Graph Data Structure - Usage Guide

A comprehensive C++ graph implementation supporting 11 different graph types with generic type support (int, float, char, string, custom types).

Table of Contents

Quick Start

cpp

```cpp
#include "graph.h"

int main() {
    // Create a directed graph with integer vertices
    DirectedGraph<int> graph;

    // Add edges (vertices are auto-created)
    graph.addEdge(1, 2);
    graph.addEdge(2, 3);
    graph.addEdge(3, 4);

    // Display the graph
    graph.display();

    return 0;
}
```

Compile:

```bash
g++ -std=c++11 your_program.cpp -o your_program
./your_program
```

Installation

1. **Download** the `graph.cpp` file
2. **Include** in your project:
   - Option A: Use as header by renaming to `graph.h`
   - Option B: Compile separately and link
3. **Compile** with C++11 or later

```bash
# Option A: Single file compilation
g++ -std=c++11 main.cpp -o program

# Option B: Separate compilation
g++ -std=c++11 -c graph.cpp -o graph.o
g++ -std=c++11 main.cpp graph.o -o program
```

Basic Usage

Creating a Graph

```cpp
// Syntax: GraphType<VertexType> graphName;

DirectedGraph<int> intGraph;          // Directed graph with integers
UndirectedGraph<string> stringGraph;  // Undirected graph with strings
WeightedGraph<char> charGraph(true);  // Weighted directed graph with chars
```

Adding Vertices

```cpp
```

```cpp
// Vertices are automatically added when creating edges
graph.addEdge(1, 2);  // Adds vertices 1 and 2 if they don't exist

// For graphs that need explicit vertex addition (Null, Complete):
NullGraph<int> nullG;
nullG.addVertex(1);
nullG.addVertex(2);
nullG.addVertex(3);
```

Adding Edges

cpp

```cpp
// Unweighted graph
graph.addEdge(source, destination);

// Weighted graph
weightedGraph.addEdge(source, destination, weight);

// Examples:
DirectedGraph<int> dg;
dg.addEdge(1, 2);       // Edge from 1 to 2

WeightedGraph<string> wg(true);  // directed
wg.addEdge("A", "B", 10);       // A -> B with weight 10
```

Displaying Graphs

cpp

```cpp
graph.display();  // Prints adjacency list representation
```

Getting Graph Information

cpp

```cpp
int vertices = graph.getNumVertices();  // Get vertex count
int edges = graph.getNumEdges();        // Get edge count
bool connected = graph.isConnected();    // Check connectivity
set<T> verts = graph.getVertices();     // Get all vertices
```

Graph Types and Use Cases

1. Null Graph - No Edges

```cpp
NullGraph<int> graph;
graph.addVertex(1);
graph.addVertex(2);
graph.addVertex(3);
// Cannot add edges
```

Use Cases:

- Initial state in graph construction algorithms
- Representing isolated entities
- Testing vertex-only operations

---

2. Trivial Graph - Single Vertex

```cpp
TrivialGraph<char> graph('A');
// Single vertex, no edges
```

Use Cases:

- Base case in recursive algorithms
- Singleton pattern implementations
- Minimal graph testing

---

3. Undirected Graph - Bidirectional Edges

```cpp
UndirectedGraph<string> socialNetwork;
socialNetwork.addEdge("Alice", "Bob");     // Alice ↔ Bob
socialNetwork.addEdge("Bob", "Charlie");   // Bob ↔ Charlie
```

Use Cases:

- **Social Networks**: Friendships (symmetric relationships)

- **Road Networks**: Bidirectional roads
- **Computer Networks**: Peer-to-peer connections
- **Collaboration Graphs**: Co-authorship, team projects

Example: Social Network

```cpp
UndirectedGraph<string> friends;
friends.addEdge("Alice", "Bob");
friends.addEdge("Bob", "Charlie");
friends.addEdge("Alice", "David");

if (friends.isConnected()) {
    cout << "Everyone is connected through mutual friends!" << endl;
}
```

---

4. Directed Graph - One-way Edges

```cpp
DirectedGraph<int> webGraph;
webGraph.addEdge(1, 2);  // Page 1 -> Page 2
webGraph.addEdge(2, 3);  // Page 2 -> Page 3
webGraph.addEdge(3, 1);  // Page 3 -> Page 1
```

