Assignment Specification: Generic Graph Library in C++ with Python Integration

Objective:
Develop a generic graph library in C++ that provides support for various graph algorithms and types. The library should be callable from Python.
Requirements:
Create a generic Graph class with the following:
Support for directed and undirected graphs
Support for node and edge creation
Detection of cycles
Implement the following graph algorithms:
Node colouring
Edge colouring
Detect and complete edges
Connected components
Graph centrality (Katz's)
Prim's algorithm
Kruskal's algorithm
Iterative Depth First Search (DFS)
Uniform Cost Search (UCS)
A* Search
Provide Python bindings using Pybind11 or similar library for the following:
Graph creation
Algorithm execution
Result retrieval
Test the library using sample code.

Guidelines:
Do not use any external graph libraries, such as Boost Graph Library.
Use the C++ Standard Template Library (STL) for container classes and algorithms where applicable.
Write clear and concise comments to explain the functionality of the code.
Make sure to follow good coding practices, such as using meaningful names for variables and functions, consistent indentation, and organizing code into appropriate files.

Class and Function Signatures:

```
class Graph {
public:
    Graph(bool directed);
    void addNode(const T& data);
    void addEdge(const T& src, const T& dest, const W& weight = W());
    bool hasCycle() const;
    // Algorithm implementations
    std::vector<int> nodeColoring() const;
    std::vector<int> edgeColoring() const;
    void completeEdges();
    std::vector<std::vector<T>> connectedComponents() const;
    std::vector<double> katzCentrality(double alpha, double beta) const;
    std::vector<Edge> primMST() const;
    std::vector<Edge> kruskalMST() const;
    std::vector<T> iterativeDFS(const T& start) const;
    std::vector<T> uniformCostSearch(const T& start, const T& goal) const;
    std::vector<T> aStarSearch(const T& start, const T& goal, std::function<double(T, T)> heuristic)
const;
};

// Pybind11 integration (inside a separate .   file)
PYBIND11_MODULE(graphlib, m) {
    py::class_<Graph>(m, "Graph")
        // Constructors, member functions, etc.
        ;
}
```

Sample Test Code (Python):

```
import graphlib

# Create an undirected graph
g = graphlib.Graph(False)

# Add nodes
g.addNode("A")
g.addNode("B")
g.addNode("C")

# Add edges
g.addEdge("A", "B", 1)
g.addEdge("B", "C", 2)

# Run algorithms
connected_components = g.connectedComponents()
katz_centrality = g.katzCentrality(0.1, 1)
prim_mst = g.primMST()

# Print results
print("Connected Components:", connected_components)
print("Katz Centrality:", katz_centrality)
print("Prim's MST:", prim_mst)
```

Submission:
Provide the complete source code for the library and Python integration.
Include a README.md file with instructions on how to compile and use the library.
Include a requirements.txt file specifying the required Python packages for testing.

Additional Class and Function Signatures:

To support generic node and edge types, the Graph class should be templated on two types: T for the node data type and W for the edge weight type. Furthermore, you should define an Edge class for representing edges with source, destination, and weight. Here are the updated signatures:

```
template <typename T, typename W>
class Edge {
public:
   Edge(const T& src, const T& dest, const W& weight);
   T getSource() const;
   T getDestination() const;
   W getWeight() const;
};

template <typename T, typename W>
class Graph {
public:
   Graph(bool directed);
   void addNode(const T& data);
   void addEdge(const T& src, const T& dest, const W& weight = W());
   bool hasCycle() const;

   // Algorithm implementations
   std::vector<int> nodeColoring() const;
   std::vector<int> edgeColoring() const;
   void completeEdges();
   std::vector<std::vector<T>> connectedComponents() const;
   std::vector<double> katzCentrality(double alpha, double beta) const;
   std::vector<Edge<T, W>> primMST() const;
   std::vector<Edge<T, W>> kruskalMST() const;
   std::vector<T> iterativeDFS(const T& start) const;
   std::vector<T> uniformCostSearch(const T& start, const T& goal) const;
   std::vector<T> aStarSearch(const T& start, const T& goal, std::function<double(T, T)> heuristic)
const;
};

// Pybind11 integration (inside a separate .   file)
PYBIND11_MODULE(graphlib, m) {
   py::class_<Edge<std::string, int>>(m, "Edge")
      // Constructors, member functions, etc.
      ;

   py::class_<Graph<std::string, int>>(m, "Graph")
      // Constructors, member functions, etc.
      ;
}
```

Sample Test Code (Python):

```
import graphlib

# Create an undirected graph
g = graphlib.Graph(False)

# Add nodes
g.addNode("A")
g.addNode("B")
g.addNode("C")

# Add edges
g.addEdge("A", "B", 1)
g.addEdge("B", "C", 2)

# Run algorithms
connected_components = g.connectedComponents()
katz_centrality = g.katzCentrality(0.1, 1)
prim_mst = g.primMST()

# Print results
print("Connected Components:", connected_components)
print("Katz Centrality:", katz_centrality)
print("Prim's MST:", [(e.getSource(), e.getDestination(), e.getWeight()) for e in prim_mst])
```

Note:

Ensure the Edge class is also exposed through Pybind11, allowing Python to work with the results of algorithms that return edges.
The Graph class is templated on both the node type and the edge weight type, but the example provided is for a specific instantiation (std::string for nodes and int for edge weights). You may need to provide multiple instantiations or use type erasure techniques to support various types in Python.
Remember to include a README.md file with instructions on how to compile and use the library and a requirements.txt file specifying the required Python packages for testing.

Additional Considerations:

Error handling:
Ensure that your library handles common error scenarios gracefully. For example, when adding an edge between non-existent nodes or attempting to run an algorithm on an empty graph, the library should return appropriate error messages or throw exceptions.

Performance optimization:
While implementing the graph algorithms, consider using appropriate data structures and algorithms to optimize the performance. For example, use a priority queue for Prim's and Kruskal's algorithms to speed up the process of finding the minimum-weight edge.

Testing and validation:
Test your library thoroughly with various input scenarios to ensure that it works correctly and efficiently. Validate the results of each algorithm against known solutions or by comparing with the output of existing graph libraries.

Documentation:
Provide comprehensive documentation for each class and function, describing the purpose, input parameters, return values, and any assumptions or constraints. This will make it easier for users to understand and use your library effectively.

Example usage:
Include examples demonstrating how to use the library and its algorithms, both in C++ and Python. This will serve as a valuable resource for users and help them get started quickly.

Submission Checklist:

Complete C++ source code for the generic graph library, including the Graph and Edge classes, and implementations of the required algorithms.
C++ source code for the Pybind11 integration, exposing the Graph and Edge classes and their functions to Python.
A README.md file containing instructions on how to compile, link, and use the library, both in C++ and Python.
A requirements.txt file specifying the required Python packages for testing the library.
Sample test code (in Python) demonstrating the use of the library and its algorithms, as well as validating the results.

Submission date is Last ISA5 day(Elective 4) + 2.