Problem 1: Geography Problem

Description of problems:

You have a set of words. A list of words in legal if the last letter of each word is the same as the first letter of the following word. The problem is to find a legal list of words using all the words exactly once. Assume that initially the first word is there, and you just need to complete it.

The components for solving the problem:

1. **State:**

   In code, the state represents the current word in the word chain. It's a word from the list of words, and the goal is to form a chain of words where each word's last letter matches the first letter of the following word.

2. **Initial State:**

   The initial state is the starting word for list of word chain. It's typically the first word in list of words.

3. **Goal Test:**

   The goal test checks if a valid word chain has been formed, i.e., if the length of the word chain is equal to the total number of words in the list.

4. **Successor Function:**

   The successor function generates the next possible states (words) that can follow the current word in the word chain. In your code, it looks for words that start with the last letter of the current word.

Code

Now, let's discuss the output and the order in which nodes (states) are visited during the search.

This order depends on the specifics of the search algorithm (in this case, depth-first search).

```python
from collections import deque

def find_word_chain(original_list_of_words, search_method):

    def solve_geograpgy_problem_using_bfs(queue, recursion_depth, steps):
        while queue:
            current_word, current_sequence = queue.popleft()
            if len(current_sequence) == len(original_list_of_words):
                return current_sequence

            recursion_depth += 1

            last_letter = current_word[-1]
            match_list = graph_of_first_letter.get(last_letter)

            # If there are no words starting with the last letter, return None
            if match_list is None:
                return None

            for word in match_list:
                steps += 1
                # Print the state (word) as it is visited
                print("Recursion depth: {depth}, steps: {steps} and visiting {word}".f
                    depth=recursion_depth,
                    steps=steps,
                    word=word,
                ))

                if word not in current_sequence:
                    new_sequence = current_sequence + [word]
                    queue.append((word, new_sequence))

        return None  # No valid sequence found

    # Depth-first search function to find a word chain
    def solve_geograpgy_problem_using_dfs(word, stack, recursion_depth, steps):
        if len(stack) == len(original_list_of_words):
            return stack

        last_letter = word[-1]
        match_list = graph_of_first_letter.get(last_letter)
        # If there are no words starting with the last letter, return None
        if match_list is None:
            return None

        recursion_depth += 1

        for next_word in match_list:
            if next_word not in stack:
                stack.append(next_word)

                steps += 1
                # Print the state (word) as it is visited
                print("Recursion depth: {depth}, steps: {steps} and visiting {word}".f
                    depth=recursion_depth,
                    steps=steps,
                    word=next_word,
                ))
```

```python
                # Recursively search for the next word in the chain
                result = solve_geograpgy_problem_using_dfs(next_word, stack, recursion
                if result:
                    return result

                # Backtrack by removing the current word
                stack.pop()

    initial_word = original_list_of_words[0]

    original_list_of_words = set(original_list_of_words)
    if len(original_list_of_words) == 1:
        return initial_word

    # Create a graph_of_first_letter where each word is a node with outgoing edges to
    graph_of_first_letter = {}
    for word in original_list_of_words:
        if word[0] not in graph_of_first_letter:
            graph_of_first_letter[word[0]] = []
        graph_of_first_letter[word[0]].append(word)

    recursion_depth = 0
    steps = 0

    if search_method == 'BFS':
        queue = deque([(initial_word, [initial_word,])])
        return solve_geograpgy_problem_using_bfs(queue, recursion_depth, steps)

    elif search_method == 'DFS':
        stack = [initial_word,]
        return solve_geograpgy_problem_using_dfs(initial_word, stack, recursion_depth,

if __name__ == '__main__':
    # Example list_of_words to form a word chain
    list_of_words = ["ABC", "CDE", "CFG", "EHE", "EIJ", "GHK", "GLC"]
    # list_of_words = ["apple", "lion", "nut", "elephant", "tiger", 'redpoll', 'apple'

    while True:
        search_method = input("Choose BFS or DFS (Enter 'BFS' or 'DFS'): ").strip().up
        if search_method in ['BFS', 'DFS']:
            break

    # Find and print the word sequence
    sequence_of_chain_words = find_word_chain(list_of_words, search_method)

    if sequence_of_chain_words:
        sequence_of_chain_words = ' -> '.join(sequence_of_chain_words)
        print(sequence_of_chain_words)
    else:
        print("No valid word sequence found.")
```

