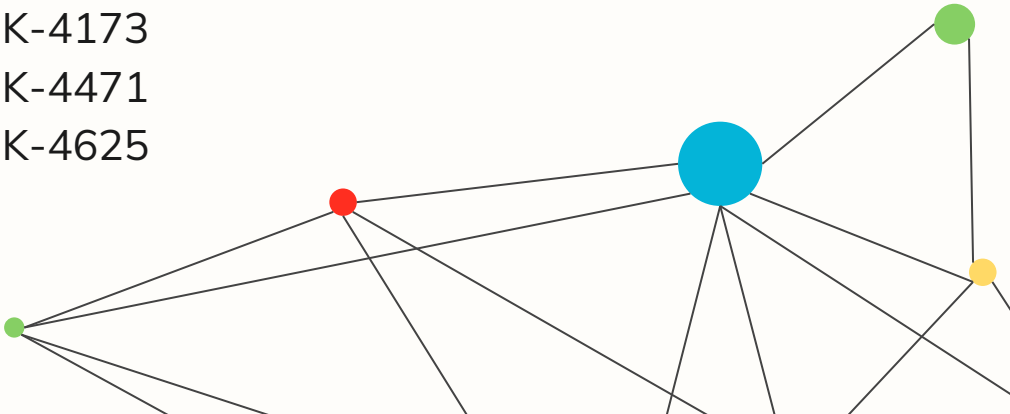
BCS-5G

Traveling Salesman Problem using Nearest Neighbour Algorithm

22K-4173
22K-4471
22K-4625



Introduction

What is the Nearest-Neighbour Algorithm?

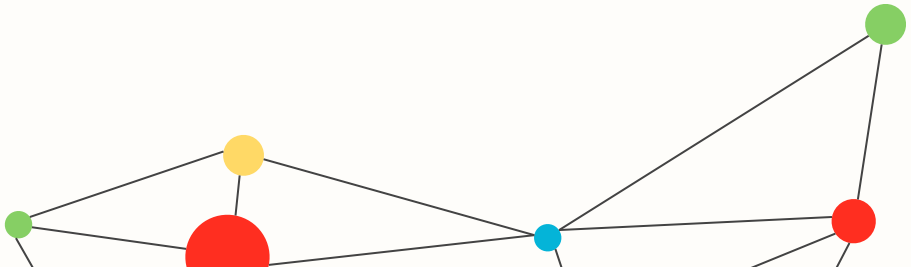
- Greedy algorithm to find an approximate solution to the Hamiltonian Cycle problem.
- Visits the nearest unvisited vertex until all vertices are covered.

Intuition & Need

- Finding an optimal solution can be time-consuming for large graphs.
- Provides a quick, approximate solution by making local optimal choices.

Applications

- Traveling Salesman Problem (TSP)
- Vehicle Routing Problems (VRP)
- Network Design



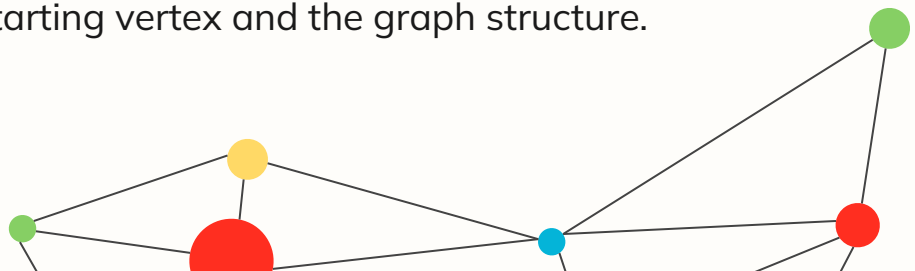
How the Nearest Neighbor Algorithm Works

Step-by-Step Process

- Start at a random vertex.
- Select the closest unvisited vertex.
- Mark the selected vertex as visited.
- Repeat until all vertices are visited.
- Return to the starting vertex to complete the cycle.

