HPC Software
Assignment 5
Travelling Salesman Problem (TSP)

Robert Cronin 12313002

March 26, 2017

The file `tsp.c` can be compiled for both open and closed Travelling Salesman tours. Adding the flag -DCLOSED will create the latter. Adding the flag -DPLOT will create a data plot that tracks the best distance found over time. See the `README` file for more details.

1 Brute Force

By adding the flag -b at runtime the code will attempt to calculate the best route by brute force. This means it checks every permutation and stores the minimum tour it finds.

As the number of cities grows however, this method becomes infeasible. Each added city adds a multiple of n possible permutations (where n is the number of cities). The time taken to find the best route therefore grows at an order of approximately $n!$. The data shown in Table 1 and the corresponding Figure 2 illustrates this[1].

Looking at the data we see it is a bit erratic for small numbers since it calculates it so quickly. Then it settles down after 11 cities and roughly takes n times as long as it did for n-1 cities, to find a solution for n cities. We can use this data to roughly estimate the amount of time it would take for x cities as

$$time(x) = x * time(x-1)$$

and so to extrapolate from our data this would be

$$time(x) = \frac{x!}{13!}442s$$

[1]These times use an open loop. Closed loops have less permutations and would therefore take less time for each n.

Cities	Time	Time Multiplier
1	0.000007	N/A
2	0.000003	0.428571
3	0.000008	2.666667
4	0.000015	1.875000
5	0.000043	2.866667
6	0.000168	3.906977
7	0.000985	5.863095
8	0.008297	8.423350
9	0.048105	5.797879
10	0.219334	4.559484
11	2.654830	12.104051
12	32.508106	12.244892
13	442.039480	13.597823

Figure 1: Time taken for TSP using Brute Force

We can then estimate:

- 20 cities would take $1.72 * 10^{11}s = 5475$ years
- 50 cities would take $2.15 * 10^{57}s = 6.84 * 10^{49}$ years
- 100 cities would take $6.62 * 10^{150}s = 2.1 * 10^{143}$ years

So clearly finding the optimal solution in this way is not possible for any reasonable sized number of cities.

2 Heuristic Solver

To overcome this, some heuristic solver is required. I chose to implement a combination of simulated annealing and 2-opt/Pairwise Exchange.

2.1 Simulated Annealing

Simulated Annealing works by proposing a new state s' at each iteration that is a neighbour of the current state s (our states are a permutation of the cities). The cost of the new state is then calculated (which in this case is the length of the tour).

A probability is then calculated based on the difference in the cost of s and s' and also on a variable known as the temperature. The probability is 1

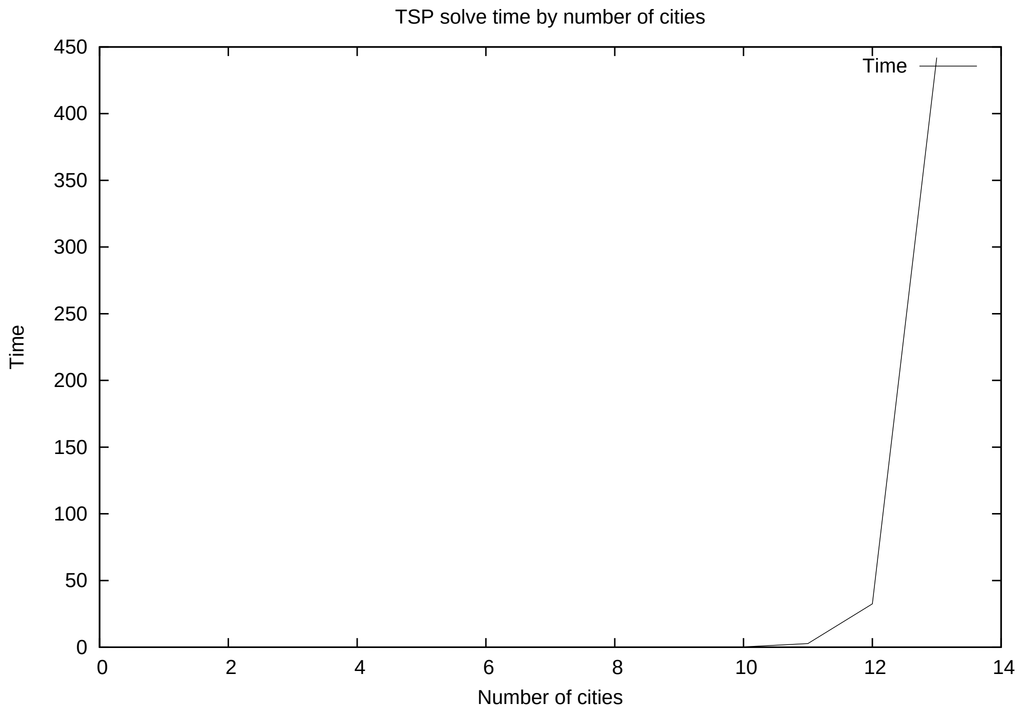

Figure 2: Time taken for TSP using Brute Force

if $cost(s') < cost(s)$. Otherwise it depends on the costs and the temperature. As the temperature decreases, so does the probability for these more costly states and typically the more costly s' is, the lower the probability.