Explanation of code

Here's explaination of the code for finding a word chain from a list of words using depth-first search (DFS):

Explanation: This code aims to find a word chain from a list of words using either Breadth-First Search (BFS) or Depth-First Search (DFS) based on the user's choice. Let's break down the code step by step:

1. The `find_word_chain` function is defined, which takes two arguments: `original_list_of_words` (the list of words to form a word chain from) and `search_method` (the search method to use: 'BFS' or 'DFS').

2. Inside the `find_word_chain` function, there are two sub-functions:

   - `solve_geograpgy_problem_using_bfs`: This function uses BFS to find a word chain. It starts with an initial word, builds a graph of words based on the last letter of each word, and explores the graph using a queue. The goal is to find a sequence of words that can form a valid word chain.

   - `solve_geograpgy_problem_using_dfs`: This function uses DFS to find a word chain. It recursively explores paths, starting from an initial word and backtracking when needed. Like the BFS version, it aims to find a sequence of words that can form a valid word chain.

3. The code prepares the initial word and a graph (`graph_of_first_letter`) that maps the last letter of each word to a list of words starting with that letter.

4. Depending on the `search_method` chosen by the user (BFS or DFS), the code calls the respective search function to find the word chain.

5. The word chain, if found, is returned as a list of words.

6. Finally, the code prompts the user to choose between BFS and DFS and then prints the resulting word chain if one is found.

Note: The example word list `list_of_words` provided in the code is used for testing. You can replace it with your own list of words to find a word chain from.

Output:

Output from the Program:

```
list_of_words = ["apple", "lion", "nut", "elephant", "tiger", 'redpoll']

Choose BFS or DFS (Enter 'BFS' or 'DFS'): bfs
Recursion depth: 1, steps: 1 and visiting elephant
Recursion depth: 2, steps: 2 and visiting tiger
Recursion depth: 3, steps: 3 and visiting redpoll
Recursion depth: 4, steps: 4 and visiting lion
Recursion depth: 5, steps: 5 and visiting nut
apple -> elephant -> tiger -> redpoll -> lion -> nut

Choose BFS or DFS (Enter 'BFS' or 'DFS'): dfs
Recursion depth: 1, steps: 1 and visiting elephant
Recursion depth: 2, steps: 2 and visiting tiger
Recursion depth: 3, steps: 3 and visiting redpoll
Recursion depth: 4, steps: 4 and visiting lion
Recursion depth: 5, steps: 5 and visiting nut
apple -> elephant -> tiger -> redpoll -> lion -> nut
```

```
list_of_words = ["ABC", "CDE", "CFG", "EHE", "EIJ", "GHK", "GLC"]

Choose BFS or DFS (Enter 'BFS' or 'DFS'): bfs
Recursion depth: 1, steps: 1 and visiting CDE
Recursion depth: 1, steps: 2 and visiting CFG
Recursion depth: 2, steps: 3 and visiting EHE
Recursion depth: 2, steps: 4 and visiting EIJ
Recursion depth: 3, steps: 5 and visiting GLC
Recursion depth: 3, steps: 6 and visiting GHK
Recursion depth: 4, steps: 7 and visiting EHE
Recursion depth: 4, steps: 8 and visiting EIJ
No valid word sequence found.

Choose BFS or DFS (Enter 'BFS' or 'DFS'): dfs
Recursion depth: 1, steps: 1 and visiting CDE
Recursion depth: 2, steps: 2 and visiting EIJ
Recursion depth: 2, steps: 3 and visiting EHE
Recursion depth: 3, steps: 4 and visiting EIJ
Recursion depth: 1, steps: 2 and visiting CFG
Recursion depth: 2, steps: 3 and visiting GLC
Recursion depth: 3, steps: 4 and visiting CDE
Recursion depth: 4, steps: 5 and visiting EIJ
Recursion depth: 4, steps: 6 and visiting EHE
Recursion depth: 5, steps: 7 and visiting EIJ
Recursion depth: 2, steps: 4 and visiting GHK
No valid word sequence found.
```