Use Cases:

- **Web Page Links**: Hyperlinks (one-way)
- **Twitter Followers**: A follows B (not necessarily vice versa)
- **State Machines**: State transitions
- **Dependencies**: Module/package dependencies
- **Game AI**: State and decision trees

Example: Task Dependencies

```cpp
DirectedGraph<string> tasks;
tasks.addEdge("Design", "Development");
tasks.addEdge("Development", "Testing");
tasks.addEdge("Testing", "Deployment");
```

5. Connected Graph - All Vertices Reachable

cpp

```cpp
ConnectedGraph<int> network;
network.addEdge(1, 2);
network.addEdge(2, 3);
network.addEdge(3, 4);
network.display(); // Shows connectivity status
```

Use Cases:

- **Network Reliability**: Ensure all nodes can communicate
- **Transportation Networks**: All cities reachable
- **Sensor Networks**: Full coverage validation
- **Communication Systems**: No isolated components

Validation:

cpp

```cpp
if (network.isConnected()) {
    cout << "Network is fully connected!" << endl;
} else {
    cout << "Warning: Network has isolated components!" << endl;
}
```

6. Disconnected Graph - Isolated Components

cpp

```cpp
DisconnectedGraph<string> regions;
// Component 1
regions.addEdge("NYC", "Boston");
regions.addEdge("Boston", "Philly");

// Component 2 (isolated)
regions.addEdge("LA", "SF");
```

Use Cases:

- **Cluster Analysis**: Separate communities

- **Island Detection**: Geographic separation
- **Component Analysis**: Finding isolated subsystems
- **Network Failure Modeling**: Simulating disconnections

---

7. Complete Graph - All Vertices Connected

cpp

```cpp
CompleteGraph<char> fullMesh;
fullMesh.addVertex('A');
fullMesh.addVertex('B');
fullMesh.addVertex('C');
fullMesh.addVertex('D');
// Automatically creates all possible edges
```

Use Cases:

- **Fully Connected Networks**: Mesh topology
- **Tournament Scheduling**: Round-robin (everyone plays everyone)
- **Clique Detection**: Finding fully connected subgraphs
- **Broadcast Networks**: Every node can reach every other node

Properties:

- N vertices → N(N-1)/2 edges
- Maximum connectivity
- No single point of failure

---

8. Cyclic Graph - Contains Cycles

cpp

```cpp
CyclicGraph<int> circuit;
circuit.addEdge(1, 2);
circuit.addEdge(2, 3);
circuit.addEdge(3, 4);
circuit.addEdge(4, 1);  // Creates cycle

if (circuit.hasCycle()) {
    cout << "Cycle detected!" << endl;
}
```

Use Cases:

- **Circular Dependencies**: Detecting problematic circular references
- **Circular Routes**: Delivery circuits, patrol routes
- **Feedback Systems**: Control systems with feedback loops
- **Deadlock Detection**: Finding circular wait conditions

---

9. Directed Acyclic Graph (DAG) - No Cycles

```cpp
DirectedAcyclicGraph<string> buildSystem;
buildSystem.addEdge("Source", "Compile");
buildSystem.addEdge("Compile", "Link");
buildSystem.addEdge("Link", "Deploy");
// buildSystem.addEdge("Deploy", "Source");  // Would throw error!
```

Use Cases:

- **Build Systems**: Makefile dependencies, compilation order
- **Task Scheduling**: Project management (PERT/CPM)
- **Version Control**: Git commit history
- **Data Processing Pipelines**: ETL workflows
- **Course Prerequisites**: Academic planning
- **Inheritance Hierarchies**: Class relationships

Example: Build Pipeline

```cpp
```

```
DirectedAcyclicGraph<string> pipeline;
pipeline.addEdge("fetch_data", "clean_data");
pipeline.addEdge("clean_data", "transform_data");
pipeline.addEdge("transform_data", "load_data");
pipeline.addEdge("load_data", "generate_report");
```

Cycle Prevention:

```cpp
try {
    dag.addEdge("Deploy", "Source");  // Would create cycle
} catch (const logic_error& e) {
    cout << "Cannot add edge: " << e.what() << endl;
}
```