Key Points

- Simple and fast to implement.
- Doesn’t guarantee the optimal solution, but provides a feasible one quickly.
- The quality of the solution depends on the starting vertex and the graph structure.



Limitations and Performance

Limitations

- Starting Vertex Sensitivity:
  The cycle's total cost can vary significantly depending on the initial starting point.

- Greedy Nature:
  Makes decisions based on local information, which may not lead to a global optimum.

Performance

- Time Complexity:
  $O(n^2)$ due to the need to evaluate the nearest neighbor for each vertex.

- Scalability:
  Performs well on small to medium-sized graphs but may struggle with very large ones.

- Comparison to Exact Algorithms:
  Much faster than exact methods, but the quality of the solution may be much worse.

Applications

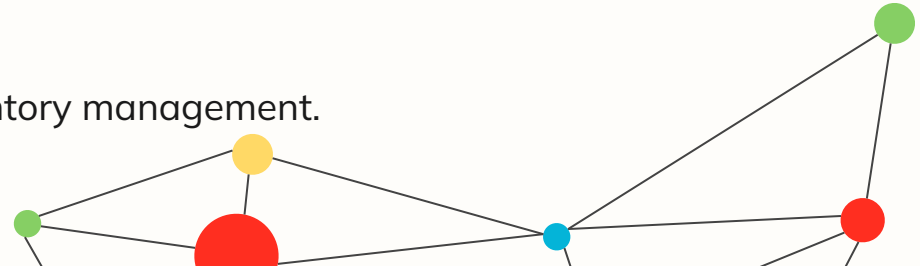
- **Traveling Salesman Problem (TSP)**
  Used as a heuristic to find a near-optimal path for a salesman visiting each city once.

- **Vehicle Routing Problem (VRP)**
  Helps optimize delivery routes for a fleet of vehicles.

- **Network Design**
  Applied in designing efficient communication paths in networks.

- **Robotics**
  Used in path planning for mobile robots to find efficient routes.

- **Logistics & Supply Chain**
  Helps optimize delivery schedules and inventory management.



Project Implementation – GUI Overview

User-Friendly Interface

- Allows selection of programming language (C++, Python, or OpenMP).
- Enables dataset selection for Hamiltonian Cycle execution.
- Visualizes the dataset as a complete graph with edge weights.

Visualization Features

- Complete Graph: Displays vertices and weighted edges.
- Hamiltonian Cycle: Highlights the cycle path and edge weights.

Project Implementation – Performance Metrics

Performance Logging

- Logs runtime in milliseconds for each algorithm run.
- Records the number of vertices and programming language used.

Performance Comparison

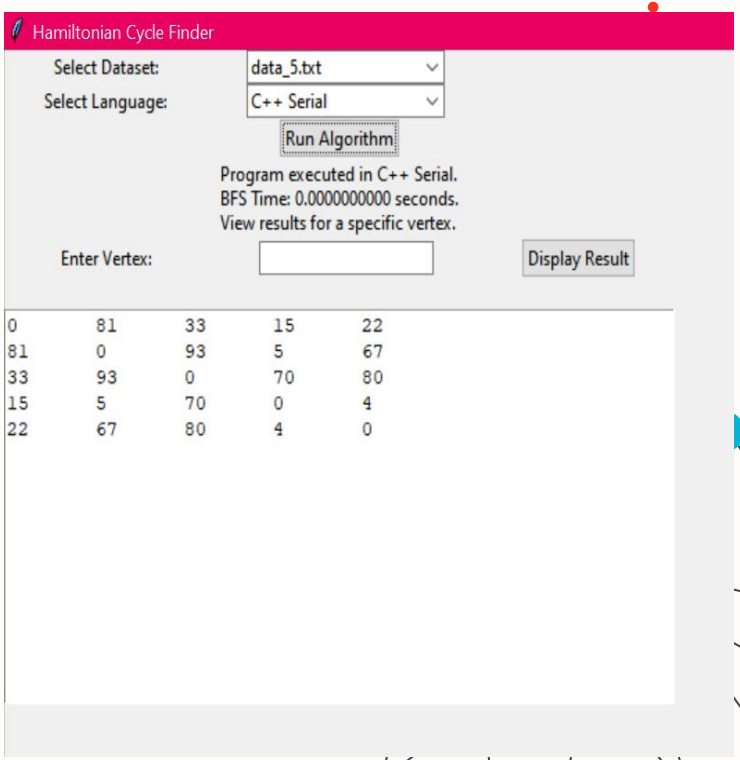
- Compares runtimes across languages (C++, Python, OpenMP).
- Graphs illustrate computational efficiency and implementation overhead.

