES : [O@@H]CO-[PHI]
Tokens: ['[O@@H]', 'C', 'O', '[PHI]']
Is the sequence valid : False
========================================================
SMILES : [COO-@]C[@NH3+]
Tokens: ['[COO-@]', 'C', '[@NH3+]']
Is the sequence valid : True
========================================================
```

---

6 Problem 6

6.1 Requirements

- **Reconstructing a Circular Genome**: Given a set of overlapping short reads (3-mers) generated from sequencing a circular plasmid, the objective is to reconstruct the original plasmid sequence.
- **De Bruijn Graph Construction**: Create a de Bruijn graph from the reads, where nodes represent overlapping (k-1)-mers, and directed edges represent 3-mers (reads) linking these nodes.
- **Graph Visualization**: Use the NetworkX library to visually represent the de Bruijn graph, showing nodes as unique (k-1)-mers and edges as transitions between them.
- **Finding an Eulerian Circuit**: Traverse the de Bruijn graph to find an Eulerian circuit, ensuring that each k-mer is visited once. This circuit represents the reconstructed plasmid sequence.

6.2 Implementation

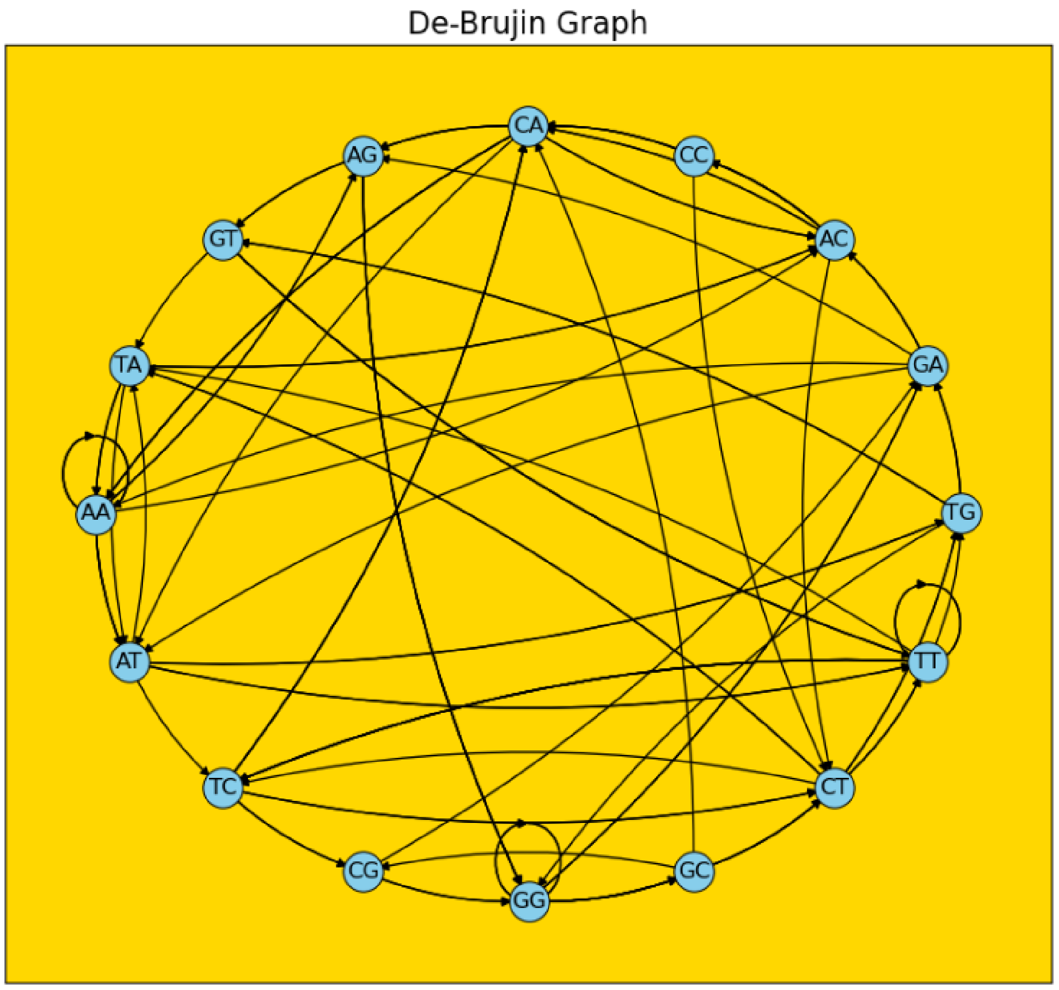
- **Genome_sequence**: This function takes in genomic sequence data and builds a directed multigraph (De Bruijn graph) using NetworkX.
  Each sequence is divided into prefix and suffix nodes, creating directed edges between them, effectively capturing overlaps between k-mers (subsequences).
- **Graph Visualization:** The graph is displayed with Matplotlib in a circular layout. Nodes are rendered in blue, edges with arrows, and the background in gold for clarity. The edge weights are not rendered to ensure readability of output
- **Eulerian Circuit Check:** The prefix and suffix lists constructed at the start are used to determine if a given dataset has eulerian paths, this is done using the fact that the in-degree of each node in the graph must be equal to it's out-degree.
  This fact can be used to claim that the number of times a (k-1)mer appears in the prefix list, it must appear the exact same number of times in the suffix list. Thus the lists are sorted and only when they are equal are Eulerian Circuits possible for the data set
- **Genome Reconstruction:** If an Eulerian circuit exists, it traverses the graph to reconstruct the genome sequence. If no circuit exists, it outputs a message indicating this, and the function returns None, as requested in the question rules.
- **Output Information:** On successful reconstruction, the function prints the genome sequence, node count, and edge count.

The number of nodes in a general De Brujin Graph will be $(4)^{k-1}$. This is because whenever k is a small number (3 here for example) in a database of large quantity of neighbours it is very likely all (k-1)mers will be present (number of kmers >> $(4)^{k-1}$, which is true generally).

The number of edges in the graph denotes the number of input number of kmers as each k-mer will form an edge between it's corresponding prefix (k-1)mer and suffix (k-1)mer, And this is the same thing that we observe in the graph as well.

6.3 Running Test Cases

The graph shown below has edges with weights as the kmer itself, and every edge is directed from the prefix of the kmer to the suffix of the kmer.
The weights are not shown in the graph for enhanced clarity as the text bubbles cover the nodes and some edges, they have been omitted.



```
The Genome Sequence is :
TGTTTTCGGGGCGGCTTAAAAAGGCTAAGGCAGGCTTCAATTCAATTCAACCAGTTCTCTACCAGGAG
TTGGAATACCACACCTGTATCGACTGACATGA
=====================================================================
The number of nodes in the graph : 16
=====================================================================
The total number of edges in the graph : 100
=====================================================================
```