Data Structure for Graph

The graph was stored as an adjacency matrix using a 2D array of ints. With 100 nodes, this required 100 * 100 * 4 bytes = 40,000 bytes = 40 KB of RAM to store the entire graph. An adjacency matrix was chosen because it allows fast lookup of edges between nodes with O(1) time complexity. This speed was needed for the frequent graph queries by the threads.

Additional Data Structures

In addition to the graph adjacency matrix, I used the following extra data structures:

Path arrays: Integer arrays were used to store the computed paths between nodes. This allowed paths to be passed between threads efficiently.

Thread data struct: A simple struct was used to pass thread ids into the thread functions.

Mutex locks: Mutex locks protected access to shared data structures like the graph and path counter to prevent race conditions.

Condition variables: Condition variables allowed threads to notify each other when new path data was available. This reduced idle waiting.

Locks and Synchronization

The key locks used were:

A mutex lock around the graph data structure to prevent concurrent access and corruption. This ensured graph queries and updates were atomic.

A mutex lock around the global path counter to prevent race conditions as multiple threads incremented it.

Mutex locks when accessing any shared data structure like the path arrays or log files.

The locks allowed threads to operate in parallel on independent data while synchronizing access to shared state. Condition variables reduced idle waiting for new data.

Debugging and Errors

The main errors encountered were due to race conditions on shared data. These would result in crashes or invalid graph state. Enabling mutex locks around all shared accesses fixed these issues.

I also encountered performance issues due to too much locking. I optimized this by only locking shared resources when necessary and using condition variables.

Careful modular testing of each thread function also helped isolate bugs. Validating the logs and graph output ensured correctness.

Overall, the multi-threaded design was complex but the use of synchronization primitives like mutexes prevented race conditions. Modular testing and validation were key to debugging the system.