---

10. Bipartite Graph - Two-Set Division

```cpp
BipartiteGraph<string> matching;
// Set 1: Students
// Set 2: Courses
matching.addEdge("Alice", "Math");
matching.addEdge("Alice", "Physics");
matching.addEdge("Bob", "Math");
matching.addEdge("Charlie", "Physics");
```

Use Cases:

- **Job Matching**: Candidates ↔ Jobs
- **Student-Course**: Enrollment systems
- **Resource Allocation**: Tasks ↔ Workers
- **Network Flow**: Source ↔ Sink problems
- **Recommendation Systems**: Users ↔ Items
- **Dating Apps**: People ↔ Compatible matches

Example: Job Assignment

```cpp
```

```
BipartiteGraph<string> jobMatch;
jobMatch.addEdge("Developer_A", "Project_1");
jobMatch.addEdge("Developer_A", "Project_3");
jobMatch.addEdge("Developer_B", "Project_2");
jobMatch.addEdge("Designer_C", "Project_1");
```

Validation:

```cpp
// Graph validates bipartite property automatically
try {
    biGraph.addEdge("Student1", "Student2");  // Might break bipartiteness
} catch (const logic_error& e) {
    cout << "Edge would violate bipartite property!" << endl;
}
```

---

11. Weighted Graph - Edges with Costs

```cpp
WeightedGraph<string> routes(true);  // directed
routes.addEdge("NYC", "Boston", 215);      // 215 miles
routes.addEdge("Boston", "Philly", 305);  // 305 miles
routes.addEdge("NYC", "Philly", 95);       // 95 miles
```

Use Cases:

- **Navigation Systems**: Roads with distances/time
- **Network Routing**: Links with bandwidth/latency
- **Flight Networks**: Routes with prices/duration
- **Logistics**: Shipping costs, delivery time
- **Social Networks**: Relationship strength
- **Shortest Path Problems**: Dijkstra, Bellman-Ford

Example: Shipping Network

```cpp
```

```cpp
WeightedGraph<string> shipping(true);
shipping.addEdge("Warehouse", "Store_A", 50);   // $50 shipping
shipping.addEdge("Warehouse", "Store_B", 75);   // $75 shipping
shipping.addEdge("Store_A", "Store_B", 25);     // $25 shipping
```

Undirected vs Directed:

```cpp
// Undirected weighted graph (bidirectional costs)
WeightedGraph<string> roads(false);
roads.addEdge("CityA", "CityB", 100);  // Same cost both ways

// Directed weighted graph (one-way or different costs)
WeightedGraph<string> flights(true);
flights.addEdge("NYC", "LA", 300);     // NYC → LA: $300
flights.addEdge("LA", "NYC", 250);     // LA → NYC: $250 (different!)
```

Method Reference

Constructor Parameters

Graph Type	Template	Constructor Args	Notes
`Graph<T>`	Required	`bool directed, bool weighted`	Base class
`NullGraph<T>`	Required	None	No edges allowed
`TrivialGraph<T>`	Required	`T vertex`	Single vertex
`UndirectedGraph<T>`	Required	None	Bidirectional edges
`DirectedGraph<T>`	Required	None	One-way edges
`ConnectedGraph<T>`	Required	None	Validates connectivity
`DisconnectedGraph<T>`	Required	None	Allows isolated components
`CompleteGraph<T>`	Required	None	Auto-generates edges
`CyclicGraph<T>`	Required	`bool directed = false`	Detects cycles
`DirectedAcyclicGraph<T>`	Required	None	Prevents cycles
`BipartiteGraph<T>`	Required	None	Validates two-set property
`WeightedGraph<T>`	Required	`bool directed = false`	Edges with weights

Core Methods

`addVertex(T vertex)`

Explicitly adds a vertex to the graph.

```cpp
CompleteGraph<int> graph;
graph.addVertex(1);  // Add vertex 1
graph.addVertex(2);  // Add vertex 2 (auto-connects to 1)
```

`addEdge(T src, T dest, int weight = 1)`

Adds an edge between two vertices.

Parameters:

- `src`: Source vertex
- `dest`: Destination vertex
- `weight`: Edge weight (default = 1)

```cpp
// Unweighted
graph.addEdge(1, 2);

// Weighted
weightedGraph.addEdge("A", "B", 50);
```

Auto-creates vertices if they don't exist.