Problem 2: PCP Problem

Description of problems:

You have a set of dominoes. Each one contains a string on the top and a string on the bottom. The problem is to find a sequence of dominoes such that the concatenation of the strings on the top gives the same string as the concatenation of all the strings on the bottom. Dominoes may be used as many times as you want. Assume that the first domino is there.

let's define the components for solving the problem:

1. **State:**

   In this problem, a state represents the current sequence of dominoes that you have placed, which forms a chain. The state changes as you add more dominoes to the sequence.

2. **Initial State:** The initial state represents the starting point of the sequence. It includes the possible matching dominoes.

3. **Goal Test:**

   The goal test checks whether the current sequence of dominoes forms a chain where the concatenation of the top strings matches the concatenation of the bottom strings. If the top and bottom strings of the entire chain are the same, the goal test is satisfied, indicating that you have successfully solved the problem.

4. **Successor Function:** The successor function generates the next possible states by considering which dominoes can be added to the current sequence. For each state, the successor function provides a list of valid next states based on the dominoes that can be added to the chain. These valid next states represent the different possible extensions of the current chain.

In summary, this problem involves finding a sequence of dominoes, starting with the initial state, where the top strings of the dominoes form the same string as the concatenation of all the bottom strings. The successor function helps explore possible sequences, and the goal test checks if the current sequence satisfies the problem's condition.

Code

Now, let's discuss the output and the order in which nodes (states) are visited during the search.

This order depends on the specifics of the search algorithm (in this case, depth-first search).

```python
from collections import deque

def match_top_bottom_domino(dominos, search_method):

    def solve_pcp_problem_using_bfs(queue, recursion_depth, steps):
        # print(queue)
        while queue:
            current_path = queue.popleft()
            recursion_depth += 1

            for path in current_path:
                for domino in dominos:
```

```python
                steps += 1

                print("Recursion depth: {depth}, steps: {steps} and visting: {path
                    depth=recursion_depth,
                    paths=domino,
                    steps=steps
                ))

                new_path = path.copy()
                new_path.append(domino)

                top_domino = "".join([d[0] for d in new_path])
                bottom_domino = "".join([d[1] for d in new_path])

                # Check if top and bottom are equal
                if top_domino == bottom_domino:
                    return [top_domino, bottom_domino]

                # Check for partial match and add to new_possible_paths
                min_len = min(len(top_domino), len(bottom_domino))
                if top_domino[:min_len] == bottom_domino[:min_len]:
                    queue.append([new_path,])

    def solve_pcp_problem_using_dfs(all_possible_paths, recursion_depth, steps):
        # Base case: If there are no more paths to explore, return None
        if not all_possible_paths:
            return None

        new_possible_paths = []
        recursion_depth += 1
        for path in all_possible_paths:
            for domino in dominos:
                new_path = path.copy()
                new_path.append(domino)

                steps += 1

                print("Recursion depth: {depth}, steps: {steps} and visting: {paths}".
                    depth=recursion_depth,
                    paths=domino,
                    steps=steps
                ))

                # Concatenate top and bottom of the dominoes in the path
                top_domino = "".join([d[0] for d in new_path])
                bottom_domino = "".join([d[1] for d in new_path])

                # Check if top and bottom are equal
                if top_domino == bottom_domino:
                    return [top_domino, bottom_domino]

                # Check for partial match and add to new_possible_paths
                min_len = min(len(top_domino), len(bottom_domino))
                if top_domino[:min_len] == bottom_domino[:min_len]:
                    new_possible_paths.append(new_path)

        # Continue the search with new_possible_paths
        return solve_pcp_problem_using_dfs(new_possible_paths, recursion_depth, steps)

    if len(dominos) == 1:
        return None

    recursion_depth = 0
    steps = 0

    # Initialize all_possible_paths with dominoes having an initial partial match
    all_possible_paths = []
    for domino in dominos:
        top_domino, bottom_domino = domino
        min_len = min(len(top_domino), len(bottom_domino))

        if top_domino[:min_len] == bottom_domino[:min_len]:
            all_possible_paths.append([domino,])

    if search_method == 'BFS':
        queue = deque([all_possible_paths, ])
        return solve_pcp_problem_using_bfs(queue, recursion_depth, steps)

    elif search_method == 'DFS':
        return solve_pcp_problem_using_dfs(all_possible_paths, recursion_depth, steps)

if __name__ == '__name__':
    # Example dominoes
    dominoes = [
        ("bba", "b"),
        ("ba", "baa"),
        ("ba", "aba"),
        ("ab", "bba"),
    ]

    # dominoes = [
    #     ("MOM", "M"),
    #     ("O", "OMOMO"),
    # ]

    # dominoes = [
    #     ('AA', 'A')
    # ]

    while True:
        search_method = input("Choose BFS or DFS (Enter 'BFS' or 'DFS'): ").strip().up
        if search_method in ['BFS', 'DFS']:
            break

    # Find a solution for the PCP problem
    solution = match_top_bottom_domino(dominoes, search_method)
    if solution:
        print(f"Match found: {solution[0]} domino top is equal to {solution[1]} domino
    else:
        print("No match found.")
```

