**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_GT_Nearest_Neighbour_Algorithm.git>**

**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>**

**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:

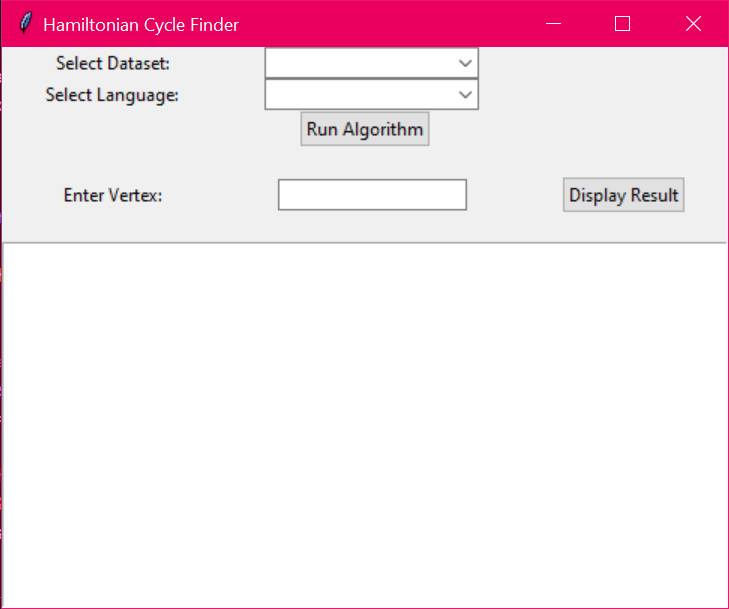
* The number of vertices in the dataset.
* The runtime (in milliseconds).
* The programming language used.

**Performance Comparison**

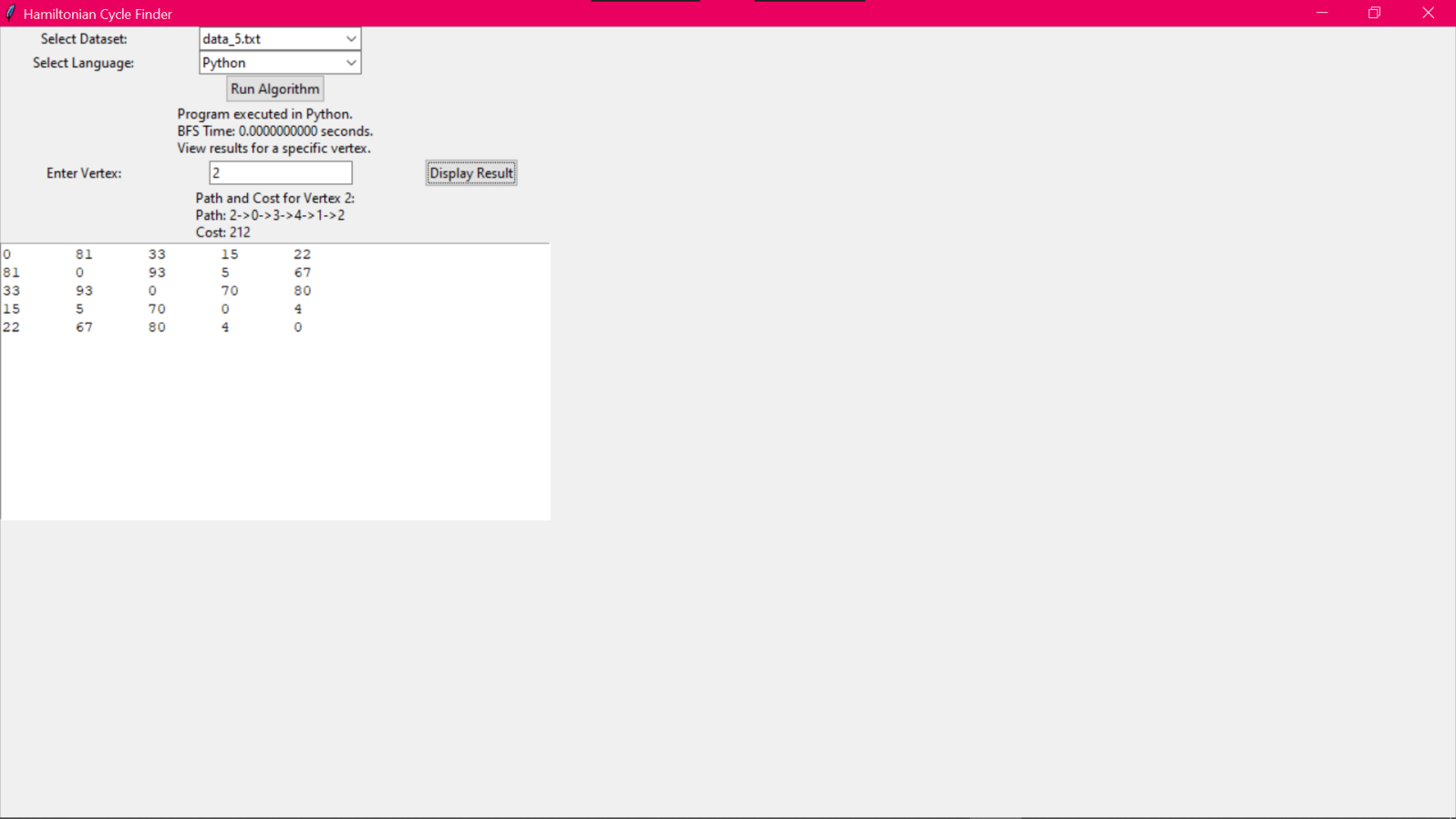
The collected performance metrics are utilized to create comparative graphs:

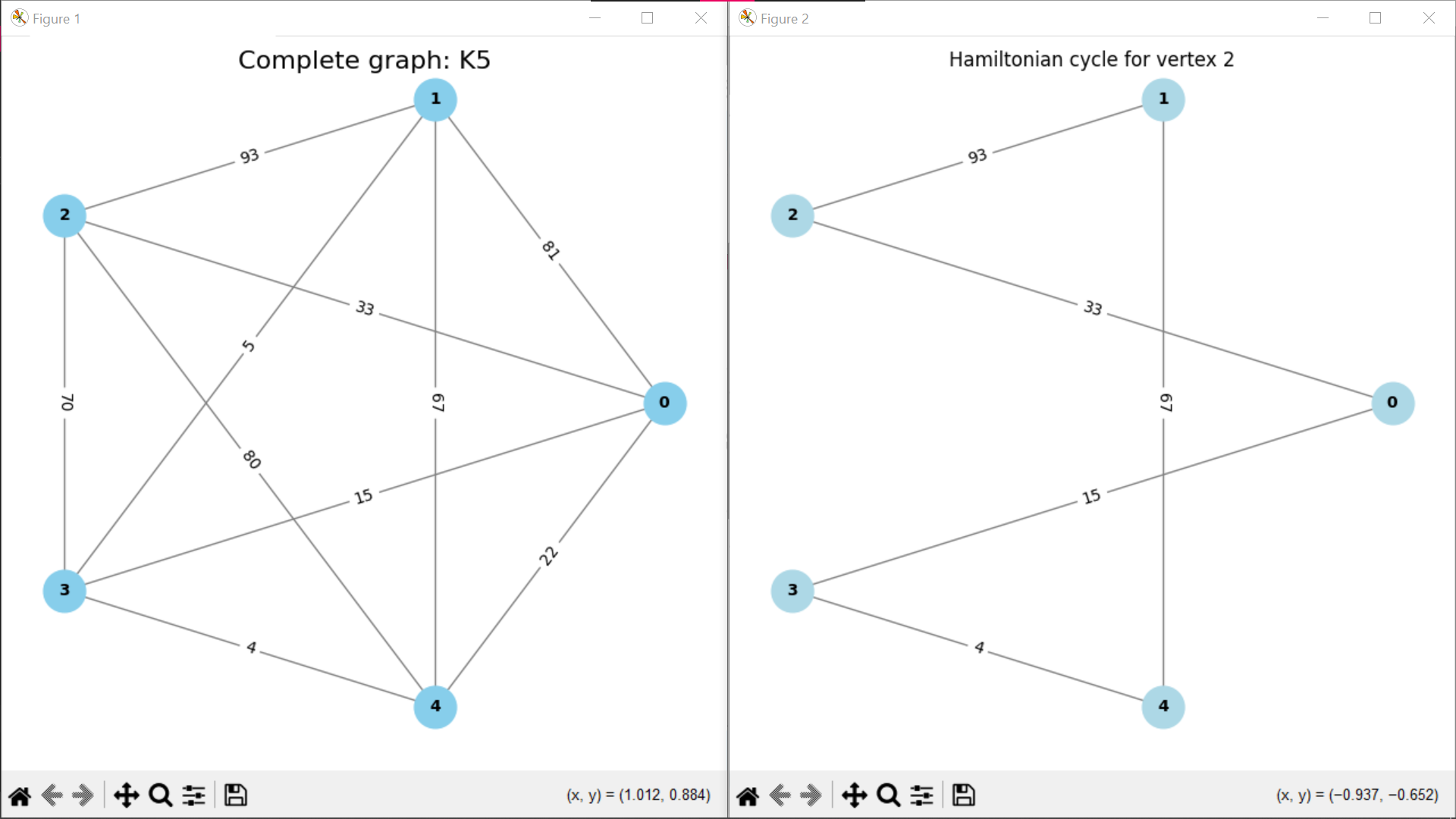
* These graphs illustrate runtime differences across programming languages for the same dataset.
* Users can analyze how each language performs relative to the others, offering insights into computational efficiency and implementation overhead.