The proposed state s' is then accepted as the new state with this probability or else remains at s. The reasoning for potentially accepting a worse state is to avoid getting stuck at a local minimum.

As we accept new states the temperature decreases so that over time we zone in on some solution. The rate at which the temperature decreases depends on some cooling factor typically of the form $T' = \beta T$. This β tends to work best for values very close to 1 such as $\beta = 0.9999$. Some methods implement β to be variable over time.

The method stops when either the temperature drops below a certain threshold or s' has been rejected a certain number of times in a row.

2.1.1 My Version of Simulated Annealing

I chose

$$\mathbb{P}(\text{accept } s'|s) = exp\left(\frac{cost(s) - cost(s')}{cost(s) * T}\right)$$

which satisfies the conditions mentioned before.

I initialised $T = 10$ with default cooling of $\beta = 0.9999$ applied everytime an outright better state is found. My temperature threshold was 0.00001 and I also finished the algorithm if there were $r * \sqrt{n}$ rejects in a row (r defaults to 1000).

Note: β and r can be modified at runtime with the '-c' (for cooling) and '-r' flags. The higher each value is, the better the result but the longer the runtime. I attempted to scale the reject limit as n grows by using a square root allowing larger datasets to take more time by default (but not directly linearly, since doubling r more than doubles the runtime).

2.2 Pairwise Exchange/2 opt

Simulated Annealing requires some method of proposing a new state s' from state s. The suggested method for TSP is to pick a random city along the tour and swap it with the next city to create your proposal s'. This results in a method that does seem to converge to somewhere close to the optimal solution.

However, I found that by implementing Pairwise Exchange it produced more accurate results in quicker time. This method involves again picking a random point but then picking another random point somewhere further along the tour and reversing the tour between those two points to create s'.

2.3 Greedy Algorithm/Nearest Neighbour

Another suggestion is to to initialise the tour to a state found using a greedy algorithm (nearest neighbour in this case) instead of starting from a random state. This involves finding the closest unvisited city at each point to create the initial path. However, I found that by starting with this, the algorithm was slightly less efficient, potentially due to getting stuck near a local minimum initially more easily. Therefore, by default, the algorithm runs starting from a random permutation.

Using the greedy method by itself can be run by using the flag '-m 3' or in order to initialise to this state when running simulated annealing use the '-g' flag.

3 Testing the Algorithm

Many of the datafiles found at http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html can be used to test the algorithm by using '-f [filename]'. Specifically any file which has either "NODE_COORD_SECTION", "DISPLAY_DATA_SECTION" or "EDGE_WEIGHT_SECTION" before the list of coordinates or distance matrix will work with `tsp.c`.

A few of these are provided in the `inputs/` directory. All the tours on the website provide an optimal value based on a closed loop so the closed version is used from here on out. The optimal values found so far from the website are provided in the `optimal.txt` file.

3.1 Symmetric TSP

A sample output using the berlin52 data set is shown in Figure 3 which was calculated in 3.77s and is within 0.03% of the optimal path (7544.37 vs 7542).

