Operations system -Final project

2024, Semester B

Submit: 322530080_ 207199282

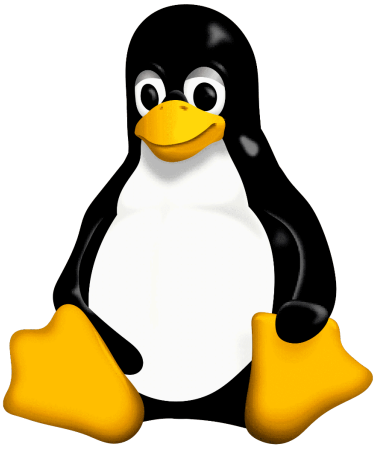


Table of Contents

1. Introduction

This document provides an in-depth explanation of the server-client application project developed in C++. The project centers around graph operations and computations of Minimum Spanning Trees (MST) using Kruskal's and Prim's algorithms. The application is designed to handle multiple client connections concurrently and perform computationally intensive tasks efficiently. This is achieved through the implementation of several design patterns, including the Active Object pattern, Thread Pool pattern, and Factory pattern, as well as careful concurrency management using mutexes and synchronization mechanisms.

2. Project Overview

The project's primary goal is to create a robust server that can handle multiple clients simultaneously, allowing them to perform various operations on graphs. Clients can:

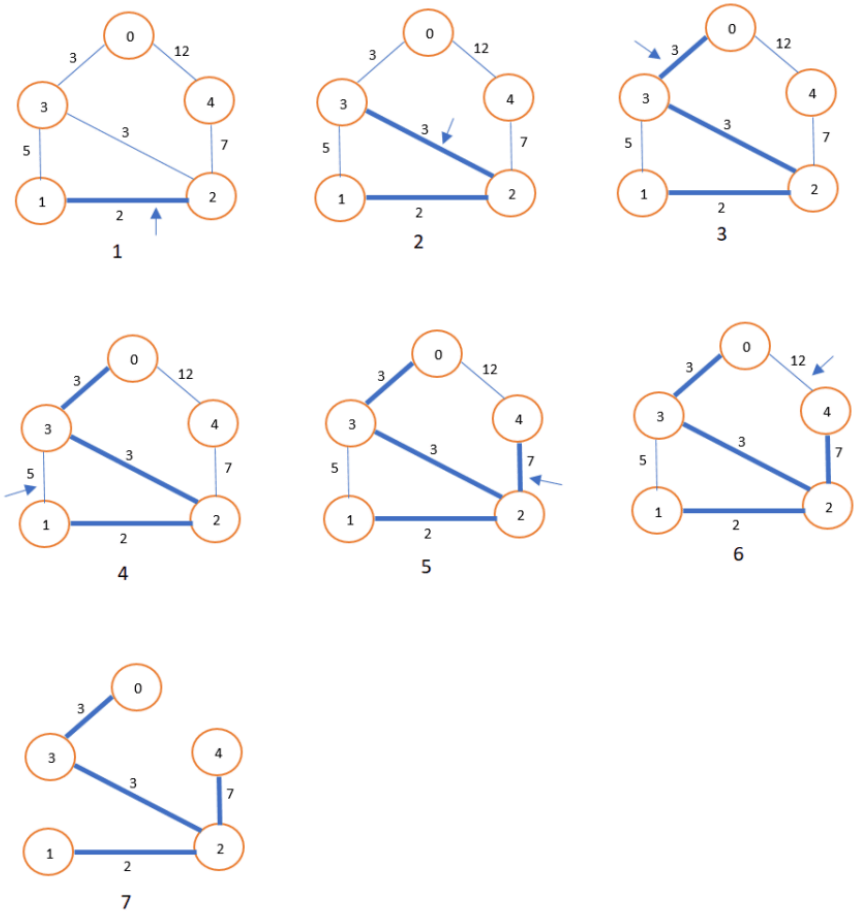
- **Select an MST Algorithm:** Choose between Kruskal's or Prim's algorithm for MST computation.
- **Create and Manipulate Graphs:** Input custom graphs by specifying vertices and edges, and modify existing graphs by adding or removing edges.
- **Perform Graph Operations:** Calculate the total weight of the MST, determine the longest or shortest paths between vertices, and compute average distances within the graph.