`display() const`

Prints the graph's adjacency list.

```cpp
graph.display();
```

Output formats:

- Directed: `vertex --> neighbor1, neighbor2`
- Undirected: `vertex --- neighbor1, neighbor2`
- Weighted: `vertex --(weight)--> neighbor`

`getNumVertices() const`

Returns the number of vertices.

```cpp
int count = graph.getNumVertices();
```

getNumEdges() const

Returns the number of edges.

```cpp
int edges = graph.getNumEdges();
// For undirected graphs, counts each edge once
```

isConnected() const

Checks if all vertices are reachable from any vertex.

```cpp
if (graph.isConnected()) {
    cout << "Graph is fully connected" << endl;
}
```

getVertices() const

Returns a set of all vertices.

```cpp
set<string> vertices = graph.getVertices();
for (const auto& v : vertices) {
    cout << v << " ";
}
```

Graph Property Methods (New!)

getDegree(T vertex) const

Gets the degree of a specific vertex (number of edges connected to it).

```cpp
```

```cpp
UndirectedGraph<string> social;
social.addEdge("Alice", "Bob");
social.addEdge("Alice", "Charlie");
social.addEdge("Alice", "David");

int degree = social.getDegree("Alice");  // Returns 3
cout << "Alice has " << degree << " connections" << endl;
```

For directed graphs: Returns out-degree (outgoing edges).

`getInDegree(T vertex) const`

Gets the in-degree of a vertex (incoming edges). For directed graphs only.

```cpp
DirectedGraph<int> web;
web.addEdge(1, 2);
web.addEdge(3, 2);
web.addEdge(4, 2);

int inDeg = web.getInDegree(2);   // Returns 3
int outDeg = web.getDegree(2);    // Returns 0
```

For undirected graphs: Same as `getDegree()`.

`getMinDegree() const`

Returns the minimum degree among all vertices (also known as graph's minimum connectivity).

```cpp
UndirectedGraph<int> graph;
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(2, 4);
graph.addEdge(3, 4);

int minDeg = graph.getMinDegree();  // Returns 1 (vertex 1 has degree 1)
```

Use Cases:

- Network vulnerability analysis (vertex with min degree is potential bottleneck)
- Finding leaf nodes or peripheral vertices

`getMaxDegree() const`

Returns the maximum degree among all vertices (highest connectivity).

```cpp
int maxDeg = graph.getMaxDegree();  // Returns 3 (vertex 2 has degree 3)
```

Use Cases:

- Finding hub nodes in networks
- Identifying most connected entities

`getDistance(T src, T dest) const`

Calculates the shortest path distance between two vertices using BFS.

```cpp
UndirectedGraph<string> cities;
cities.addEdge("NYC", "Boston");
cities.addEdge("Boston", "Portland");
cities.addEdge("NYC", "Philly");

int dist = cities.getDistance("NYC", "Portland");  // Returns 2
cout << "Shortest path: " << dist << " hops" << endl;

int noPath = cities.getDistance("NYC", "LA");  // Returns -1 (no path)
```

Returns:

- Number of edges in shortest path
- `-1` if no path exists
- `0` if source equals destination

Note: This counts edges (hops), not weights. For weighted shortest paths, implement Dijkstra's algorithm separately.

`getRadius() const`

Calculates the graph radius (minimum eccentricity).

```cpp
int radius = graph.getRadius();
```

Definition:

- Eccentricity of vertex v = maximum distance from v to any other vertex
- Radius = minimum eccentricity among all vertices
- Represents the "center" distance of the graph

Returns:

- Graph radius value
- `-1` if graph is disconnected
- `0` if graph has 0 or 1 vertices

Use Cases:

- Finding optimal center location (e.g., warehouse placement)
- Network design optimization

`getDiameter() const`

Calculates the graph diameter (maximum shortest path between any two vertices).

```cpp
int diameter = graph.getDiameter();
```

Definition: Maximum distance between any pair of vertices.

