Random Graph Generation

The following part of code generates random graphs, which serve as test cases:

```
def random_graph(num_vertices, edge_probability=0.5):
    """
    Creates a random graph using NetworkX and converts it to an adjacency
matrix.
    """
    G = nx.gnp_random_graph(num_vertices, edge_probability)
    adj_matrix = nx.to_numpy_array(G, dtype=int)
    return adj_matrix
```

The function random_graph(num_vertices, edge_probability=0.5) generates random graphs of different sizes. This is essentially creating test cases with varying numbers of vertices to check the Hamiltonian cycle detection.

Hamiltonian Cycle Search

This part of the code is responsible for checking if there is a Hamiltonian cycle in the generated random graphs:

```
def find_hamiltonian_cycle(graph):
    """
    Tries to find a Hamiltonian cycle by checking all permutations.
    Returns the cycle if found, otherwise None.
    """
    n = len(graph)
    for perm in itertools.permutations(range(n)):
        if is_hamiltonian_cycle(graph, perm):
            return perm  # Found a Hamiltonian cycle
    return None
```

The function find_hamiltonian_cycle(graph) is called within the main() function to check if a Hamiltonian cycle exists in each randomly generated graph. It uses brute force by checking all permutations of vertices to see if any form a Hamiltonian cycle.

Multiple Trials per Graph Size

The code tests 50 random graphs for each graph size to gather more data and ensure consistency:

```
    sizes = range(4, 11)  # Vertex sizes from 4 to 10
    trials_per_size = 50  # 50 trials per vertex size
```

The code performs 50 trials per graph size to gather sufficient data. This serves as a repeated test for each graph size, ensuring that the results are statistically significant rather than an outcome from a single graph instance.

Execution Time Measurement

The execution time for finding a Hamiltonian cycle is measured for each trial using the following code:

```
start_time = time.perf_counter()  # Use high-resolution timer
            cycle = find_hamiltonian_cycle(graph)
            exec_time = time.perf_counter() - start_time  # Measure
execution time

            execution_times.append(exec_time)
```
This measures the performance of the cycle detection algorithm, effectively testing its efficiency with different graph sizes.

Cycle Detection Output

The detected Hamiltonian cycles (if any) are stored and printed, allowing you to check whether the algorithm works correctly:

```
if cycle:
       cycles_found[size].append(cycle)

       # Output average execution time for the current size
       print(f"Size {size} Execution time: {avg_time:.4f} seconds avg.")
```
The Hamiltonian cycles found (if any) are stored and printed, which allows you to verify whether the algorithm correctly identifies valid cycles in the random graphs.