Project Demonstration on Complete Graph K5

Dataset & Language Selection
Select the dataset (e.g., data_5.txt) and programming language (e.g., C++ Serial) from the dropdown menus.
The adjacency matrix is displayed.

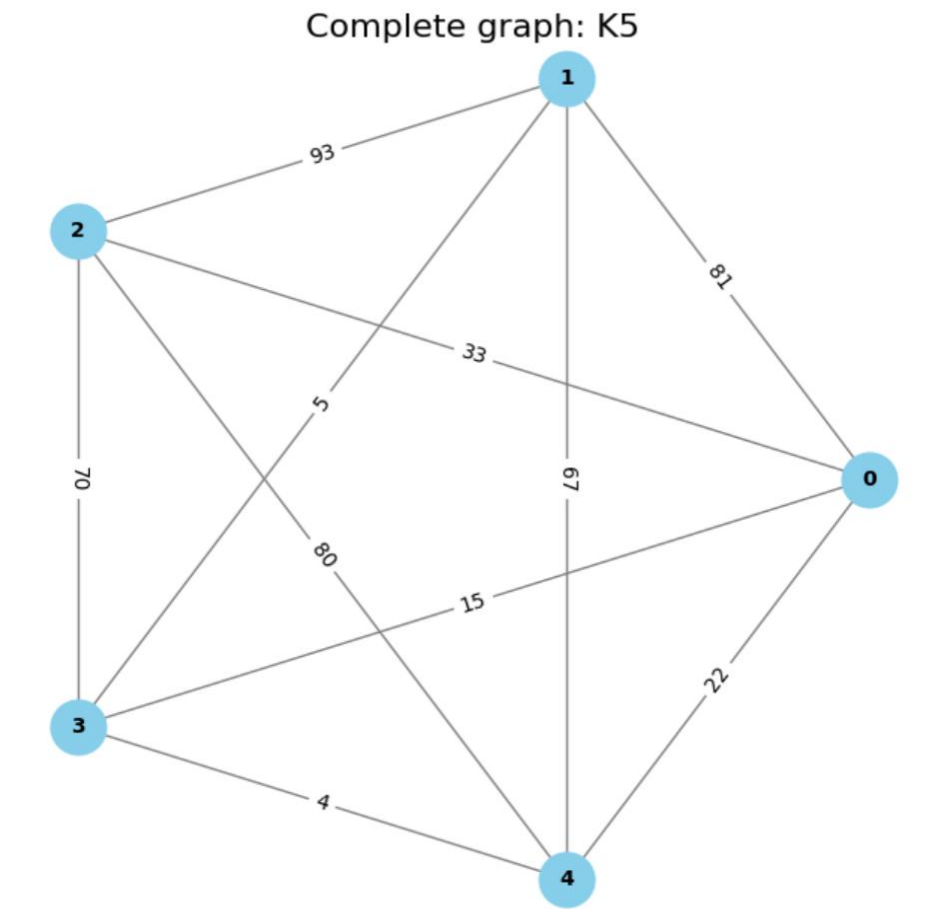
Run the Algorithm
Click Run Algorithm to execute the Hamiltonian cycle algorithm.

