

Project Title

Hamiltonian Cycle based Traveling Salesman Problem Using C++ , Python, & OpenMP

Group Members

- 22k-4173
- 22k-4471
- 22k-4625

Problem Statement

This project addresses the Hamiltonian Cycle problem, a fundamental concept in Graph Theory. The problem involves determining a cycle in a graph that visits every vertex exactly once before returning to the starting vertex. This is a challenging NP-complete problem, particularly for large, complex graphs.

The nearest neighbor heuristic offers a greedy, non-optimal solution to this problem. Starting from a randomly chosen vertex, the algorithm repeatedly selects the nearest unvisited vertex until all vertices are visited. While this approach does not guarantee an optimal Hamiltonian cycle, it provides an efficient approximation suitable for comparing algorithmic implementations across different programming languages.

GitHub Repository

https://github.com/ibrahim-012/Project_PDC_Hamiltonian_Cycles

Algorithm Explanation

[https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_College_Mathematics_for_Everyday_Life_\(Inigo_et_al\)/06%3A_Graph_Theory/6.04%3A_Hamiltonian_Circuits](https://math.libretexts.org/Bookshelves/Applied_Mathematics/Book%3A_College_Mathematics_for_Everyday_Life_(Inigo_et_al)/06%3A_Graph_Theory/6.04%3A_Hamiltonian_Circuits)

Project Implementation

Graphical User Interface (GUI)

The project includes a user-friendly GUI that enables users to interact with the algorithm seamlessly. Through this interface, users can:

- Select the programming language (Serial C++ for efficiency, Python for simplicity, or OpenMP for parallel processing).
- Choose a dataset to execute the Hamiltonian Cycle algorithm.
- Visualize the dataset, which is represented as a complete graph with clearly marked edge weights.

Visualization Features

The GUI also provides graphical representations to enhance user understanding:

- **Complete Graph Visualization:** The given dataset is displayed as a complete graph where vertices and weighted edges are prominently shown.
- **Hamiltonian Cycle Visualization:** After selecting a starting vertex, the GUI highlights the Hamiltonian cycle, showcasing the path and corresponding edge weights.

Performance Metrics Logging

The time taken to compute Hamiltonian cycles for all vertices in the selected dataset is logged automatically. These runtime metrics are stored in a file for further analysis. Each log entry includes:

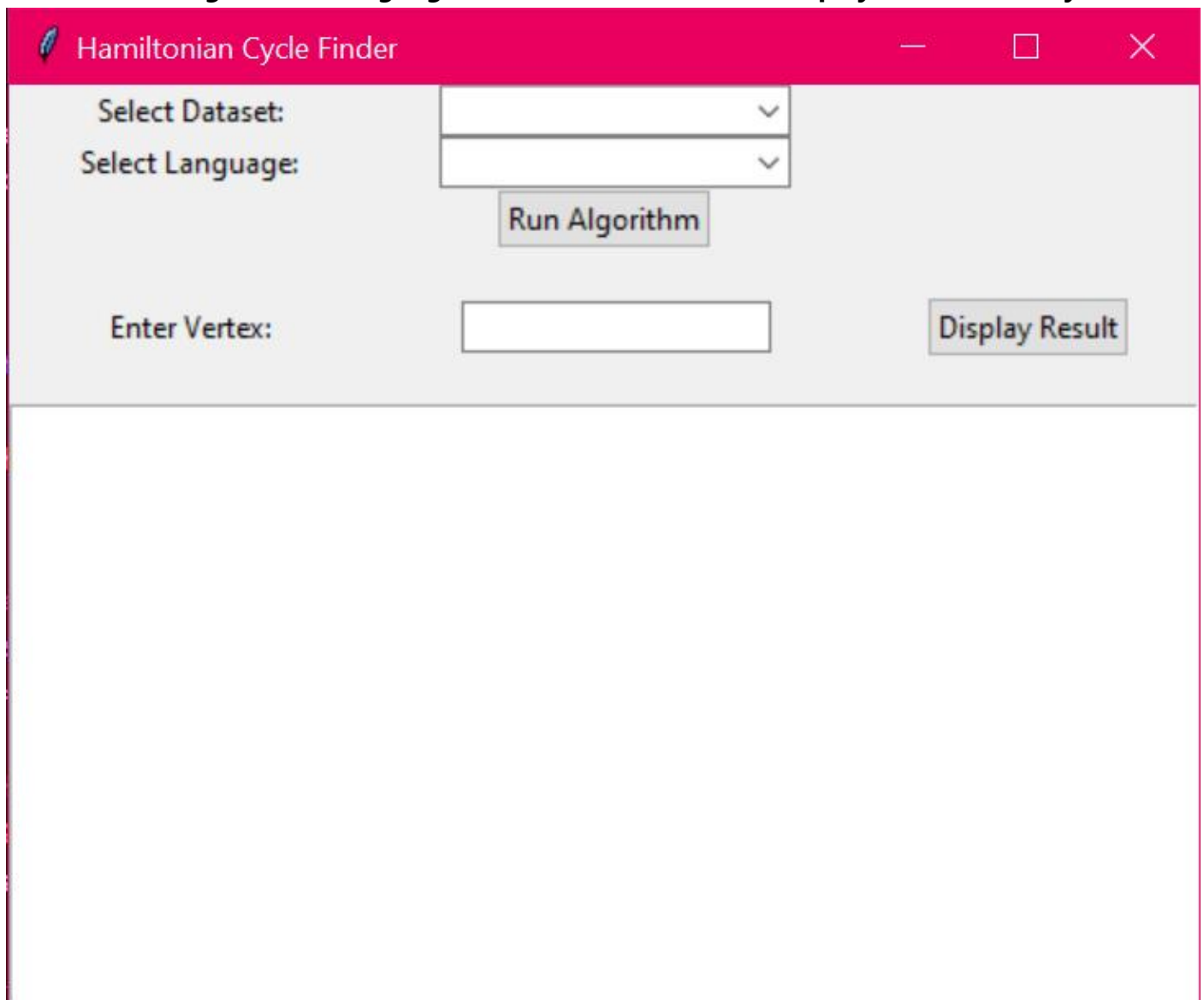
- The number of vertices in the dataset.
- The runtime (in milliseconds).
- The programming language used.

Performance Comparison

The collected performance metrics are utilized to create comparative graphs:

- These graphs illustrate runtime differences across programming languages for the same dataset.
- Users can analyze how each language performs relative to the others, offering insights into computational efficiency and implementation overhead.

GUI for selecting dataset, language, and vertex for which to display Hamiltonian Cycle



The image shows a graphical user interface (GUI) titled "Hamiltonian Cycle Finder". The window has a red title bar with standard minimize, maximize, and close buttons. The main area is light gray and contains several input fields and buttons. At the top left, there are two labels: "Select Dataset:" and "Select Language:", each followed by a white dropdown menu with a small downward arrow. Below these is a button labeled "Run Algorithm". Further down, there is a label "Enter Vertex:" followed by a white text input field. To the right of this input field is a button labeled "Display Result". The bottom half of the window is a large, empty white rectangular area, likely intended for displaying the results of the algorithm.

Hamiltonian Cycle Finder

Select Dataset:

Select Language:

Run Algorithm

Enter Vertex:

Display Result

Sample Run on Complete Graph on 5 vertices

Hamiltonian Cycle Finder

Select Dataset:

Select Language:

Run Algorithm

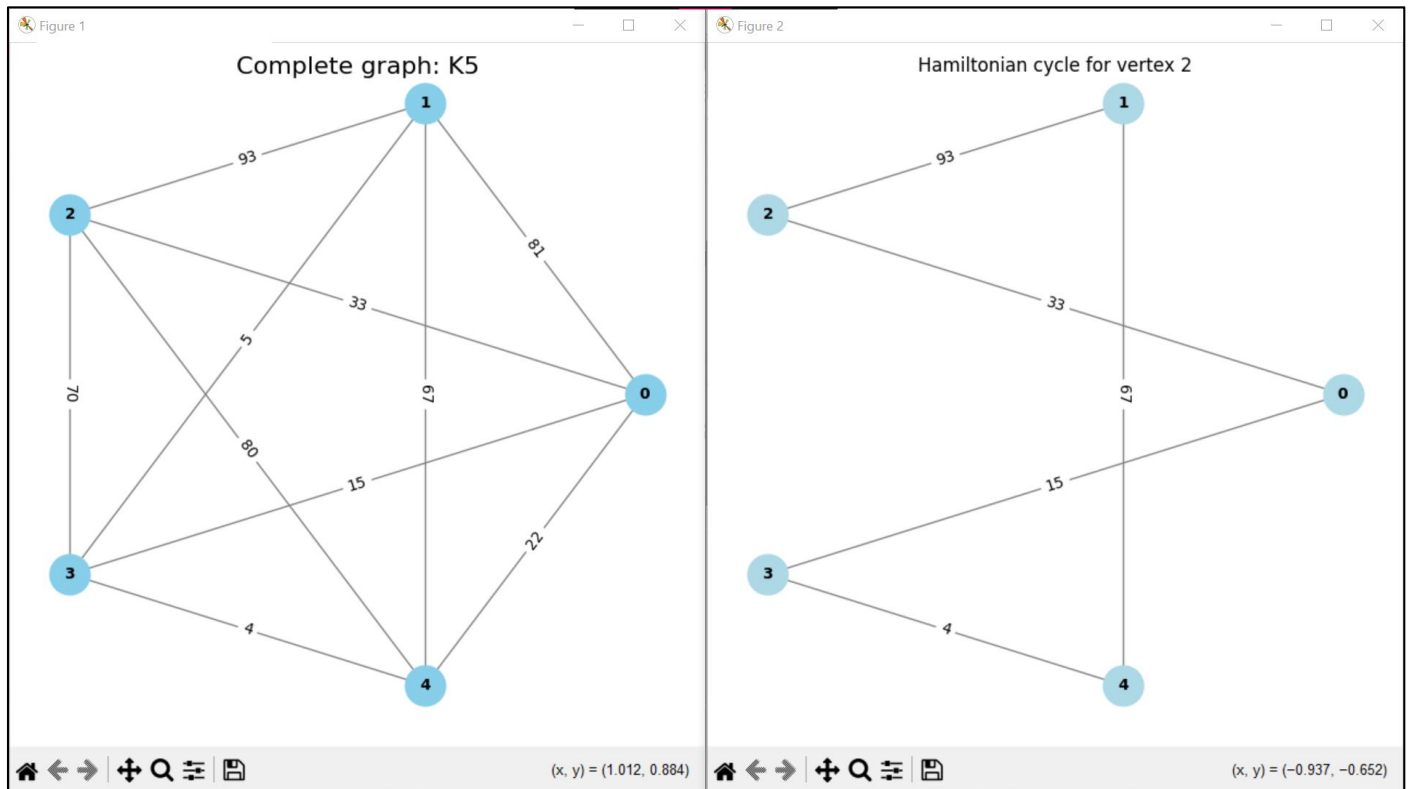
Program executed in Python.
BFS Time: 0.0000000000 seconds.
View results for a specific vertex.

Enter Vertex:

Display Result

Path and Cost for Vertex 2:
Path: 2->0->3->4->1->2
Cost: 212

0	81	33	15	22
81	0	93	5	67
33	93	0	70	80
15	5	70	0	4
22	67	80	4	0



Sample Run on Complete Graph on 10 vertices

Hamiltonian Cycle Finder

Select Dataset:

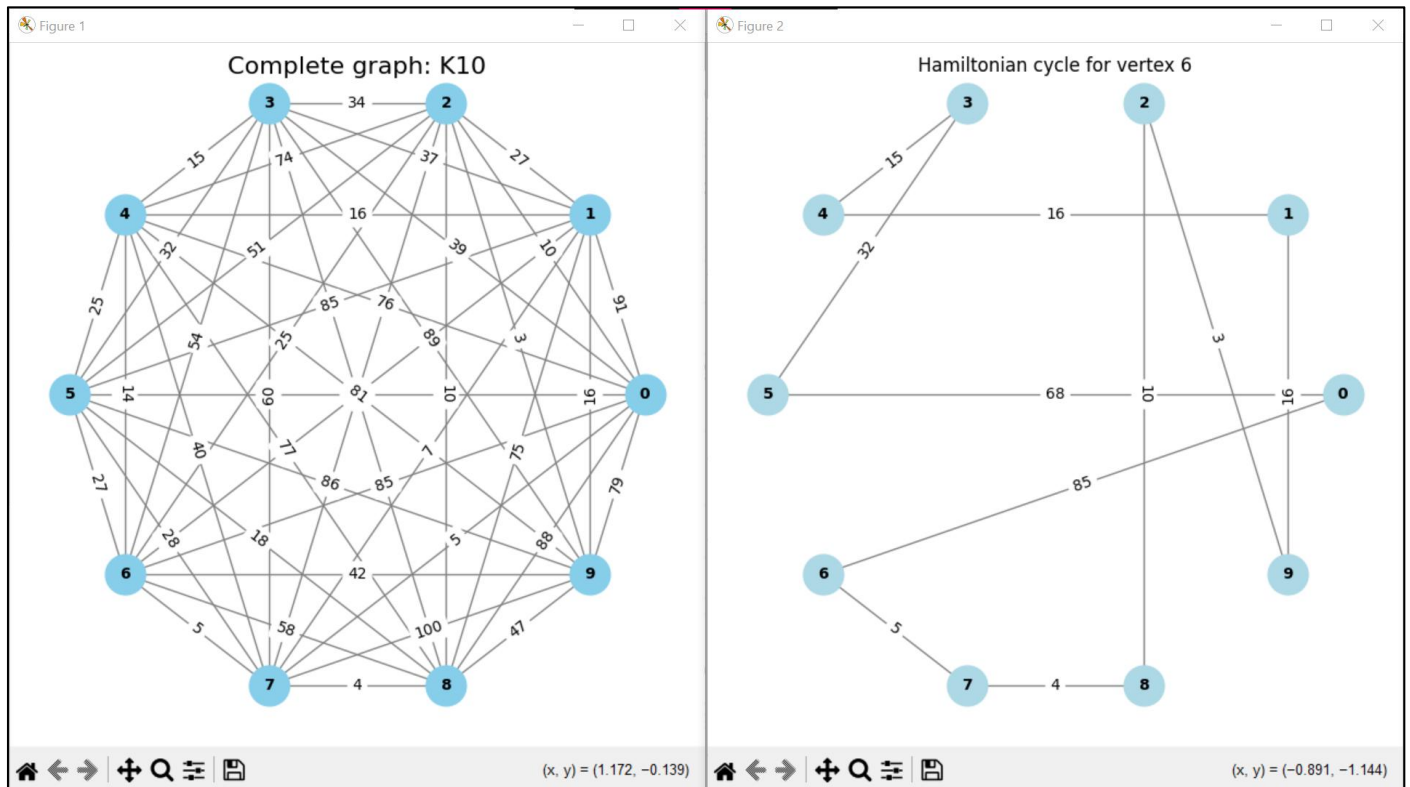
Select Language:

Program executed in Python.
BFS Time: 0.0000000000 seconds.
View results for a specific vertex.

Enter Vertex:

Path and Cost for Vertex 6:
Path: 6->7->8->2->9->1->4->3->5->0->6
Cost: 254

0	91	10	39	76	68	85	5
88	79						
91	0	27	37	16	85	67	7
75	16						
10	27	0	34	74	51	25	65
10	3						
39	37	34	0	15	32	54	60
50	89						
76	16	74	15	0	25	14	40
77	81						
68	85	51	32	25	0	27	28
18	86						
85	67	25	54	14	27	0	5
58	42						
5	7	65	60	40	28	5	0