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

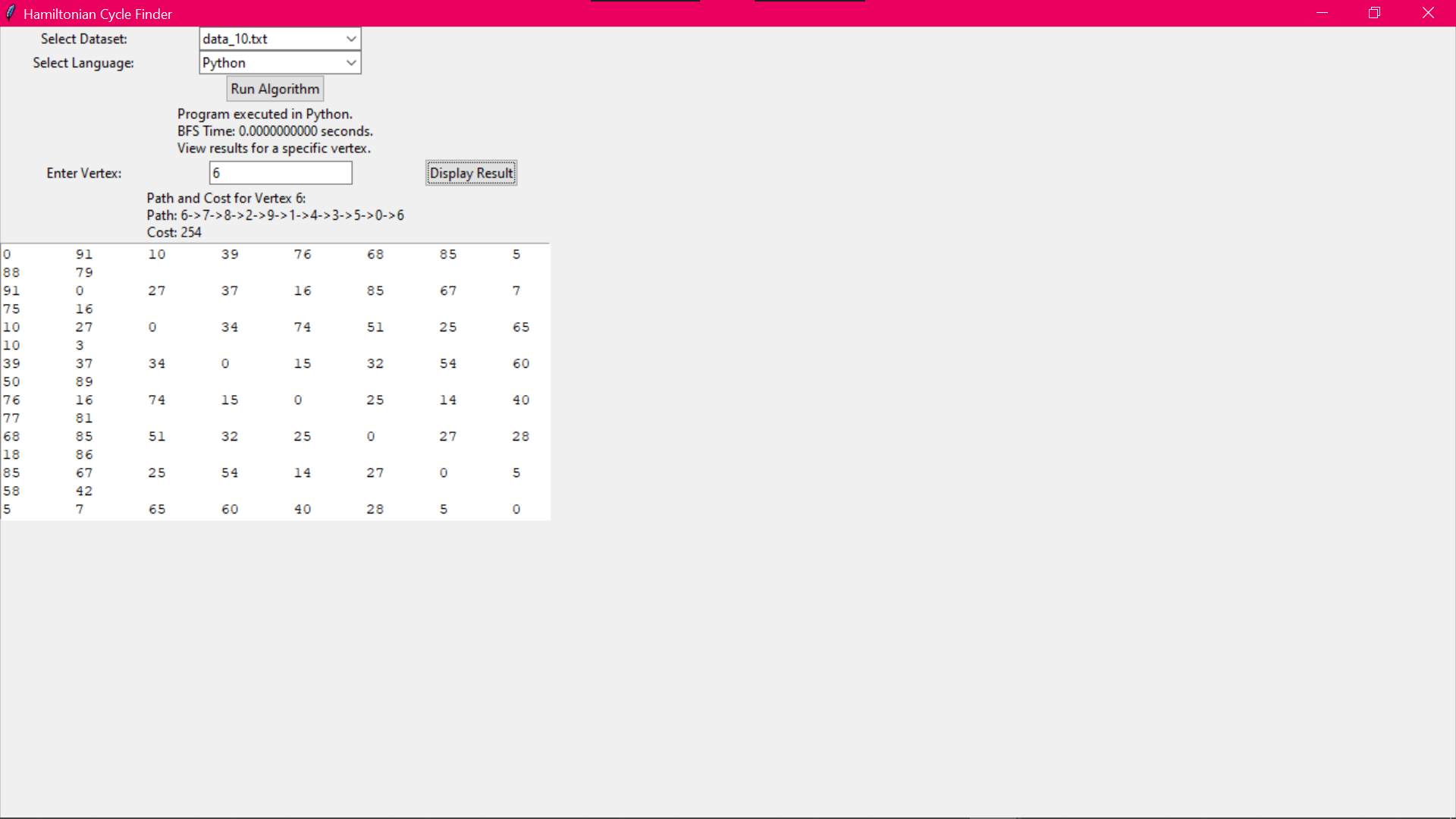