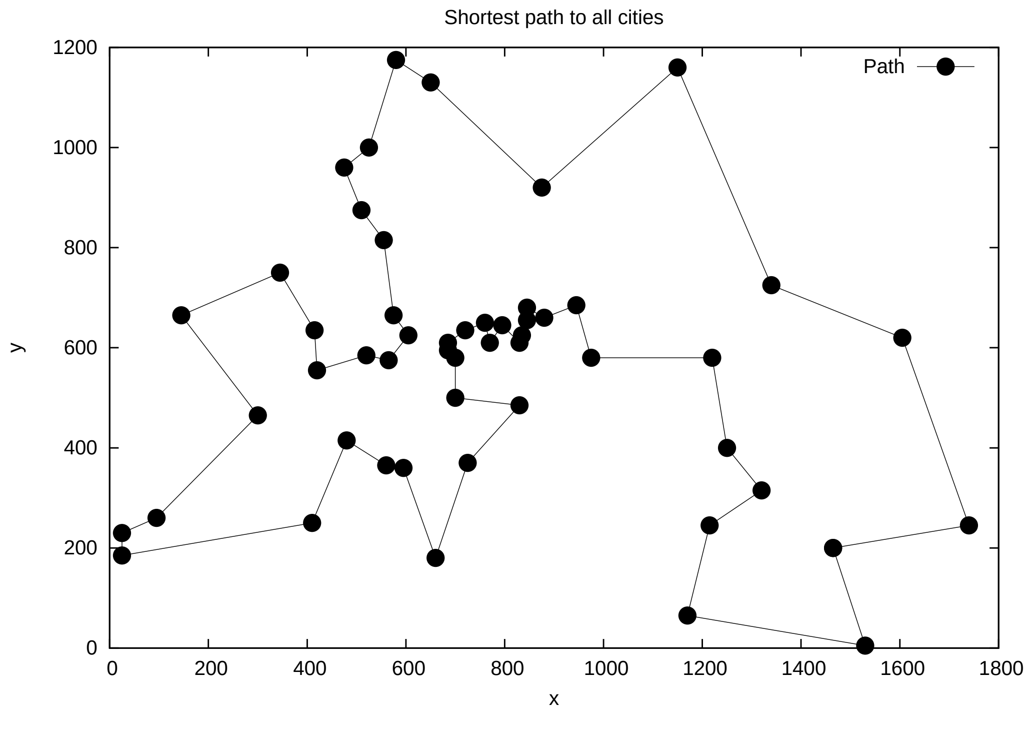

Figure 3: Berlin52 TSP solution

To see how well this method works I ran a number of test files and normalised the results based on the optimal to see how quickly they converge towards 1. The result of which can be seen in Figure 4. The number in the dataset name represents the number of cities.

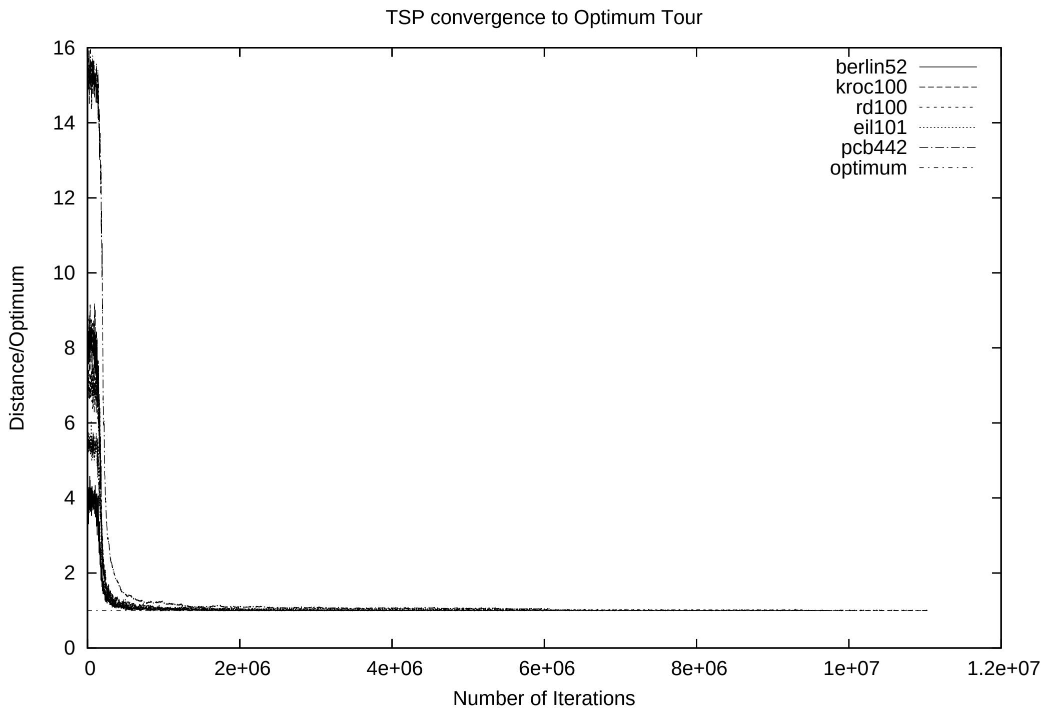

Figure 4: Convergence to solution of TSP

The errors for these files range from 0.03% for berlin52 to 4.29% for pb442, taking from 4.67s to 18.5s respectively[2]. It is to be expected that larger datasets would take longer and be less accurate as seen here but getting within 4% within 20s is pretty good for 442 data points when we consider the length of time to brute force 100 data points from before.

We see that all the data sets converge within 200,000 iterations and then only slightly fluctuate after that point. However after the 200,000 iterations the temperature has presumably gotten so low that we rarely experiment with a state that is more costly, and so after that point we tend to be stuck in some local minimum.

[2]The act of recording the data points slows down the code to some extent

3.2 Asymmetric TSP (ATSP)

We can also try with a few Asymmetric datasets like before as seen in Figure 5.

