TRINITY COLLEGE DUBLIN, SCHOOL OF MATHEMATICS

Assignment #5, Module: MA5611

Gustavo Ramirez

March 28, 2017

1 PART 1

For the implementation of this part (which can be found in the file ./task1/tsp-serial.c), a simple brute force approach was used. This approach was based on recursivity, which gives all possible combinations of the n points. Once compiled, the program can be used in the following way: if called just with a -n flag, then the points are randomely generated, but if called with both -n and -f flags, then the program takes n points/lines from the input file.

2 PART 2

For plotting time data, gnuplot was used.
From the fitting of the execution times (see the ./task2/fit.log file), the following values are obtained:
$a = 1.39018 \cdot 10^{-11}$
$b = 2.59096$
$c = 0.965511$
for the following form of the exponential:

$$f(x) = a \cdot e^{x \cdot b} + c$$

and therefore:

$$f(x) = 1.39018 \cdot 10^{-11} \cdot e^{x \cdot 2.6} + 0.965511$$

from which the times for execution are, approximately (in seconds and days):

$$T(20) \sim 532579070245\ seconds \sim 16887.97\ years$$

$$T(50) \sim 4 \cdot 10^{45}\ seconds \sim 10^{38}\ years$$

$$T(20) \sim 114 \cdot 10^{100}\ seconds \sim 4 \cdot 10^{94}\ years$$

3 PART 3

As seen from the previous section, the execution times for the brute force method on the TSP problem go to "inifinity" (many years) very quickly. For this reason, an alternative approach is necessary.
The compilation and execution of this program (which can be found at ./task3/tsp_simm_annealing.c) are the same as for the program in Part 1.
To solve the TSP problem in a considerably small amount of time, Simulated Annealing was used.
Here is a description of the steps taken on that algorithm (and more generally, on the specific implementation taken on ./task3/tsp_simm_annealing.c):

1. set all initial points choosing an order (i.e. an initial solution)
2. randomly choose any two points and swap them
3. check if the new route is better than the previous one, and if not, give an extra chance to choose that route from a flat random distribution (this choosing mechanism is the same as in Metropolis algorithm)
4. go back to step 2 and do N times
5. do steps from 2 to 4 for each temperature, going from a high temperature and slowly decreasing

then, as can be seen, Simulated Annealing is like the simple Metropolis algorithm, but with variable temperature. There is no stop condition, but rather, an equal number of iterations is used on each run of the simulation (within one same program execution).
For the Metropolis step, the distance per tour is used instead of the Hamiltonian.
Important note: the implementation only works with files with the format of file at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/att532.tsp.

4 PART 4

To check the correct functionality of the program developed for *task 3*, the code written for *task 1* was used, generating data with the latter to check the former, for low numbers of cities.

When the problem grows, for example when there are 532 cities (as in the att532.tsp file), the solution is far from the best, e.g. for an initial temperature of 10^8, 50000 iterations and $\alpha = 0.99$, the best solution is 1441896.6, far from the best solution found in the TSPLIB website, i.e. 27686.
Although originally no stop criteria was used, an extra code was added to check after how many local iterations the algorithm converged (but after that, it was let run, just in case the perturbation step allowed to reach a better solution). The stop criteria can be turned off by deleting the last if statement in the simulated_annealing() function.

4.1 MPI/PARALLEL APPROACH

Both approaches are parallelizable: the one from task 1 (i.e. the brute force approach) and the one using Simulated Annealing algorithm. Because the first one takes forever to perform all the possible combinations, and because the second gives bad precision results, a parallel implementation could be very helpful.
For the first approach, the brute force one, a direct parallel implementation is probably not very useful, i.e. given n cities and having i processors, assign n/i cities to each processor and let each one of them to process all possible combinations for their n/i assigned cities. This direct parallelization approach only reduces the computation time by a factor of i, which doesn't help as the execution goes exponential on the number of cities.
A direct parallelization of Simulated Annealing wouldn't be a very good idea either, as Markov Chains are sequential by nature; would be better to choose an algorithm which is highly parallelizable by its nature.
If writing the parallel code from scratch, would be better to use an algorithm who adapts better to the possible topologies of MPI; then, the cities could be mapped to a virtual topologie (a graph topology, to be more specific), from which the problem can be subdivided and closer neighbours can be packed and assigned to each processor, reducing the problem to multiple small problems, but avoiding the interaction of different areas in the graph. The subdivision of tasks can be made by first analysing (roughly) the density of cities, and from there distributing cities over processors.
One last detail is important on the parallelization, and that's the separating borders of each region in the topologically separated total volume, because it might be that the best solution (smaller distance) is actually between two neighbours living in two different sub-regions.