Returns:

- Graph diameter value
- `-1` if graph is disconnected
- `0` if graph has 0 or 1 vertices

Use Cases:

- Network latency analysis (worst-case communication time)
- Social network "six degrees of separation"

Relationship: `Radius ≤ Diameter ≤ 2 × Radius`

`getGirth() const`

Calculates the length of the shortest cycle in the graph.

```cpp
```

```cpp
UndirectedGraph<int> graph;
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(3, 4);
graph.addEdge(4, 1);  // Creates 4-cycle
graph.addEdge(2, 4);  // Creates 3-cycle

int girth = graph.getGirth();  // Returns 3 (smallest cycle)
```

Returns:

- Length of shortest cycle
- `-1` if graph is acyclic (no cycles)

Use Cases:

- Graph theory analysis
- Network loop detection
- Cycle-based optimization

`getCircumference() const`

Calculates the length of the longest cycle in the graph.

```cpp
int circumference = graph.getCircumference();
```

Returns:

- Length of longest cycle
- `-1` if graph is acyclic

Use Cases:

- Finding longest circular paths
- Tour planning
- Network resilience analysis

`displayProperties() const`

Displays a comprehensive summary of all graph properties.

```cpp
```

```
graph.displayProperties();
```

Output Example:

```
=== Graph Properties ===
Number of Vertices: 5
Number of Edges: 6
Minimum Degree (Min vertex connections): 1
Maximum Degree (Max vertex connections): 3
Graph Radius (Min eccentricity): 2
Graph Diameter (Max shortest path): 3
Girth (Shortest cycle): 3
Circumference (Longest cycle): 5
Connected: Yes
========================
```

Use Cases:

- Quick graph analysis
- Debugging and validation
- Comparing different graph structures

Special Methods

`hasCycle() const` - CyclicGraph only

Detects if the graph contains any cycle.

```cpp
CyclicGraph<int> graph;
graph.addEdge(1, 2);
graph.addEdge(2, 3);
graph.addEdge(3, 1);  // Creates cycle

if (graph.hasCycle()) {
    cout << "Cycle detected!" << endl;
}
```

Advanced Examples

Example 1: Social Network Analysis

cpp

```cpp
UndirectedGraph<string> socialNetwork;

// Add friendships
socialNetwork.addEdge("Alice", "Bob");
socialNetwork.addEdge("Bob", "Charlie");
socialNetwork.addEdge("Charlie", "David");
socialNetwork.addEdge("David", "Alice");
socialNetwork.addEdge("Alice", "Eve");

// Check connectivity
if (socialNetwork.isConnected()) {
    cout << "All users are connected through mutual friends" << endl;
}

// Display network
socialNetwork.display();

// Get statistics
cout << "Total users: " << socialNetwork.getNumVertices() << endl;
cout << "Total friendships: " << socialNetwork.getNumEdges() << endl;
```

Example 2: Task Dependency System (DAG)

cpp

```
DirectedAcyclicGraph<string> projectTasks;

try {
    // Define task dependencies
    projectTasks.addEdge("Requirements", "Design");
    projectTasks.addEdge("Design", "Frontend");
    projectTasks.addEdge("Design", "Backend");
    projectTasks.addEdge("Frontend", "Integration");
    projectTasks.addEdge("Backend", "Integration");
    projectTasks.addEdge("Integration", "Testing");
    projectTasks.addEdge("Testing", "Deployment");

    // This would fail (creates cycle):
    // projectTasks.addEdge("Deployment", "Requirements");

    cout << "Task dependency graph created successfully" << endl;
    projectTasks.display();

} catch (const logic_error& e) {
    cerr << "Error: " << e.what() << endl;
}
```

Example 3: Weighted Route Planning

```cpp
WeightedGraph<string> cityNetwork(false);  // undirected

// Add routes with distances (km)
cityNetwork.addEdge("New York", "Boston", 346);
cityNetwork.addEdge("New York", "Philadelphia", 152);
cityNetwork.addEdge("Boston", "Philadelphia", 435);
cityNetwork.addEdge("Philadelphia", "Washington DC", 225);
cityNetwork.addEdge("New York", "Washington DC", 362);

cityNetwork.display();

cout << "Total cities: " << cityNetwork.getNumVertices() << endl;
cout << "Total routes: " << cityNetwork.getNumEdges() << endl;
```