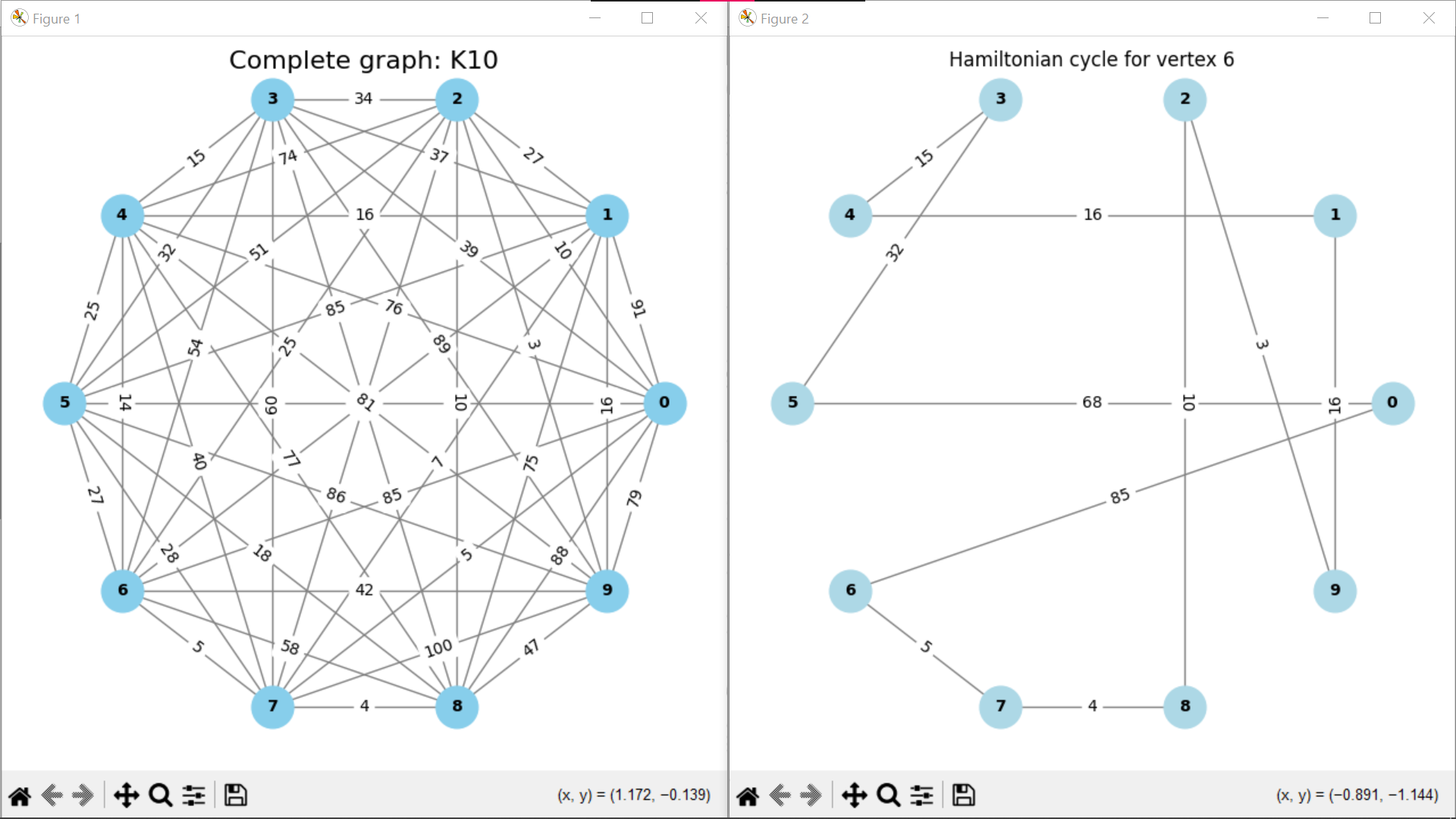
**Sample Run on Complete Graph on 5 vertices**





