Paul Hudgins
CMSC 502
Assignment 1

## Implementation

This assignment was implemented in python3 using the mpi4py module. Instructions for performing a user install for this module on the class server are included in the project's README.

The project includes a single-threaded implementation of Held-Karp (dynamic programming solution to TSP), and multiprocess and MPI TSP solutions which both use a recursive bisection algorithm with Held-Karp being used for the smallest (n=10) subdivisions.

Trial runs were performed with cities generated by python code, but the submitted code reads cities from stdin. See README for usage. Due to the way mpiexec reads from stdin, more than 2000 cities cannot be input from stdin for tsp_mpi.py. For inputting larger sets, heldkarp.read_cities() could be modified to read from a file.

## Recursive Bisection Algorithm

At each step of the approximation algorithm, a set of sqrt(n) edge points is determined. Each step returns all optimum paths starting and ending at each combination of edgepoints, or sqrt(n)^2 = O(n) paths. If the number of cities is less than 10, a modified Held-Karp algorithm is run sqrt(n) times.

If the number of cities is more than 10, the set of cities is bisected along the x or y axis, alternating, and subsolutions are computed and stitched together at the closest points between the two subsets. This repeats recursively. This takes O(n) time for an O(n*log(n)) total time. Path length could be marginally decreased at the cost of time complexity in future implementations by comparing all likely stitching points for each top-level start and end points.

## Multiprocess and MPI Implementations

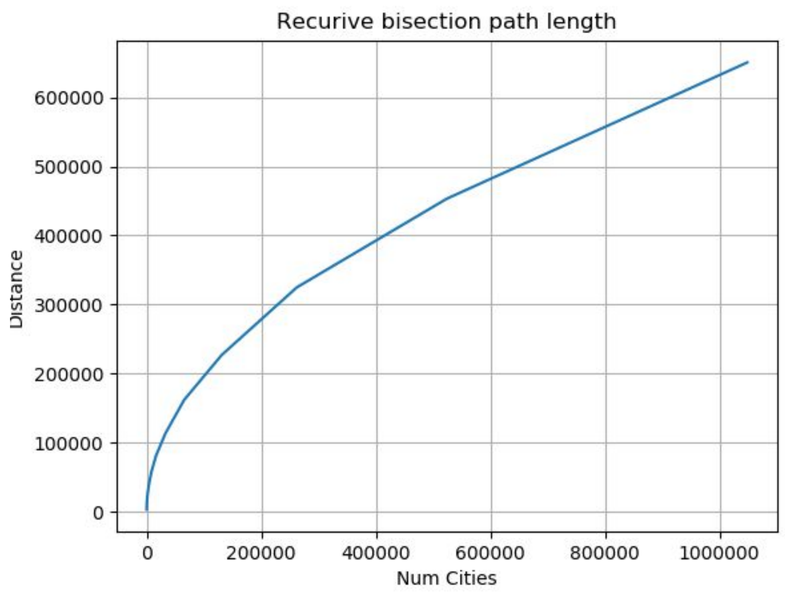
The multithreaded portion of this assignment was implemented with the python multiprocessing module, because the cPython interpreter uses a lock that prevents multicore speedups with threaded applications. Threads could be with python implementations not available on the server.

For the multiprocess implementation, each bisection forked a new process up to a specified depth. The number of processes is 2^depth.

For the MPI implementation, each bisection sends half of its workload to a process with rank = current_rank + 2^current_depth, resulting in an even division of workload that is analogous to the multiprocess implementation. For example, the rank=0 process will send half of its work to rank=1, then half of the remaining work to rank=2, and half of the remaining work to rank=4, and so on. The rank=1 process will send half of its work to rank=3, then half of the remaining to rank=7.

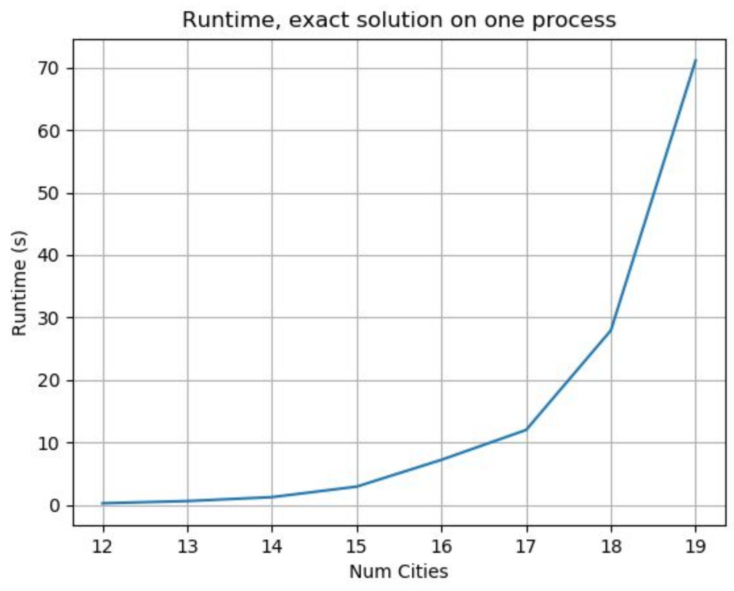
## Performance: Quality

For up to 20 cities, this method produced a 12-21% increase in path length compared to the optimum solution. This is expected to stay roughly constant for increased numbers of cities, because the ratio of optimal (Held-Karp) and non-optimal (stitching) connections should remain constant. The graph below shows path length produced by the recursive bisection algorithm for number of cities, given a 500x500 grid.