To achieve high performance and responsiveness, the server employs a combination of multithreading and design patterns that effectively manage concurrent tasks and resource utilization. The Active Object pattern allows the server to offload computational tasks to a dedicated thread, while the Thread Pool pattern manages multiple client-handling threads efficiently. The Factory pattern is used to create instances of MST computation algorithms based on client preferences.

3. Minimum Spanning Trees (MST)

An MST is a subset of edges in a connected, undirected graph that connects all the vertices with the minimum possible total edge weight, without forming any cycles. MSTs are fundamental in optimizing network designs, such as minimizing the length of cable needed to connect a set of nodes.

- **Kruskal's Algorithm**



Overview

Kruskal's algorithm is a greedy algorithm that constructs an MST by selecting edges in order of increasing weight, ensuring that no cycles are formed in the process. It is particularly effective for sparse graphs.

Process

1. **Edge Sorting:** All edges of the graph are sorted in non-decreasing order based on their weights.
2. **Initialization:** Each vertex is treated as a separate tree or set (using disjoint-set data structures).
3. **Edge Selection:** Iterate through the sorted edges and, for each edge, determine if adding it to the MST would form a cycle:
    - **If no cycle is formed:** Add the edge to the MST and merge the two sets containing the vertices connected by the edge.
    - **If a cycle is formed:** Skip the edge.
4. **Completion:** Repeat the process until all vertices are connected, resulting in an MST.

Advantages

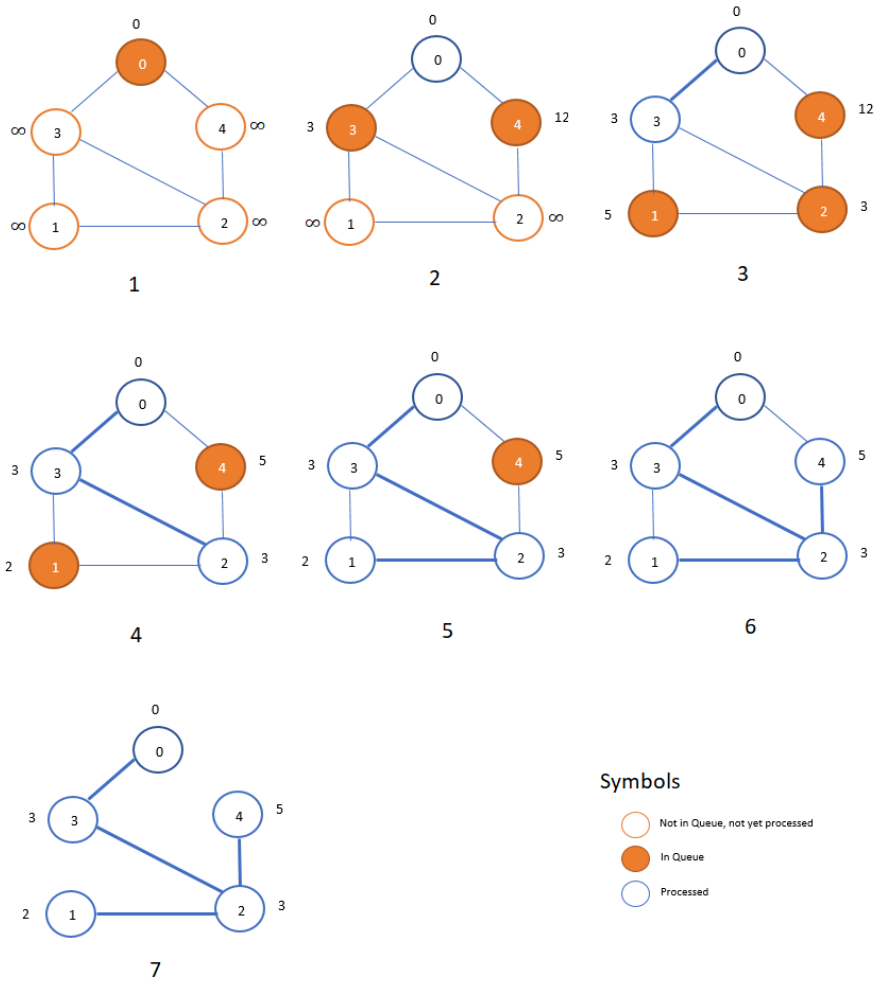
- **Efficiency with Sparse Graphs:** Kruskal's algorithm performs well with graphs that have fewer edges.
- **Simplicity:** The algorithm is straightforward to implement and understand.