**Sample Run on Complete Graph on 10 vertices**





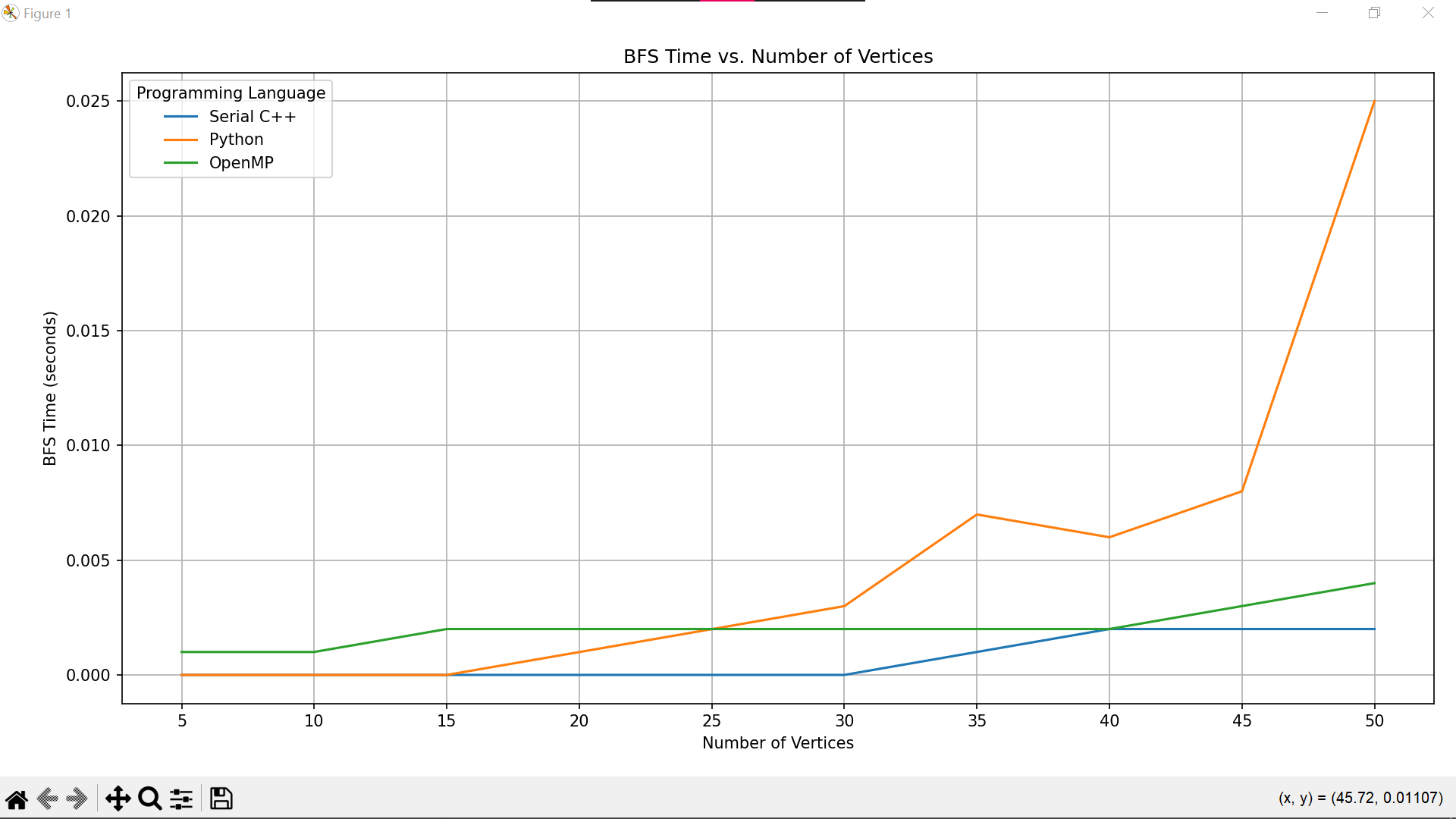
**Runtime Comparison for Sample Datasets**