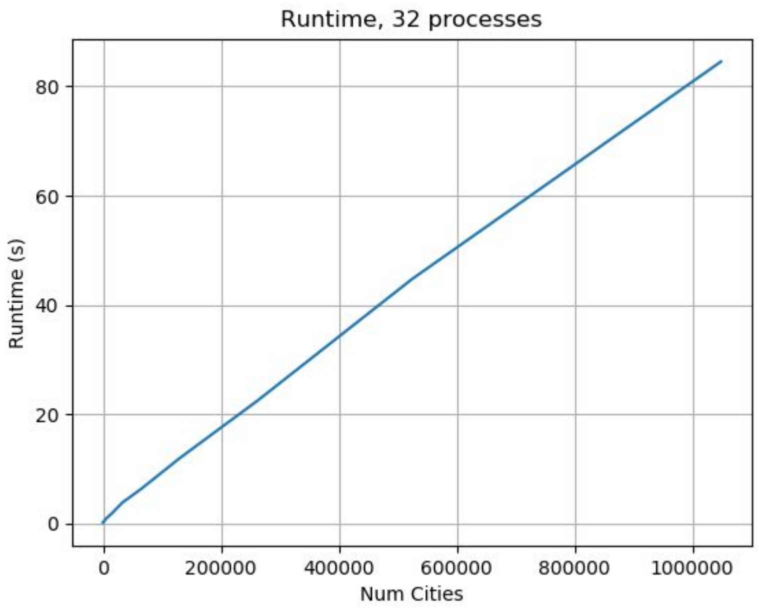


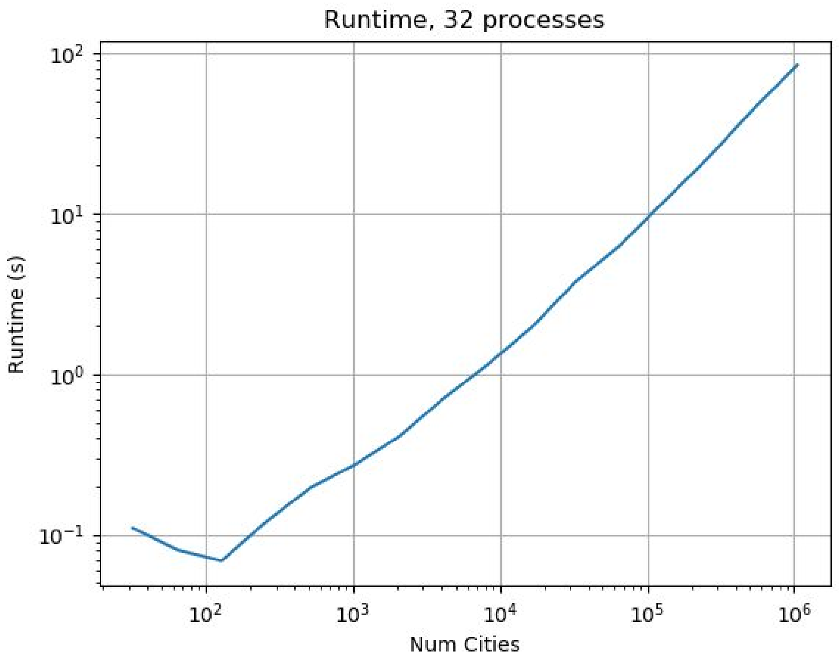
## Performance: Speed

As expected, the single-threaded exact solution takes exponential time.

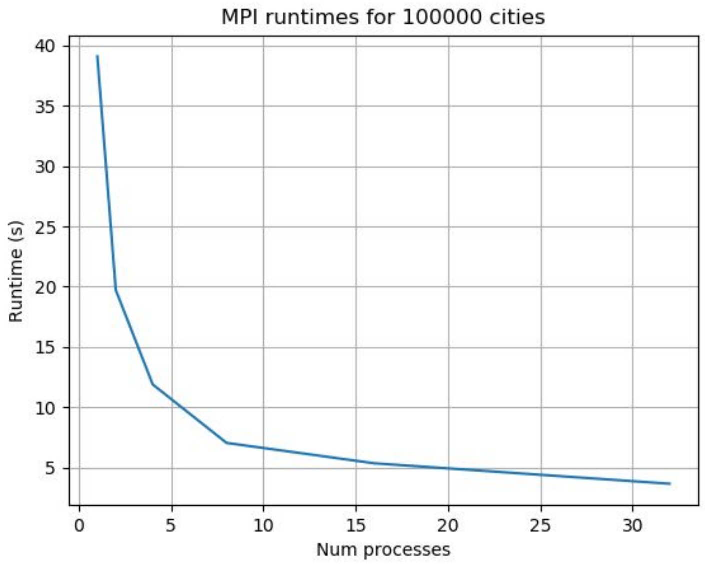


The multiprocess solution using recursive bisection scales very well to large number of cities, as shown below. Note that the total computational cost of the algorithm should be O(n*log(n)), but the cost of the final step which must be executed on a single core is O(n).





The MPI-based solution showed similar but marginally faster performance compared to the multiprocess solution. The large majority of required communication is in returning solutions to be merged. It is unclear how much of the runtime was taken by this communication, and it will need to be determined by further experiments. For 1,000,000 cities, the MPI method took 39 seconds on one core, 19 seconds on two cores, and 3.7 seconds on 32 cores. This shows diminishing returns, perhaps due to server limitations, with a 100% speedup with one extra core and a 980% speedup with 31 extra cores.


MPI runtimes for 100000 cities
Runtime (s)
Num processes