Usage in the Project

Clients can select Kruskal's algorithm for MST computation. The server implements the algorithm by:

- Sorting the edges received from the client.
- Using disjoint-set data structures to manage vertex sets.
- Ensuring that the MST constructed is cycle-free and connects all vertices.

- **Prim's Algorithm**



Overview

Prim's algorithm is another greedy algorithm that builds an MST by starting from an arbitrary vertex and continuously adding the smallest edge that connects a vertex in the MST to a vertex outside of it. It is particularly efficient for dense graphs.

Process

1. **Initialization:** Start with an arbitrary vertex and mark it as part of the MST.
2. **Edge Selection:** At each step, select the smallest edge that connects a vertex in the MST to a vertex outside of it.
3. **Updating MST:** Add the selected edge and the new vertex to the MST.
4. **Completion:** Repeat the process until all vertices are included in the MST.

Advantages

- **Efficiency with Dense Graphs:** Prim's algorithm is efficient when dealing with graphs that have many edges.
- **Incremental Growth:** The MST grows one edge at a time, always maintaining a connected tree.

Usage in the Project

When a client selects Prim's algorithm, the server:

- Starts from a specified or arbitrary vertex.
- Utilizes a priority queue to select the minimum weight edges efficiently.
- Ensures that the MST remains connected and includes all vertices by the end of the process.

4. Project Files and Their Roles

The project is organized into several files, each serving a specific purpose:

- **`Server.cpp`:** The main server application that initializes the server, listens for client connections, and handles client requests. It integrates the thread pool and active object to manage concurrency.
- **`Graph.h` / `Graph.cpp`:** Defines the `Graph` class, representing the graph structure. Key responsibilities include:
    - Managing vertices and edges.
    - Providing methods to create new graphs, add edges, remove edges, and retrieve graph properties.
    - Ensuring thread safety when the graph is accessed or modified concurrently.
- **`KruskalMST.h` / `KruskalMST.cpp`:** Implements Kruskal's algorithm. Key functionalities:
    - Sorting edges based on weight.
    - Managing disjoint sets to detect cycles.
    - Constructing the MST and providing methods to retrieve MST properties, such as total weight.
- **`PrimMST.h` / `PrimMST.cpp`:** Implements Prim's algorithm. Key functionalities:
    - Utilizing a priority queue (often implemented as a min-heap) to select the next minimum edge.
    - Maintaining a set of visited vertices to prevent cycles.
    - Constructing the MST incrementally and providing methods to retrieve its properties.
- **`MSTFactory.h` / `MSTFactory.cpp`:** Implements the Factory design pattern to create instances of MST computation algorithms based on the client's choice. It abstracts the creation logic and provides a unified interface for the server to obtain MST solver instances.
- **`ActiveObject.h` / `ActiveObject.cpp`:** Implements the Active Object pattern. Key responsibilities:
    - Managing a task queue where computational tasks are submitted.

    - Running a dedicated worker thread that processes tasks asynchronously.
    - Ensuring thread-safe access to the task queue using mutexes and condition variables.
- **`ThreadPool.h` / `ThreadPool.cpp`:** Manages a pool of worker threads that handle client connections concurrently. Key features:
    - Maintaining a fixed number of threads to avoid the overhead of frequent thread creation and destruction.
    - Providing methods to enqueue tasks (client-handling functions) and manage task execution.
- **Mutexes and Synchronization Mechanisms:** Used throughout the project to ensure thread safety, particularly when accessing shared resources like the graph and task queues.

5. Libraries Used and Their Purposes

The project leverages both standard C++ libraries and system-specific libraries to facilitate networking, multithreading, and data manipulation.

Standard Libraries

- **`<iostream>`, `<string>`, `<sstream>`:**
  - Used for input/output operations.
  - String manipulation and formatting.
  - Parsing client input.
- **`<vector>`, `<list>`, `<queue>`, `<unordered_map>`:**
  - Data structures for storing graph vertices, edges, and task queues.
  - Efficient data retrieval and manipulation.
- **`<thread>`, `<mutex>`, `<condition_variable>`:**
  - Support for multithreading.
  - Synchronization primitives to manage access to shared resources and coordinate thread execution.
- **`<functional>`:**
  - Allows the use of `std::function` to encapsulate tasks and pass them around as objects.
- **`<algorithm>`:**
  - Provides algorithms like sorting, which is essential for Kruskal's algorithm.
  - Utilities for transforming and manipulating data structures.
- **`<chrono>`:**
  - Used for time-based operations, such as implementing delays or measuring execution time.