Execution Time Display
The execution time for the Hamiltonian cycle computation is shown under "Program executed in C++ Serial."



Project Demonstration on Complete Graph K5

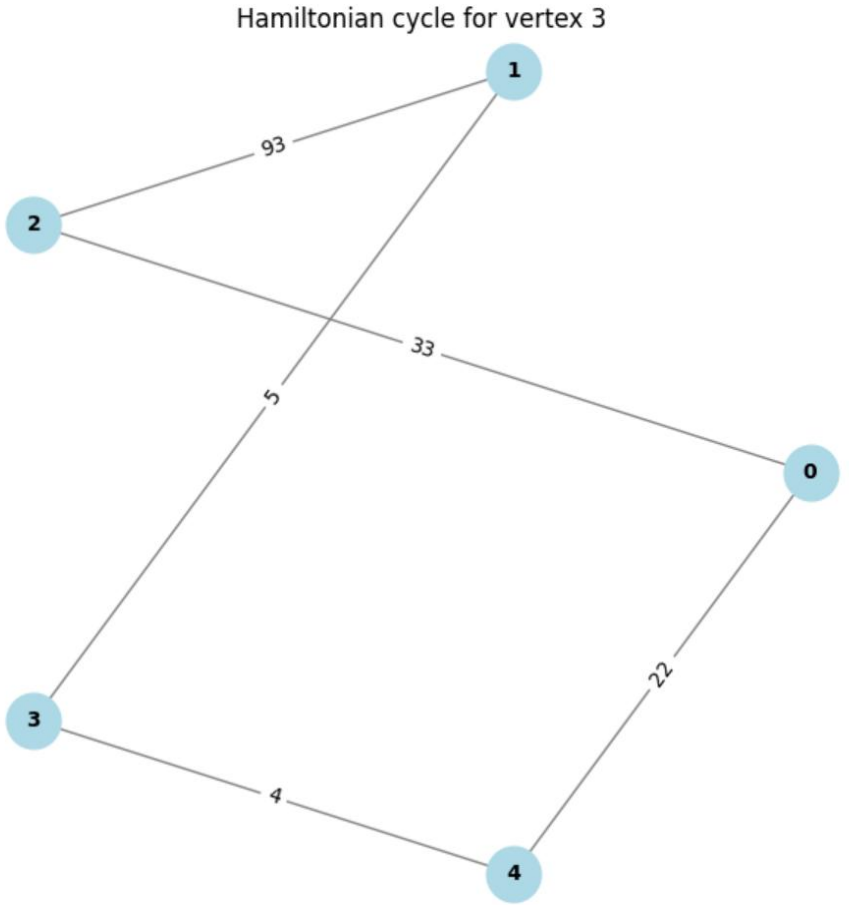
Complete Graph Visualization:
The complete graph, represented by the adjacency matrix, is displayed first, showing the vertices and edge weights for the selected dataset.



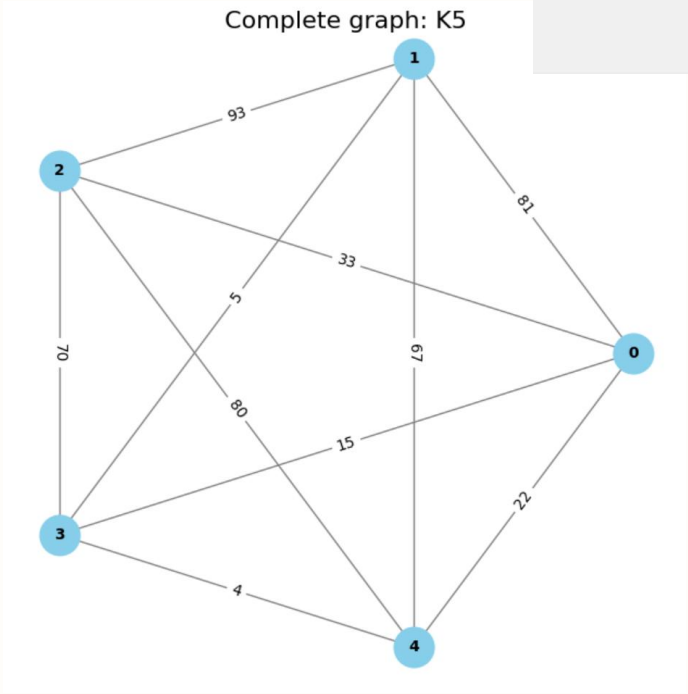
Project Demonstration on Complete Graph K5