Explanation of code

Here's an explanation of the provided Python code for solving the Post Correspondence Problem (PCP):

1. The `match_top_bottom_domino` function takes two arguments: `dominos` (a list of dominos represented as pairs of strings) and `search_method` (the search method to use, either 'BFS' or 'DFS').
2. There are two sub-functions for solving the PCP problem:
   - `solve_pcp_problem_using_bfs`: This function uses BFS to find a solution. It starts with an empty queue containing an initial path and explores all possible paths, checking for matches at each step.
   - `solve_pcp_problem_using_dfs`: This function uses DFS to find a solution. It recursively explores paths, adding dominos to the path and backtracking when necessary.
3. If the list of dominos contains only one domino, it's impossible to form a sequence that satisfies the PCP, so the function returns `None` in that case.
4. The code initializes `recursion_depth` and `steps` counters to keep track of the depth of recursion and the number of steps taken in the search process.
5. The code then initializes `all_possible_paths` with dominos that have an initial partial match (i.e., the top and bottom have a common prefix). These paths serve as starting points for both BFS and DFS.
6. Depending on the `search_method` chosen by the user, the code either calls the BFS or DFS function to find a solution.
7. After finding a solution, if one exists, it prints the matching top and bottom dominos along with their sizes.
8. The code prompts the user to choose between BFS and DFS to determine the search method.
9. The example dominos are provided in the code for testing. You can modify the `dominos` list with your own set of dominos to find a solution for a specific PCP instance.
10. The code uses the `if __name__ == '__main__':` block to ensure that it only executes the PCP solving logic when the script is run directly and not when it's imported as a module.

Remember to replace the example dominos with your specific PCP instance to find a solution for your problem.

Output of code:

Output from the Program:

```
dominoes = [
    ("bba", "b"),
    ("ba", "baa"),
    ("ba", "aba"),
    ("ab", "bba"),
]

Choose BFS or DFS (Enter 'BFS' or 'DFS'): bfs
Recursion depth: 69355, steps: 277421 and visting: ('bba', 'b')
Recursion depth: 69355, steps: 277422 and visting: ('ba', 'baa')
Recursion depth: 69355, steps: 277423 and visting: ('ba', 'aba')
Recursion depth: 69355, steps: 277424 and visting: ('ab', 'bba')
Recursion depth: 69356, steps: 277425 and visting: ('bba', 'b')
Recursion depth: 69356, steps: 277426 and visting: ('ba', 'baa')
Recursion depth: 69356, steps: 277427 and visting: ('ba', 'aba')
Recursion depth: 69356, steps: 277428 and visting: ('ab', 'bba')
Recursion depth: 69357, steps: 277429 and visting: ('bba', 'b')
Recursion depth: 69357, steps: 277430 and visting: ('ba', 'baa')
Recursion depth: 69357, steps: 277431 and visting: ('ba', 'aba')
Recursion depth: 69357, steps: 277432 and visting: ('ab', 'bba')
Recursion depth: 69358, steps: 277433 and visting: ('bba', 'b')
Recursion depth: 69358, steps: 277434 and visting: ('ba', 'baa')
Recursion depth: 69358, steps: 277435 and visting: ('ba', 'aba')
Recursion depth: 69358, steps: 277436 and visting: ('ab', 'bba')
Recursion depth: 69359, steps: 277437 and visting: ('bba', 'b')
Recursion depth: 69359, steps: 277438 and visting: ('ba', 'baa')
Recursion depth: 69359, steps: 277439 and visting: ('ba', 'aba')
Match found: baabbaababbabbabaabbaabbaababbaababbabbaababbabbabaabbbabbabaabababbabbab

Choose BFS or DFS (Enter 'BFS' or 'DFS'): dfs
Recursion depth: 65, steps: 277420 and visting: ('ab', 'bba')
Recursion depth: 65, steps: 277421 and visting: ('bba', 'b')
Recursion depth: 65, steps: 277422 and visting: ('ba', 'baa')
Recursion depth: 65, steps: 277423 and visting: ('ba', 'aba')
Recursion depth: 65, steps: 277424 and visting: ('ab', 'bba')
Recursion depth: 65, steps: 277425 and visting: ('bba', 'b')
Recursion depth: 65, steps: 277426 and visting: ('ba', 'baa')
Recursion depth: 65, steps: 277427 and visting: ('ba', 'aba')
Recursion depth: 65, steps: 277428 and visting: ('ab', 'bba')
Recursion depth: 65, steps: 277429 and visting: ('bba', 'b')
Recursion depth: 65, steps: 277430 and visting: ('ba', 'baa')
Recursion depth: 65, steps: 277431 and visting: ('ba', 'aba')
Recursion depth: 65, steps: 277432 and visting: ('ab', 'bba')
Recursion depth: 65, steps: 277433 and visting: ('bba', 'b')
Recursion depth: 65, steps: 277434 and visting: ('ba', 'baa')
Recursion depth: 65, steps: 277435 and visting: ('ba', 'aba')
Recursion depth: 65, steps: 277436 and visting: ('ab', 'bba')
Recursion depth: 65, steps: 277437 and visting: ('bba', 'b')
Recursion depth: 65, steps: 277438 and visting: ('ba', 'baa')
Recursion depth: 65, steps: 277439 and visting: ('ba', 'aba')
Match found: baabbaababbabbabaabbaabbaababbaababbabbaababbabbabaabbbabbabaababababbabbab
```

```
dominoes = [
    ('AA', 'A')
]

Choose BFS or DFS (Enter 'BFS' or 'DFS'): dfs
No match found.
```

```
dominoes = [
    ("MOM", "M"),
    ("O", "OMOMO"),
]

Choose BFS or DFS (Enter 'BFS' or 'DFS'): bfs
Recursion depth: 1, steps: 1 and visting: ('MOM', 'M')
Recursion depth: 1, steps: 2 and visting: ('O', 'OMOMO')
Recursion depth: 1, steps: 3 and visting: ('MOM', 'M')
Recursion depth: 1, steps: 4 and visting: ('O', 'OMOMO')
Recursion depth: 2, steps: 5 and visting: ('MOM', 'M')
Match found: MOMOMOM domino top is equal to MOMOMOM domino bottom with a size of 7 eac

Choose BFS or DFS (Enter 'BFS' or 'DFS'): dfs
Recursion depth: 1, steps: 1 and visting: ('MOM', 'M')
Recursion depth: 1, steps: 2 and visting: ('O', 'OMOMO')
Recursion depth: 1, steps: 3 and visting: ('MOM', 'M')
Recursion depth: 1, steps: 4 and visting: ('O', 'OMOMO')
Recursion depth: 2, steps: 5 and visting: ('MOM', 'M')
Match found: MOMOMOM domino top is equal to MOMOMOM domino bottom with a size of 7 eac
```