System Libraries

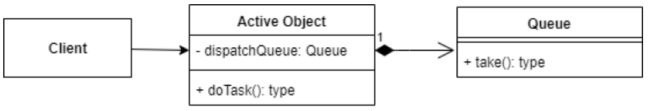
- **`<netinet/in.h>`, `<arpa/inet.h>`, `<unistd.h>`:**
  - Essential for socket programming and network communication.

    - Functions for creating sockets, binding, listening, accepting connections, and data transmission.
- **`<cstdlib>`, `<cstring>`:**
    - General utilities and string manipulation.
    - Memory allocation and deallocation.

6. Design Patterns Implemented

Design patterns are proven solutions to common software design problems. Implementing these patterns enhances code maintainability, scalability, and readability.

a. Active Object Pattern



Purpose

- Decouples method execution from method invocation.
- Enhances concurrency by allowing asynchronous execution of tasks.
- Simplifies synchronization by serializing access to shared resources.

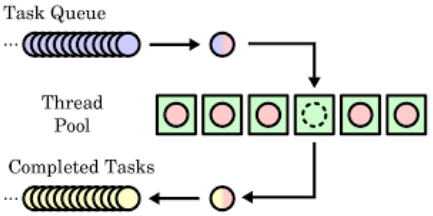
Implementation in the Project

- **Task Queue:** The `ActiveObject` class maintains a thread-safe queue of tasks (computational functions).
- **Worker Thread:** A dedicated thread continuously processes tasks from the queue.
- **Task Submission:** Client-handling threads submit tasks to the `ActiveObject` instead of executing them directly, ensuring they remain responsive.

Benefits

- **Responsiveness:** Client-handling threads are not blocked by long-running computations.
- **Concurrency Management:** Simplifies the coordination between threads and shared resources.
- **Scalability:** The server can handle more clients without a proportional increase in computational overhead.

b. Thread Pool Pattern



Purpose

- Manages a pool of reusable threads for executing tasks.
- Reduces the overhead associated with creating and destroying threads.
- Enhances performance in applications with a high volume of short-lived tasks.

Implementation in the Project

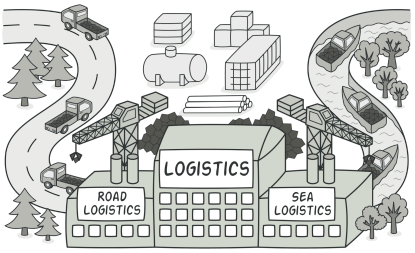
Fixed Number of Threads: The `ThreadPool` class initializes a set number of worker threads upon server startup.

- **Task Queue:** Tasks (client connections) are enqueued, and worker threads process them as they become available.
- **Thread Reuse:** Threads are reused for multiple tasks, minimizing resource consumption.

Benefits

- **Efficiency:** Reduces system resource usage by avoiding frequent thread creation.
- **Concurrency:** Allows multiple clients to be handled simultaneously.
- **Control:** The server can manage the number of concurrent threads, preventing overload.

c. Factory Design Pattern



Purpose

- Provides an interface for creating objects without specifying the exact class of the object that will be created.
- Promotes loose coupling by delegating the instantiation logic to subclasses or helper classes.

- Enhances flexibility and scalability by making it easy to introduce new types of objects.

Implementation in the Project

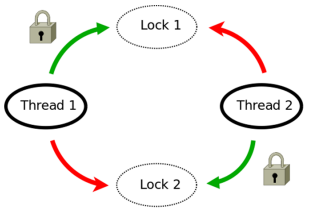
- **MSTFactory Class:** The `MSTFactory` serves as a factory for creating instances of MST computation algorithms.
- **Abstract Interface:** The factory provides a method (e.g., `createMST`) that returns an instance of an abstract `MSTSolver` interface.
- **Algorithm Selection:** Based on the client's choice (Kruskal or Prim), the factory instantiates the appropriate algorithm class without exposing the creation logic to the server.
- **Usage:**
    - **Server's Perspective:** The server requests an MST solver from the factory, specifying the desired algorithm.
    - **Factory's Role:** The factory decides which concrete class to instantiate (e.g., `KruskalMST` or `PrimMST`) and returns it as an `MSTSolver` interface.