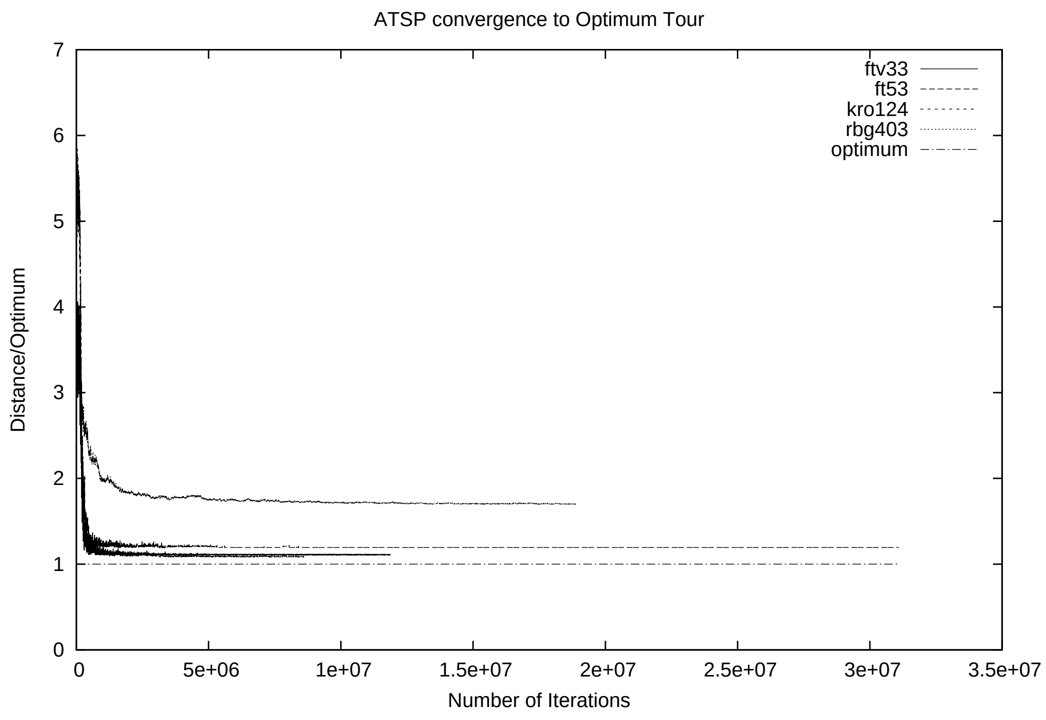

Figure 5: Convergence to solution of ATSP

We see immediately that the convergence is significantly worse for ATSP. The errors now range from 8.74% to 70.34% and the timings from 4.21s to 53.8s with both worse cases coming from the rbg403 dataset. They do still converge in roughly the same number of iterations but in all cases, the algorithm appears to get stuck at a local minimum that is far from the global minimum and can't escape from it. The asymmetric distances increase the number of permutations and hence the complexity of the problem and with it bring in more possibilities for getting stuck in local minimums.

4 Solving TSP in Parallel

4.1 Brute Force

One could implement the brute force algorithn in parallel such that if you were to find the permutations for n cities you could use n processes, where process i checks all permutations begining with city i. However this will only decrease the time by a multiple of most n, which is negligible for an $O(n!)$ algorithm.

4.2 Simulated Annealing

Instead we could try implement our serial algorithm in parallel.

We could use x processes to start from x random permutations which based on the results in Section 3 we expect will all converge to some local minimum. Then we could pick the best result from these. This is equivalent to running the same problem x times in serial, however it gives x opportunities of heading down the correct path to the global minimum rather than getting stuck at a relatively poor local min, like we saw for the ATSP.

The other extreme would be to propose x new states s'_i at each iteration and choose the best option at each step. This would be likely to converge faster although it would be far more likely to hit a local min as it will rarely accept a worse state which is the point of the simulated annealing process. The overhead of so many MPI_Send's would probably negate the convergence speed up as well.

Perhaps a combination of these two would work best. Each process starts at a random permutation and runs for a certain number of iterations. After this the solutions so far are pooled together and the best one is then distributed back to each process. The method would then repeat where each process loops for a number of steps, pools and redistributes.

The number of loops before pooling should be higher at the start to check many paths and avoid local mins and then the number should reduce as we hone in on, hopefully, something close to the global min.

This would help with both avoiding local mins and with speeding up convergence. Tweaking the number of loops and how it varies would be the biggest challenge here, by trying to find a good balance between these two issues.

4.3 Genetic Algorithms

Another method of course would be genetic algorithms, where the island model could be implemented as explored in a previous assignment.

5 References

- http://math.boisestate.edu/ wright/courses/m365/ExampleProject3.pdf
- http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html