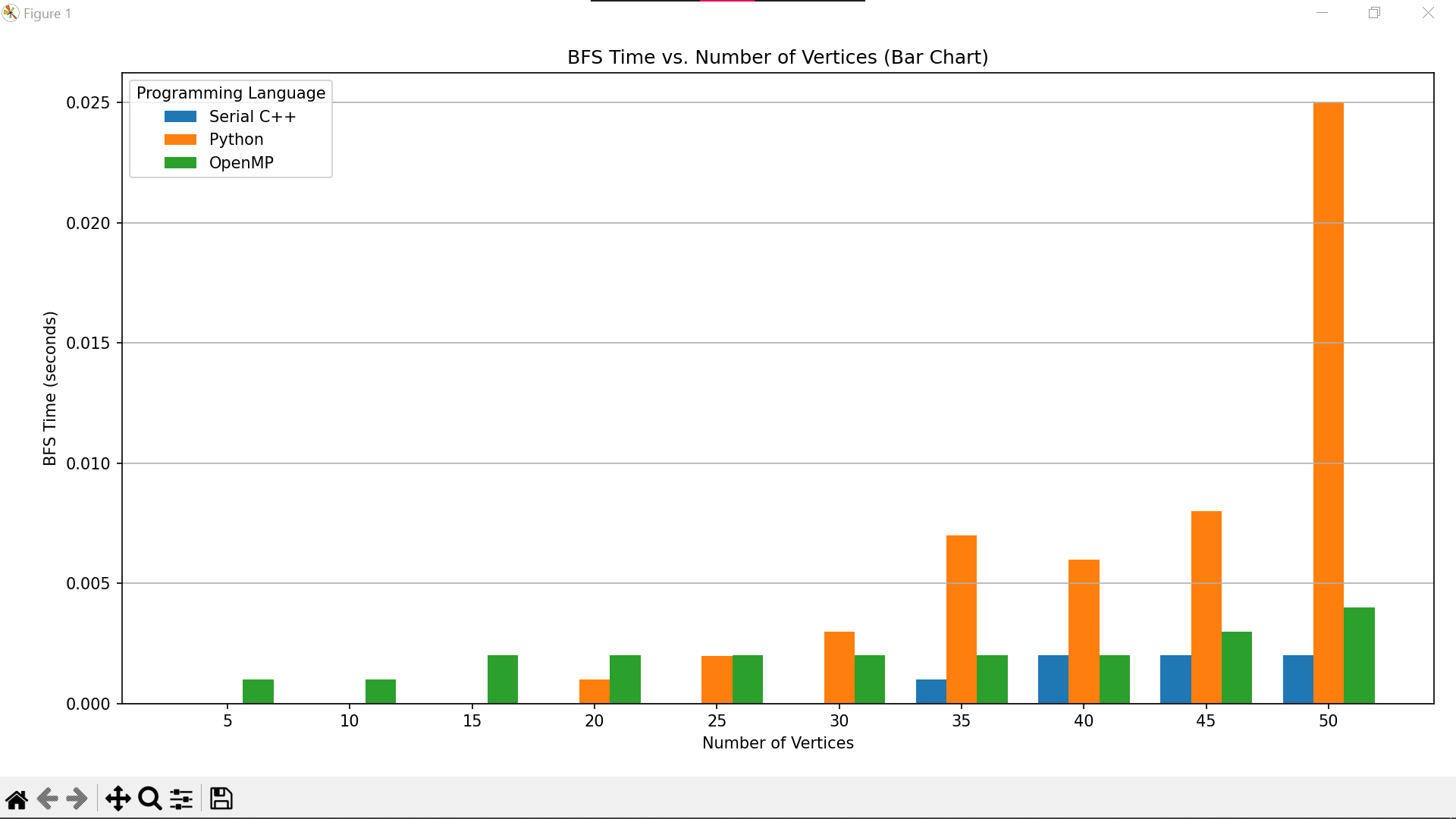
**Runtime Metrics**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **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.