Benefits

- **Encapsulation:** Creation logic is encapsulated within the factory, keeping the server code clean and focused on high-level logic.
- **Extensibility:** New MST algorithms can be added without modifying existing server code; only the factory needs to be updated.
- **Loose Coupling:** The server depends on abstractions (`MSTSolver` interface) rather than concrete implementations, promoting flexibility.

7. Concurrency Management with Mutexes

Concurrency introduces challenges such as data races and inconsistent states when multiple threads access shared resources simultaneously. The project addresses these challenges through careful concurrency management using mutexes and synchronization mechanisms.



Key Areas Utilizing Mutexes

- **Task Queues in Active Object and Thread Pool:**
  - **Mutexes:** Protect access to the task queues, ensuring that only one thread can modify the queue at a time.
  - **Condition Variables:** Used to notify worker threads when new tasks are available, allowing them to wait efficiently without busy-waiting.
- **Graph Access and Modification:**
  - **Graph Class Mutexes:** When the graph is accessed or modified (e.g., adding or removing edges), mutexes ensure that these operations are atomic and thread-safe.
  - **Synchronization:** Prevents inconsistent states or data corruption due to concurrent modifications.
- **Client Socket Communication:**
  - **Mutexes on Socket Operations:** To avoid race conditions when multiple threads might write to the same client socket, mutexes ensure that send and receive operations are performed safely.

Synchronization Mechanisms

- `std::mutex`**:** Provides mutual exclusion, ensuring that only one thread can enter a critical section at a time.
- `std::unique_lock`**:** Used with mutexes to provide exception-safe locking and unlocking.
- `std::condition_variable`**:** Allows threads to wait for certain conditions to be met before proceeding, facilitating efficient thread coordination.

Benefits of Proper Concurrency Management

- **Thread Safety:** Prevents data races and ensures the correctness of operations involving shared resources.
- **Performance:** Efficient synchronization mechanisms minimize the overhead associated with thread coordination.
- **Reliability:** Enhances the stability of the server by preventing unexpected behaviors due to concurrent access.

8. Usage of the Code

Server Operation Flow

1. **Initialization:**
    - The server starts and initializes the thread pool and active object.
    - It binds to a specified port and begins listening for incoming client connections.
2. **Accepting Connections:**
    - When a client attempts to connect, the server accepts the connection and enqueues a client-handling task to the thread pool.
3. **Client Handling:**
    - A worker thread from the thread pool picks up the client-handling task.
    - The thread manages the interaction with the client, including sending prompts and receiving inputs.
4. **Offloading Computational Tasks:**
    - For computationally intensive operations (e.g., MST computations), the client-handling thread offloads the task to the active object.
    - This allows the client-handling thread to remain responsive and handle additional client requests.
5. **Result Delivery:**
    - Once the active object completes a task, results are sent back to the client through the client-handling thread.
    - The server continues to interact with the client until the session ends.
6. **Connection Termination:**
    - When the client disconnects or the session ends, the server closes the connection and the worker thread becomes available for new tasks.