Example 4: Bipartite Matching (Students-Courses)

```cpp
```

```
BipartiteGraph<string> enrollment;

try {
    // Students enrolling in courses
    enrollment.addEdge("Alice", "Data Structures");
    enrollment.addEdge("Alice", "Algorithms");
    enrollment.addEdge("Bob", "Data Structures");
    enrollment.addEdge("Bob", "Database Systems");
    enrollment.addEdge("Charlie", "Algorithms");
    enrollment.addEdge("Charlie", "Database Systems");

    // This maintains bipartite property
    enrollment.display();

    // This would fail (students connected to students):
    // enrollment.addEdge("Alice", "Bob");

} catch (const logic_error& e) {
    cerr << "Error: " << e.what() << endl;
}
```

Example 5: Complete Network (Full Mesh)

```cpp
CompleteGraph<char> meshNetwork;

// Add nodes - they automatically connect to all existing nodes
meshNetwork.addVertex('A');
meshNetwork.addVertex('B');
meshNetwork.addVertex('C');
meshNetwork.addVertex('D');

cout << "Complete graph K" << meshNetwork.getNumVertices() << endl;
cout << "Total edges: " << meshNetwork.getNumEdges() << endl;
// K4 has 4 vertices and 6 edges: 4*(4-1)/2 = 6

meshNetwork.display();
```

Best Practices

1. Choose the Right Graph Type

```cpp
```

```cpp
// ✅ Good: Use DirectedGraph for one-way relationships
DirectedGraph<string> twitter;
twitter.addEdge("Alice", "Bob");  // Alice follows Bob

// ❌ Bad: Using UndirectedGraph for asymmetric relationships
UndirectedGraph<string> twitter;  // Makes it mutual (incorrect!)
```

2. Use Appropriate Vertex Types

```cpp
// ✅ Good: Meaningful vertex types
DirectedGraph<string> emailNetwork;
emailNetwork.addEdge("alice@example.com", "bob@example.com");

WeightedGraph<int> ipNetwork(true);  // IP addresses as integers
ipNetwork.addEdge(192168001001, 192168001002, 100);  // latency in ms

// ✅ Good: Use structs for complex data
struct City {
    string name;
    int population;
    // Must implement comparison operators for use in set/map
    bool operator<(const City& other) const {
        return name < other.name;
    }
};
```

3. Handle Exceptions Properly

```cpp
try {
    DirectedAcyclicGraph<string> dag;
    dag.addEdge("A", "B");
    dag.addEdge("B", "C");
    dag.addEdge("C", "A");  // Would create cycle
} catch (const logic_error& e) {
    cerr << "Cannot add edge: " << e.what() << endl;
    // Handle error appropriately
}
```

4. Validate Graph Properties

```cpp
```

```cpp
// Check connectivity before processing
if (!graph.isConnected()) {
    cout << "Warning: Graph has disconnected components" << endl;
    // Handle accordingly
}

// Verify cycle detection for CyclicGraph
CyclicGraph<int> graph;
// ... add edges ...
if (graph.hasCycle()) {
    cout << "Cycle detected - may need special handling" << endl;
}
```

5. Memory Efficiency

```cpp
// ✅ Good: Use appropriate types for large graphs
DirectedGraph<int> largeGraph;  // Integers use less memory than strings

// ✅ Good: Reserve space if you know the size
// (Modify the class to support this if needed)

// ❌ Bad: Using strings unnecessarily
DirectedGraph<string> graph;
graph.addEdge("1", "2");  // Use int instead!
```

Common Patterns

Pattern 1: Graph Traversal

```cpp
```

```cpp
// BFS-style traversal (using isConnected as reference)
void traverseGraph(const Graph<int>& graph) {
    set<int> visited;
    queue<int> q;

    auto vertices = graph.getVertices();
    if (vertices.empty()) return;

    int start = *vertices.begin();
    q.push(start);
    visited.insert(start);

    while (!q.empty()) {
        int current = q.front();
        q.pop();

        cout << "Visiting: " << current << endl;

        // Process neighbors (access via graph methods)
    }
}
```

Pattern 2: Path Validation

cpp

```cpp
// Check if path exists between two vertices
bool pathExists(const ConnectedGraph<string>& graph,
                const string& src, const string& dest) {
    auto vertices = graph.getVertices();

    // If both vertices exist and graph is connected, path exists
    return vertices.count(src) > 0 &&
           vertices.count(dest) > 0 &&
           graph.isConnected();
}
```

