Overview

This project provides a solution for finding **Hamiltonian Cycles** in a graph using a combination of **backtracking** and **parallelism**. The implementation demonstrates graph representation, path exploration, and parallelized recursive search with performance measurements.

Algorithms

Graph Representation

The graph is represented using an **adjacency list**:

- Nodes are stored in a list (List<Integer>).
- Edges are stored as a list of lists (List<List<Integer>>), where each inner list contains the nodes that a given node is connected to.

Hamiltonian Cycle Search

The algorithm combines **recursive backtracking** with **parallelism** to search for Hamiltonian Cycles. The steps are as follows:

1. **Base Case**:
    - If all nodes have been visited and the current node is connected to the starting node (0), a Hamiltonian Cycle is found.
    - The solution is printed.
2. **Recursive Exploration**:
    - For the current node, explore all its neighbors.
    - If a neighbor has not been visited, add it to the path, remove the edge temporarily to avoid revisiting, and recursively search from that neighbor.
    - After the recursive call, restore the edge and remove the node from the path (backtracking).
3. **Parallelized Search**:
    - Each recursive search for a neighbor runs in its own thread using a **thread pool** (ThreadPoolExecutor).
    - Threads execute the recursive search function concurrently for faster exploration.
4. **Backtracking**:
    - If a path fails to meet the criteria for a Hamiltonian Cycle, the algorithm backtracks by:
        - Removing the last added node from the path.
        - Restoring the removed edge for future paths.

Synchronization in Parallelized Variants

Parallelism is achieved by exploring possible paths concurrently using Java's ThreadPoolExecutor. The following synchronization mechanisms are used:

1. **Thread Pool Management**:
   - A fixed thread pool (Executors.newFixedThreadPool) is created with up to 4 threads.
   - Each thread executes the recursive search function for a different path.
2. **Edge Restoration**:
   - To avoid conflicts in shared graph data (edges), edges are removed and restored **synchronously**:
     - Before a recursive call, the edge is removed from the adjacency list.
     - After the recursive call completes, the edge is restored.
   - This ensures no two threads interfere with each other's traversal.
3. **Thread Termination**:
   - After submitting tasks to the thread pool, the shutdown() method is called.
   - The main thread waits for threads to complete using awaitTermination().
4. **Potential Issues**:
   - **Thread Contention**: Multiple threads modifying shared data (like the edge list) could lead to race conditions without proper synchronization.
   - **Overhead**: Creating and shutting down thread pools for every recursive call introduces significant overhead.

Performance Measurements

Environment:

- **Processor:** Modern multi-core processor (e.g., Intel Core i5-8300H).
- **Input:** Polynomials of degree 10,000.
- **Number of Threads:** 2 threads for classic parallel and dynamic threads for Karatsuba parallel.

Results (Example):

- **Random graph(10 node)**

Graph{nodes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], edges=[[0, 6, 3, 7, 8, 1], [6, 5, 2, 7, 3, 0], [5, 2, 8], [8, 7, 2, 5], [4, 9, 3, 1], [7, 4, 5, 6, 8, 9], [7, 4, 6, 3], [3, 9, 7, 6, 8], [5, 8, 6], [4, 1, 2, 7, 8]]}

Solution found:[0, 3, 2, 5, 7, 8, 6, 4, 9, 1]

Solution found:[0, 3, 2, 5, 7, 9, 8, 6, 4, 1]

Solution found:[0, 3, 2, 5, 8, 6, 7, 9, 4, 1]

Solution found:[0, 3, 2, 5, 9, 7, 8, 6, 4, 1]

Solution found:[0, 3, 2, 8, 5, 6, 7, 9, 4, 1]

Solution found:[0, 3, 2, 8, 5, 7, 6, 4, 9, 1]

Solution found:[0, 3, 2, 8, 5, 9, 7, 6, 4, 1]

```
Solution found:[0, 3, 5, 7, 9, 2, 8, 6, 4, 1]

Solution found:[0, 3, 7, 9, 2, 5, 8, 6, 4, 1]

Solution found:[0, 3, 7, 9, 2, 8, 5, 6, 4, 1]

Solution found:[0, 3, 8, 6, 7, 9, 2, 5, 4, 1]

Solution found:[0, 6, 3, 2, 8, 5, 7, 9, 4, 1]

Solution found:[0, 6, 3, 7, 9, 2, 8, 5, 4, 1]

Solution found:[0, 6, 4, 3, 2, 8, 5, 7, 9, 1]

Solution found:[0, 6, 7, 3, 2, 8, 5, 4, 9, 1]

Solution found:[0, 6, 7, 3, 2, 8, 5, 9, 4, 1]

Solution found:[0, 7, 3, 2, 5, 8, 6, 4, 9, 1]

Solution found:[0, 7, 3, 2, 5, 9, 8, 6, 4, 1]

Solution found:[0, 7, 3, 2, 8, 5, 6, 4, 9, 1]

Solution found:[0, 7, 3, 5, 9, 2, 8, 6, 4, 1]

Solution found:[0, 7, 6, 3, 2, 8, 5, 4, 9, 1]

Solution found:[0, 7, 6, 3, 2, 8, 5, 9, 4, 1]

Solution found:[0, 7, 6, 4, 3, 2, 8, 5, 9, 1]

Solution found:[0, 7, 8, 6, 3, 2, 5, 4, 9, 1]

Solution found:[0, 7, 8, 6, 3, 2, 5, 9, 4, 1]

Solution found:[0, 7, 8, 6, 4, 3, 2, 5, 9, 1]

Solution found:[0, 7, 9, 2, 8, 6, 3, 5, 4, 1]

Solution found:[0, 7, 9, 8, 6, 3, 2, 5, 4, 1]

Solution found:[0, 8, 6, 3, 2, 5, 7, 9, 4, 1]

Solution found:[0, 8, 6, 3, 7, 9, 2, 5, 4, 1]

Solution found:[0, 8, 6, 4, 3, 2, 5, 7, 9, 1]

Solution found:[0, 8, 6, 7, 3, 2, 5, 4, 9, 1]

Solution found:[0, 8, 6, 7, 3, 2, 5, 9, 4, 1]

Elapsed running time: 0.9182614s
```

- **Random graph(10 nodes) with no solutions**

```
Graph{nodes=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9], edges=[[6, 4, 8, 1], [8, 1, 5], [5, 3, 4, 8], [5, 3, 8], [8, 9], [6, 0, 3, 1],
[1, 0, 6, 5], [0, 8], [6, 2, 5, 3], [0, 4, 6, 1, 5]]}

Elapsed running time: 0.0696705s
```