Client Interaction Flow

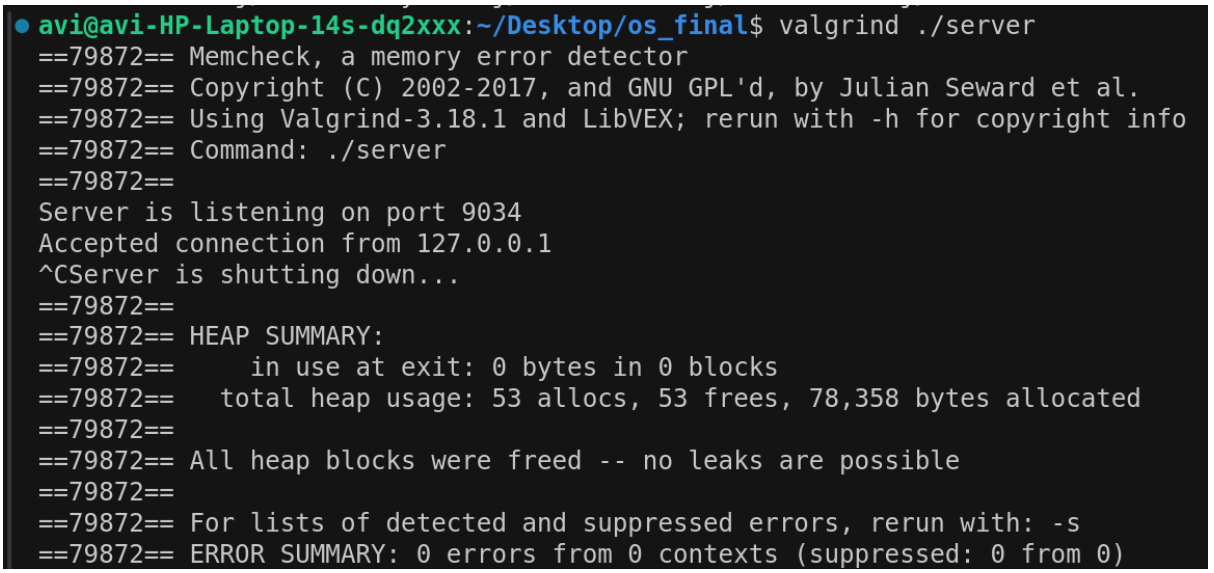
1. **Connection Establishment:**
    - The client connects to the server using a network socket.
2. **MST Algorithm Selection:**
    - The server prompts the client to choose between Kruskal's or Prim's algorithm.
    - The client sends their choice, which the server acknowledges.
3. **Graph Creation:**
    - The client specifies the number of vertices and edges.
    - The server prompts for edge details, and the client sends each edge in the specified format.
4. **MST Computation:**
    - The server automatically computes the MST using the selected algorithm.
    - The client is informed when the graph and MST are ready.
5. **Operations Menu:**
    - The server presents a menu of operations to the client.
    - The client selects an operation by entering its corresponding number.
6. **Performing Operations:**
    - Depending on the operation, the client may need to provide additional input (e.g., specifying vertices for shortest path).
    - The server processes the request, possibly offloading it to the active object.
7. **Receiving Results:**
    - The client receives the result of the operation.
    - The menu is presented again, allowing the client to perform more operations.
8. **Session Termination:**
    - The client can choose to create a new graph or disconnect.
    - Upon disconnection, the server cleans up resources associated with the client session.

9. Error Handling and Validation

- **Input Validation:**
  - The server validates client inputs to prevent invalid operations or data corruption.
  - Clients are notified of invalid inputs and prompted to try again.
- **Exception Handling:**
  - The server includes exception handling mechanisms to catch and handle unexpected errors gracefully.
  - Ensures that one client's error does not affect other clients or crash the server.
- **Resource Management:**
  - Proper cleanup of resources (e.g., closing sockets, releasing memory) to prevent leaks and ensure stability.

Valgrind checks:

- **Regular Valgrind:**



The Valgrind tests confirmed that there were no memory leaks or errors detected during execution.

- **Cg valgrind:**

```
avi@avi-HP-Laptop-14s-dq2xxx:~/Desktop/os_final$ valgrind --tool=cachegrind prog ./server
valgrind: prog: command not found
avi@avi-HP-Laptop-14s-dq2xxx:~/Desktop/os_final$ valgrind --tool=cachegrind ./server
==86410== Cachegrind, a cache and branch-prediction profiler
==86410== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==86410== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==86410== Command: ./server
==86410==
--86410-- warning: L3 cache found, using its data for the LL simulation.
Server is listening on port 9034
Accepted connection from 127.0.0.1
^CServer is shutting down...
==86410==
==86410== I   refs:      2,581,874
==86410== I1  misses:        5,476
==86410== LLi misses:        3,596
==86410== I1  miss rate:      0.21%
==86410== LLi miss rate:      0.14%
==86410==
==86410== D   refs:        871,479  (629,921 rd   + 241,558 wr)
==86410== D1  misses:       16,342  ( 13,539 rd   +   2,803 wr)
==86410== LLd misses:       10,079  (  8,096 rd   +   1,983 wr)
==86410== D1  miss rate:       1.9% (    2.1%     +     1.2%  )
==86410== LLd miss rate:       1.2% (    1.3%     +     0.8%  )
==86410==
==86410== LL refs:          21,818  ( 19,015 rd   +   2,803 wr)
==86410== LL misses:        13,675  ( 11,692 rd   +   1,983 wr)
==86410== LL miss rate:        0.4% (    0.4%     +     0.8%  )
```

The low miss rates suggest that the program is effectively utilizing the CPU cache, improving overall execution speed.