Pattern 3: Cycle Detection Pattern

cpp

```cpp
void analyzeCycles(const CyclicGraph<int>& graph) {
    if (graph.hasCycle()) {
        cout << "Graph contains cycles - unsuitable for topological sort" << endl;
        cout << "Consider using DAG for hierarchical structures" << endl;
    } else {
        cout << "Graph is acyclic - safe for dependency resolution" << endl;
    }
}
```

Pattern 4: Bipartite Validation

cpp

```cpp
void validateMatching(BipartiteGraph<string>& matching) {
    try {
        // Add edges
        matching.addEdge("Student1", "Course1");
        matching.addEdge("Student1", "Course2");

        // Bipartiteness is automatically validated
        cout << "Valid bipartite matching" << endl;

    } catch (const logic_error& e) {
        cout << "Invalid matching: " << e.what() << endl;
    }
}
```

Performance Considerations

Time Complexity

Operation	Time Complexity	Notes
`addVertex()`	O(1) average	Hash map insertion
`addEdge()`	O(1) average	Plus validation for special types
`getNumVertices()`	O(1)	Direct count
`getNumEdges()`	O(V)	Iterates through adjacency list
`isConnected()`	O(V + E)	BFS traversal
`hasCycle()`	O(V + E)	DFS traversal
`display()`	O(V + E)	Prints all edges

V = Number of Vertices, **E** = Number of Edges

Space Complexity

- **Adjacency List**: O(V + E)
- Better than adjacency matrix for sparse graphs (E << V²)

Optimization Tips

cpp

```
// ✅ Good: Add edges in batch when possible
WeightedGraph<int> graph(true);
graph.addEdge(1, 2, 10);
graph.addEdge(2, 3, 20);
graph.addEdge(3, 4, 30);
// All at once avoids repeated validation

// ✅ Good: Use smaller types when possible
DirectedGraph<short> smallGraph;  // If vertex IDs are small

// ✅ Good: Check connectivity once, cache result
bool connected = graph.isConnected();
// Use 'connected' multiple times instead of calling isConnected() repeatedly
```

Error Handling

Common Exceptions

cpp

```cpp
// 1. Adding edges to Null/Trivial graphs
try {
    NullGraph<int> ng;
    ng.addVertex(1);
    ng.addEdge(1, 2);  // Throws logic_error
} catch (const logic_error& e) {
    // Handle: "Cannot add edges to a Null Graph"
}

// 2. Creating cycles in DAG
try {
    DirectedAcyclicGraph<int> dag;
    dag.addEdge(1, 2);
    dag.addEdge(2, 3);
    dag.addEdge(3, 1);  // Throws logic_error
} catch (const logic_error& e) {
    // Handle: "Adding this edge would create a cycle in DAG"
}

// 3. Breaking bipartite property
try {
    BipartiteGraph<int> bg;
    bg.addEdge(1, 2);
    bg.addEdge(2, 3);
    bg.addEdge(3, 1);  // Throws logic_error (odd cycle)
} catch (const logic_error& e) {
    // Handle: "Adding this edge would break bipartite property"
}
```

Testing Your Implementation

```
cpp
```

```
void testGraphs() {
    // Test 1: Basic connectivity
    UndirectedGraph<int> test1;
    test1.addEdge(1, 2);
    test1.addEdge(2, 3);
    assert(test1.isConnected() == true);

    // Test 2: Disconnected components
    DisconnectedGraph<int> test2;
    test2.addEdge(1, 2);
    test2.addEdge(3, 4);
    assert(test2.isConnected() == false);

    // Test 3: Cycle detection
    CyclicGraph<int> test3;
    test3.addEdge(1, 2);
    test3.addEdge(2, 3);
    test3.addEdge(3, 1);
    assert(test3.hasCycle() == true);

    cout << "All tests passed!" << endl;
}
```

Contributing

When extending this implementation:

1. **Maintain template support** for all graph types
2. **Follow naming conventions** (camelCase for methods)
3. **Add comprehensive docstrings** for new methods
4. **Validate input** and throw appropriate exceptions
5. **Update this USAGE.md** with new features

License

This implementation is provided as-is for educational and commercial use.

Support

For issues, questions, or contributions:

- Open an issue on the repository
- Refer to the code comments for implementation details
- Check the examples above for common use cases

Happy Graph Building! 🚀