Vertex Selection
Enter a vertex and click Display Result.
Cycle Path: The path is shown in the GUI.

Cycle Visualization
A new window displays the Hamiltonian cycle with highlighted edges.

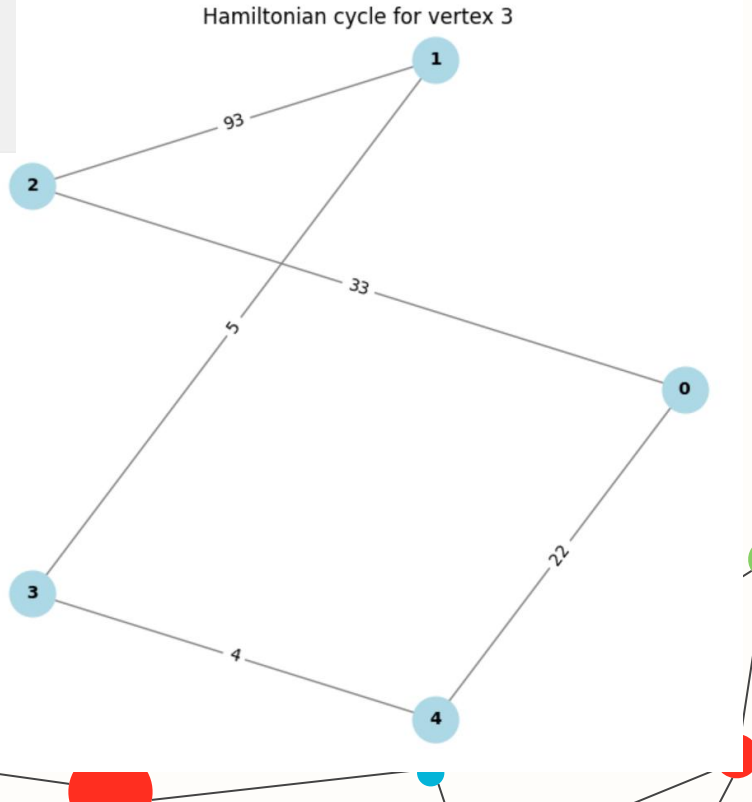


Project Demonstration on Complete Graph K5 – Hamiltonian Cycle Visual Verification