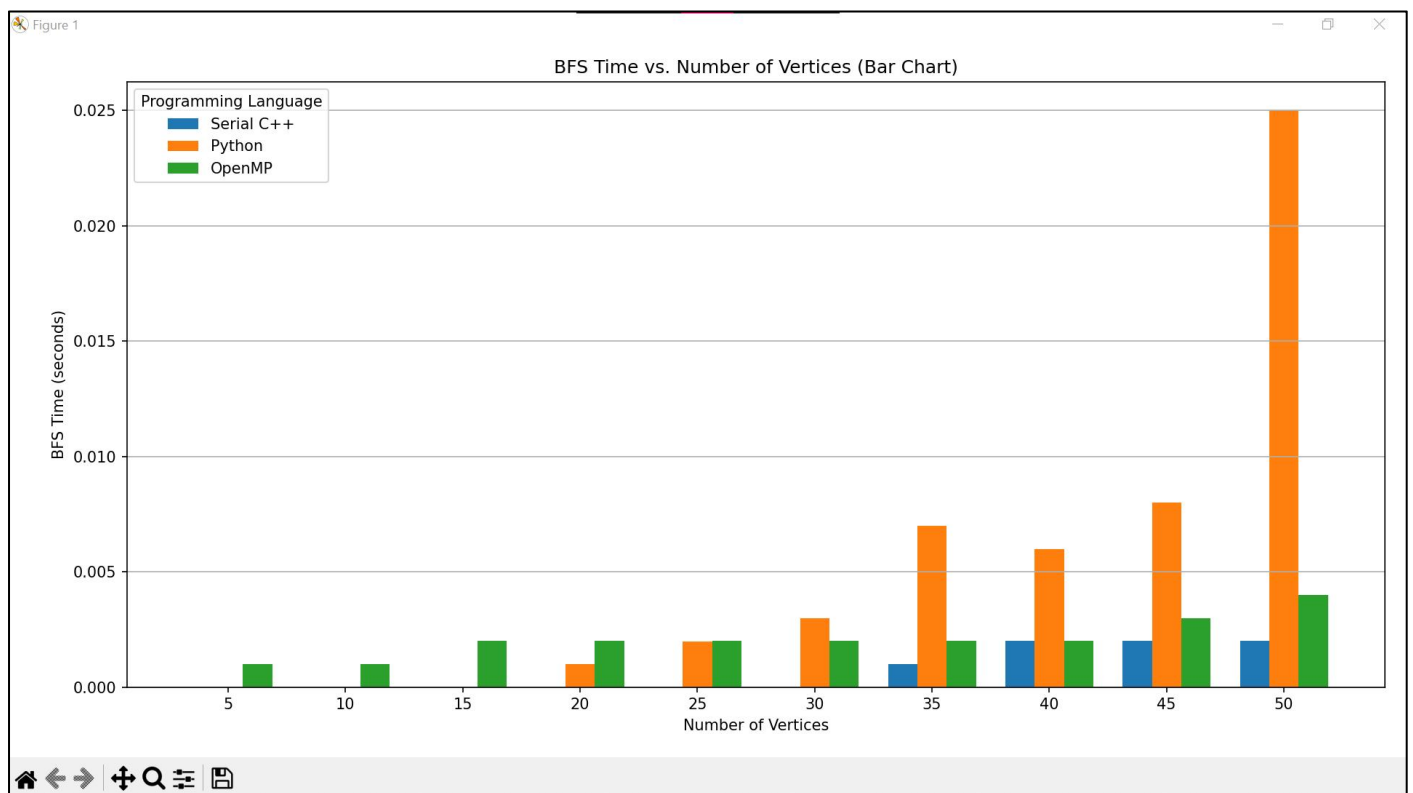
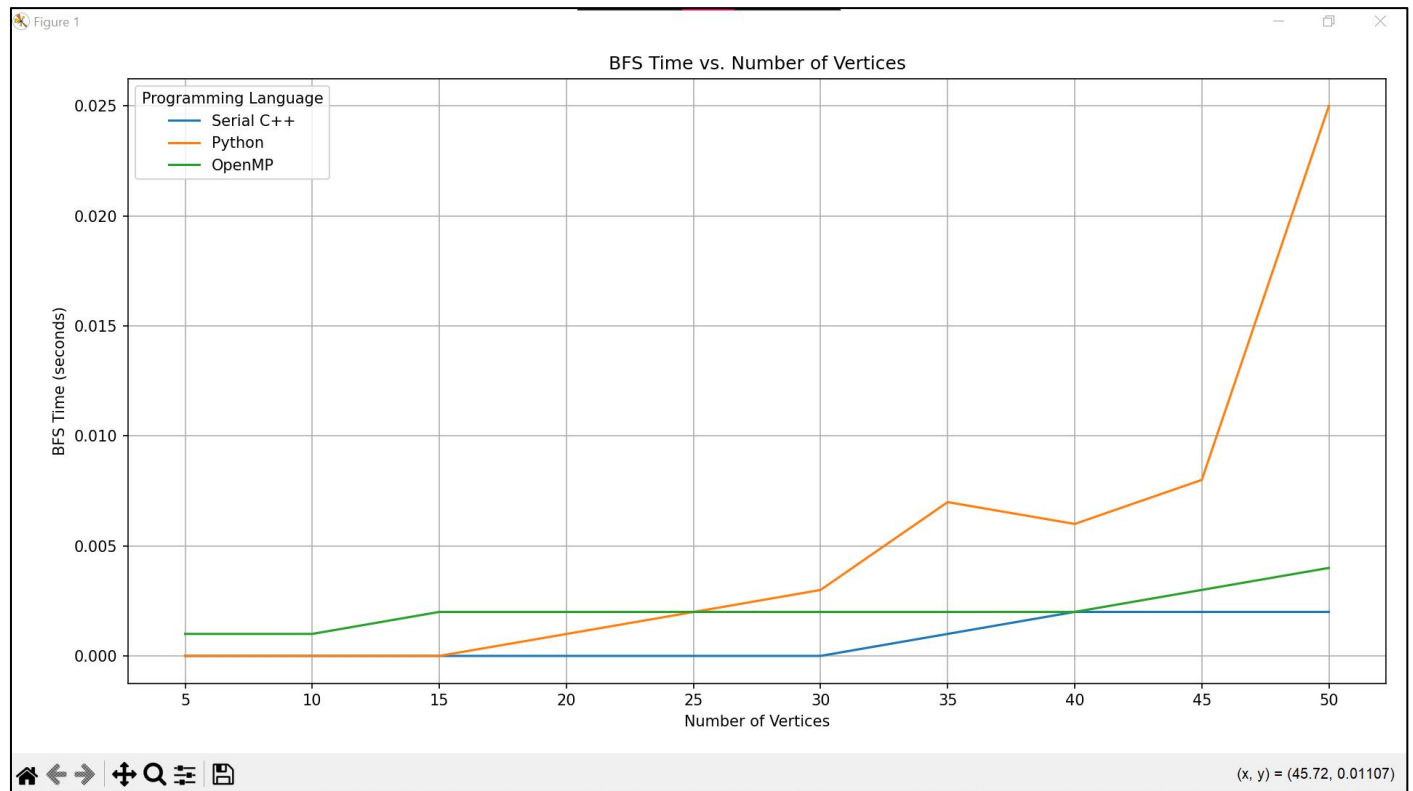
Runtime Comparison for Sample Datasets

