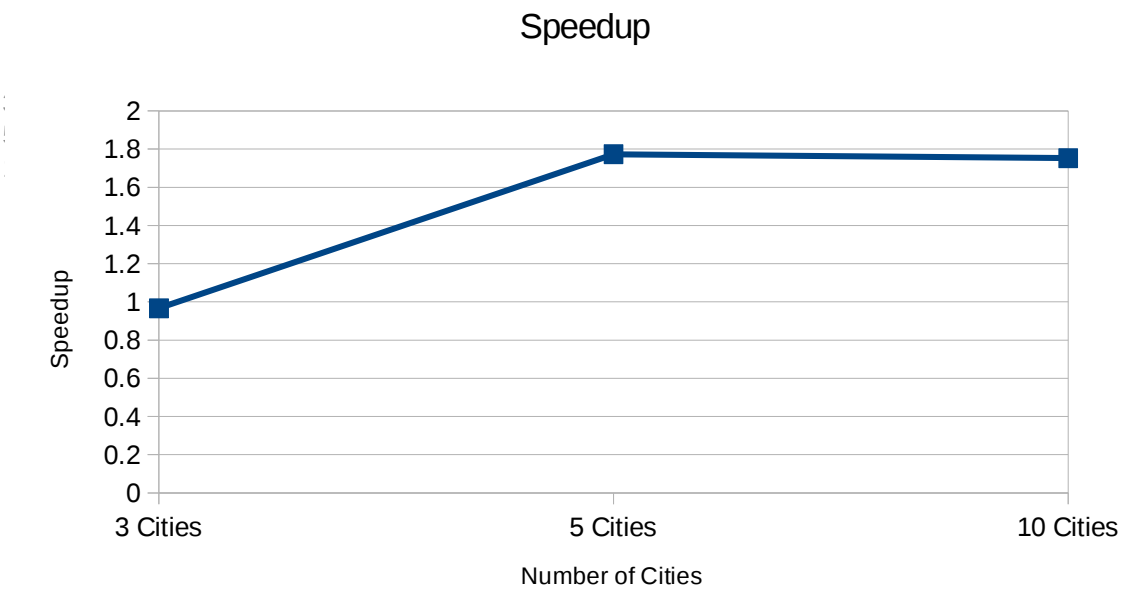
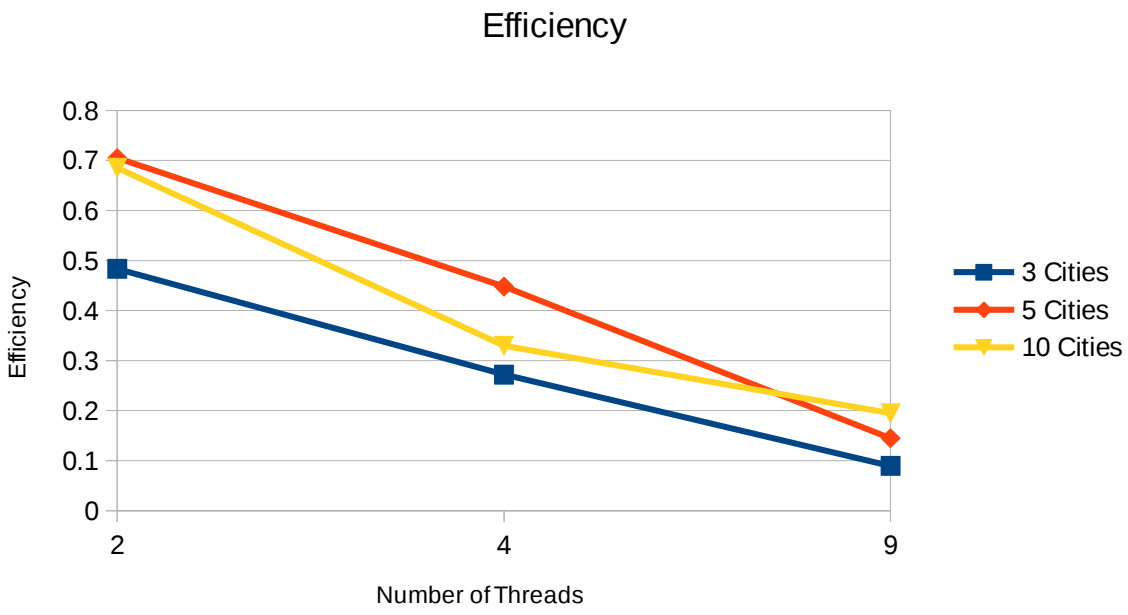
Lab 2 Submission

I parallelized my program by using a parallel for loop for the first level of recursion (I call this function ts_driver, which thereafter calls ts). My ts function uses recursion to enumerate all permutations of the cities and prunes branches of the tree whenever there is a node that contains a path with a distance larger than the min distance found so far.



The speedup chart reflects that we don't gain any performance with a small input of 3 cities. In fact, the speedup is less than one, probably because the overhead of forking processes outweighs whatever performance gain we may have received. With 5 and 10 cities, however, we see that the speedup is increased because the parallelized recursion makes the program run much faster. To calculate the speedup I used 2, 4, and 9 threads for the respective input files (this choice of thread numbers also yielded the shortest execution time).



To generate my efficiency graph, I temporarily modified my program to accept an additional command line argument to specify the number of threads to use. I ran each input file with 2, 4, and 9 threads. As we can see from the graph, the program is only weakly scalable at best. When we use 2 threads for the 3 cities input file, we get a similar efficiency calculation as when we increase the number of threads to 4 and increase the input size to 5 cities. This suggests that the efficiency is fixed when we increase the problem size at the same rate as we increase the number of threads. Unfortunately, this doesn't seem to hold up for 10 cities, as the efficiency seems to drop. I suspect that this is because the serial version of my program is very efficient and the parallel version doesn't make a huge improvement. We can easily conclude that the program is not strongly scalable because the lines of the graph all slope downward, meaning that when we increase the number of threads and keep the input size the same the efficiency goes down.