Enter Vertex: 3

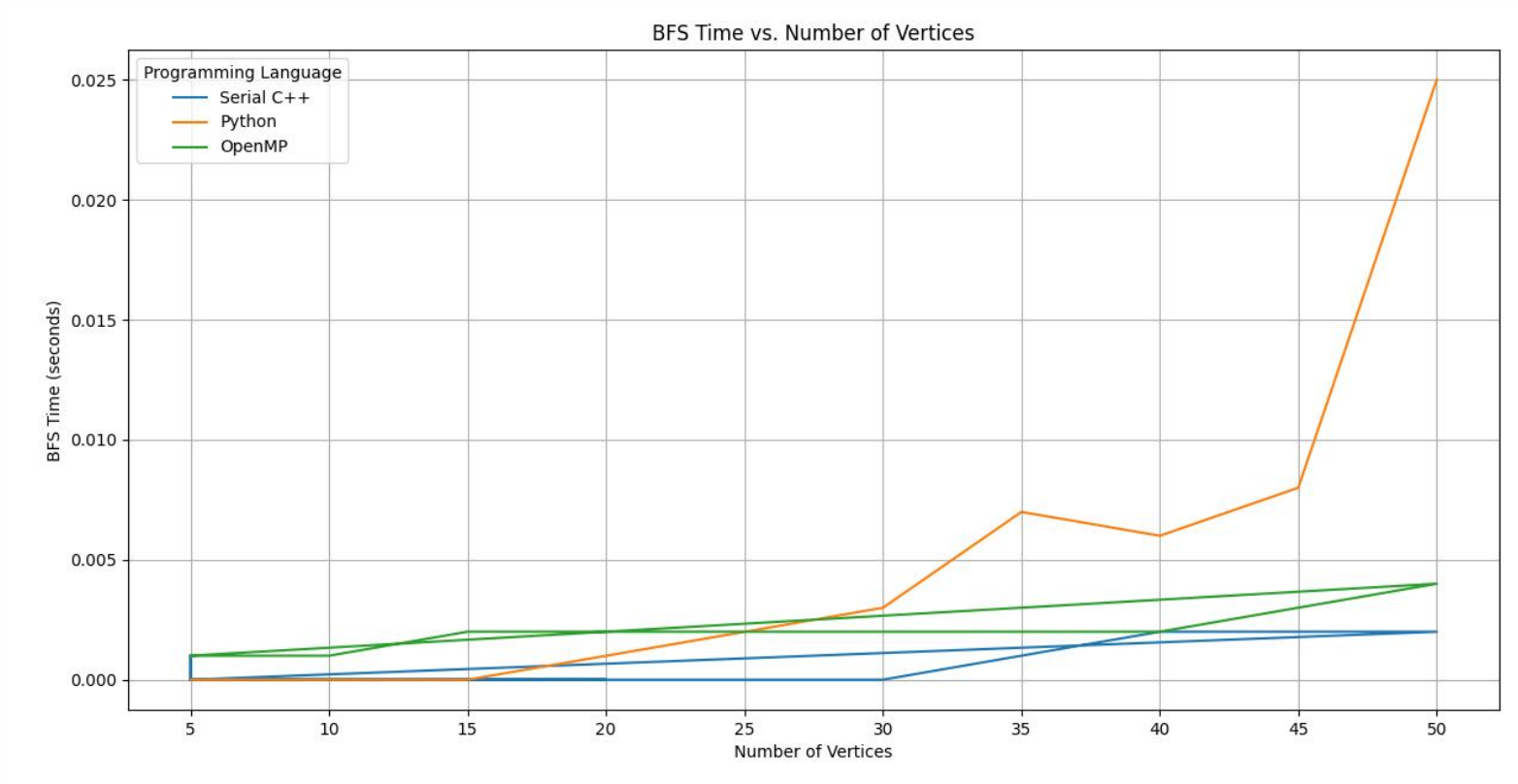
Path and Cost for Vertex 3:
Path: 3->4->0->2->1->3
Cost: 157



Performance Comparison: Runtimes Across Datasets and Languages

	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

Performance Comparison: Runtimes Across Datasets and Languages



Runtime Analysis & General Findings

C++ Serial

- Fast for small datasets due to compiled nature.
- Slower for large datasets due to lack of multi-core support.

Python

- Slower due to interpreted nature and dynamic typing.
- Performance drops significantly with large datasets.

OpenMP (Parallel C++)

- Consistent performance, especially with large datasets.
- Leverages multi-threading for efficient parallel processing.

Performance Summary by Programming Language

C++ Serial

- Fastest for small datasets due to optimized compilation.

OpenMP (Parallel C++)

- Best for larger datasets due to parallelization.

Python:

- Not suitable for large datasets due to slower performance.

Resources

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

Thanks!

CREDITS: This presentation template was created by **Slidesgo**, and includes icons by **Flaticon**, and infographics & images by **Freepik**

Please keep this slide for attribution