Runtime Metrics

	Runtime (in milliseconds)		
Input Size (vertices)	C++ Serial	Python	OpenMP
5	0.0005	0.0007	0.0009
10	0.0006	0.0009	0.0009
15	0.0007	0.0012	0.0020
20	0.0008	0.0009	0.0022
25	0.0010	0.0019	0.0024
30	0.0015	0.0029	0.0026
35	0.0017	0.0069	0.0028
40	0.0020	0.0059	0.0029
45	0.0023	0.0080	0.0030
50	0.0025	0.0250	0.0039

Runtime Comparison for Sample Datasets

Runtime Graphs



Runtime Analysis of Metrics

General Findings

C++ Serial:

- Excels for small datasets due to its compiled nature, which ensures fast execution and minimal runtime overhead.
- Becomes less effective for larger datasets as it lacks the ability to leverage multiple cores, leading to slower scaling.

Python:

- Performs relatively poorly due to its interpreted nature, higher abstraction, and reliance on dynamic typing.
- Runtime increases significantly as datasets grow, making it unsuitable for large-scale problems.

OpenMP (Parallel C++):

- Provides consistent performance across all dataset sizes, leveraging multi-threading to process tasks efficiently.
- The advantage of parallelism becomes more evident with larger datasets, making it the best choice for large-scale Hamiltonian Cycle computations.

Conclusion

C++ Serial provides the fastest results due to its optimized compilation process for small datasets,

OpenMP emerges as the clear winner, thanks to its parallelized nature as dataset size increases.

Python, while versatile and easy to use, is unsuitable for computationally intensive tasks involving large datasets.