HW 4 Report: Rose Whitt

a) Summary

I followed the general Java Lock dining philosophers algorithm. I added a parameter to my compute method that took in the number of threads available and did a for loop nthread times, where each for loop was a new thread that would be added to my array of threads. I also added a reentrant lock to the ParComponent class.

For each thread, while the current node (the first node polled from nodesLoaded) wasn't null, I attempted to contract the graph. I first did a trylock on the node to make sure another thread wasn't already working on this node, and if it was I continued to the next node. If this passed, I checked that the node was not already being processed by checking if it was dead and if it was, I unlocked the node and continued to the next node. If this condition passed as well, I could then safely retrieve the current node's minimum-cost edge. If the minimum-cost edge was null, however, the graph has been successfully contracted, meaning the current node is the contracted graph. Therefore, I set the result variable to be the current node and broke out of the for loop entirely. Now, I can safely get the neighbor node (the node attached to that minimum-cost edge). Since the two nodes can't merge if the neighbor node is locked, we call trylock on the neighbor node. If it cannot be unlocked, I release the current node's lock, add the current node to the work list (nodesLoaded) and continue to the next iteration (next node). If the neighbor can be safely unlocked, I check if the neighbor node has already been processed by checking if it is dead. If it is dead, I unlock both nodes, add the current node to the work list since it has already been processed, and continue to the next node. If the nodes make it through all of this, they can now be safely merged. I set the neighbor node to be dead to indicate that it has been processed, then merge the neighbor and current node. I then unlock both nodes and add the current node to the work list. After the while loop but before the end of the for loop, I start the current thread.

After the for loop, I join each thread. Then, if the result is not null (i.e. an MST was found), I set the total edges and total weights to be that of the results.

b) Correct and Data-Race-Free

I know that my implementation is correct and data-race free because I pass all of the correctness tests and consistently get the same results. My parallelism only executes on disjoint pairs on a limited number of threads.

It's like dining philosophers but no deadlock or livelock bc we stop when a node has already been processed, unlike in dining philosophers where philosophers continue to eat.

c) No Deadlock

There is no deadlock for the above reasons and because a Java lock implementation of the dining philosophers problem does not cause a deadlock.

d) No Livelock

There is a livelock but it's rare enough to not affect performance.