- **helgrind valgrind:**

```
avi@avi-HP-Laptop-14s-dq2xxx:~/Desktop/os_final$ valgrind --tool=helgrind ./server
==111800== Helgrind, a thread error detector
==111800== Copyright (C) 2007-2017, and GNU GPL'd, by OpenWorks LLP et al.
==111800== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==111800== Command: ./server
==111800==
Server is listening on port 9034
Accepted connection from 127.0.0.1
^CServer is shutting down...
==111800==
==111800== Use --history-level=approx or =none to gain increased speed, at
==111800== the cost of reduced accuracy of conflicting-access information
==111800== For lists of detected and suppressed errors, rerun with: -s
==111800== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 106 from 21)
```

Code coverage:

```
avi@avi-HP-Laptop-14s-dq2xxx:~/Desktop/os_final$ gcov Server.cpp
File 'Server.cpp'
Lines executed:91.60% of 262
Creating 'Server.cpp.gcov'
```

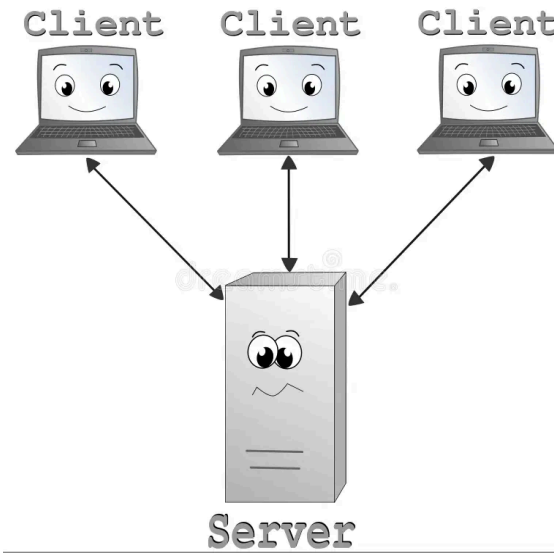
The `.gcov` files indicate that 91.6% of the code was covered during the tests executed with `gcov`. The lines that were not executed primarily pertain to exception handling cases.

10. Conclusion

This project demonstrates a comprehensive application of advanced programming concepts, including concurrency, networking, and algorithm implementation.

By integrating design patterns such as the Active Object, Thread Pool, and Factory patterns, the server achieves high levels of performance, scalability, and maintainability.

The use of mutexes and synchronisation mechanisms ensures thread safety and reliable operation in a multithreaded environment. Clients are provided with a robust interface to perform complex graph operations without experiencing delays or unresponsiveness.



OS project - MST, Strategy/Factory, Client-Server, Threads, Active Object, Thread poll (Leader-Follower) and Valgrind

Option 1: Pre-defined project

This project deals with the Minimal Spanning Tree (MST) problem on weighted - directed - graph.
You will implement a Factory for MST strategy.
We are also interested in the following data about the MST

- Total weight of the MST
- Longest distance between two vertices
- Average distance between and two edges n the graph.
  - assume distance (x,x)=0 for any X
  - We are interested in avg of all distances Xi,Xj where i=1..n j≥i.
- Shortest distance between two vertices Xi,Xj where i≠j and edge belongs to MST
- We will calculate in LF and in pipeline.

1. Implement a graph data structure - same as ex3.
2. Implement a data structure with a tree on the graph, as in section 1
3. Implement a Factory for MST algorithm that implements two algorithm solvers of
   a. Borůvka
   b. Prim
   c. Kruskal
   d. Tarjan
      You may examine MST implementation
      (Working) C++ Implementation of the Karger-Klein-Tarjan Algorithm for finding MST in expected linear time
   e. Integer MST
      https://www.sciencedirect.com/science/article/pii/S0022000005800649?via%3Dihub
4. We implement a server that Gets MST requests
   a. The server can get graphs, changes, and solve requests (same as ex 3) - Replace kosaraju command with the corresponding algorithm request.
5. The server will solve the problem and provide all the measurements above
   a. Using pipeline pattern (requires implementation of active object)
   b. Using Leader-Follower Thread poll
6. Provide Valgrind analysis (memcheck, helgrind, cg)
7. Prove code coverage of all code

Running commands:

To run the program:

Make all

./server

To Connect with a client: telnet 127.0.0.1