Finding a Least-Cost Hamiltonian Cycle With Parallel Simulated Annealing

Jack Gindi
Scientific Computing
jg6848@nyu

Brady Edwards
Scientific Computing
bse4289@nyu

Abstract

The Traveling Salesman Problem (TSP) is one of the most famous problems in computer science, and it appears in a variety of different practical contexts across different industries. Despite this, after many years of effort on this problem, there is still no known efficient exact algorithm for computing the length of the shortest possible tour. In this paper, we implement a parallel approach to solving this problem based on a meta-algorithm known as simulated annealing (SA). SA uses a temperature parameter β to control how aggressively it explores the state space as the algorithm progresses. In this work, we measure the performance and scaling properties of our algorithm as we vary the number of processes and the problem size.

1 Background

Optimizing over a large, discrete solution space is generally a difficult problem. Solving such problems exactly more often than not requires super-polynomial computation time. One of the most famous such optimization problems is the Traveling Salesman Problem (TSP), and it can be states as follows. Let $G = (V, E)$ be a fully-connected graph on n vertices $V = \{1, 2, \ldots, n\}$. For each edge $(i, j) \in E$, we have an associated weight $w(i, j)$. If a tour t is a permutation of V, then we can define its cost, denoted by C, as

$$C(t) = w(t_n, t_1) + \sum_{i=1}^{n-1} w(t_i, t_{i+1}),$$

which is simply the total weight accumulated by traversing V in the order specified by t (ending where we started). Solving TSP can then be stated as the following optimization problem:

$$\operatorname*{argmin}_t C(t).$$

While there are many heuristics one can use to solve certain TSP instances, efficient, exact, and general algorithms remain elusive. Because the space of possible tours of a graph of size n is $n!$, the naive brute force algorithm does not scale beyond graphs with tens of nodes. For a visual example of what a TSP solver might do, see Figure 1.

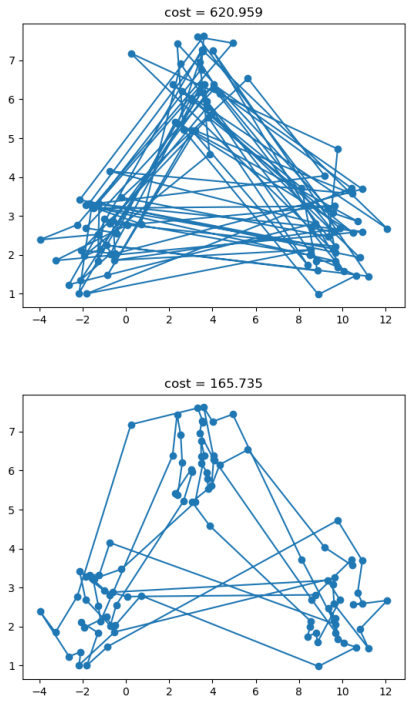

Figure 1: Two plots showing what a TSP solver would do. In this case, since the data has clusters, we would expect a low-cost tour to have many intra-cluster connections, with fewer inter-cluster connections. While the lower plot is not completely optimal, we can see that it is much better than the tour shown in the upper plot. (The costs of each tour are in the titles of the respective plots.)

The remainder of this paper is structured as follows: section 2 motivates and describes the simulated annealing algorithm, section 3 expounds the instantiation of simulated annealing for TSP, section 4 describes our use of parallelism to improve its performance, section 5 lays out our experiment

setup and results, section 6 discusses our hypotheses about dynamics we observe, and possible avenues for further investigation. (Section 8 lays out how the authors split up the work.)

2 Simulated annealing

Simulated annealing (SA) is an optimization algorithm often used for problems with large, discrete state spaces. Its name comes from a technique in metallurgy that uses temperature change to alter the physical properties of the metal as it is being shaped. When the metal is hot, it is easier to mold, but as it cools, the atoms settle into their final crystal structure.

The algorithm proceeds as follows. Let $s \in \mathcal{S}$ be the algorithm's state at time t, let $\mathcal{E}(s)$ denote the energy of s and $\Delta\mathcal{E}_{ss'}$ be the difference in energies between s and s'. (States with lower energy are preferred.) At each step of the algorithm, the probability of transitioning from s to $s' \in \mathcal{S}$ is given by

$$P(\mathcal{E}(s), \mathcal{E}(s'); \beta(t))$$

where $\beta(t)$ is an inverse temperature parameter that changes as the algorithm progresses. We then sample a state from $P(\mathcal{E}(s), \cdot; \beta(t))$ and repeat the process until a stopping condition is met.

Intuitively, when β is small, the algorithm is free to search the entire state space. This is analogous to the "hot" state of a metallurgical process. However, as $\beta \to \infty$ (lowering the temperature), the probability of choosing a state at a higher energy level than the current state goes to zero, and it can be shown that the only remaining options are in the global minimum almost surely. Key to this process, as in metallurgy, is how we raise the β parameter. If we raise it too quickly, we will get stuck in a local minimum; if we raise it too slowly then we will be stuck searching the entire state space making little more progress than brute force.

3 SA applied to TSP

To apply SA to solve TSP, we use the following setup:

- Our states are permutations of indices $\{1, \ldots, n\}$, where n is the size of the input graph. Each index corresponds to a point in 2D space.
- From a given state s, a transition consists of swapping a pair of indices in s.
- The probability distribution combines a Boltzman distribution and a uniform distribution:

$$P(\mathcal{E}(s), \mathcal{E}(s'); \beta(t)) = \begin{cases} \frac{e^{-\beta(t)\Delta\mathcal{E}}}{n^2} & \Delta\mathcal{E} > 0 \\ \frac{1}{n^2} & \Delta\mathcal{E} \leq 0 \end{cases}$$

- The energy (objective) value of a state s is the total Euclidean tour length incurred by visiting the states in the order s (and one additional step back to the beginning).

At each step, we use the aforementioned distribution to sample the next state.

4 Parallelization

To parallelize our implementation, given a starting state s, we used OpenMPI to initialize p nodes in order to sample p independent trajectories of length k through the state space. From among the p resulting states, we choose our next state s' to be the one with the smallest objective value. We then repeat the process starting from s' until we satisfy a stopping criterion of not improving the objective for x iterations. The reduction of the β parameter is carried out over the entire algorithm, and the value is the same across all processes, even as they explore different trajectories.

Intuitively, parallelization allows the algorithm to explore a larger portion of the state space more quickly, which should lead to convergence in a smaller number of iterations.

5 Experiment Results

In this section, we discuss two sets of experiments we ran to better understand the impact of different problem sizes and compute resources on the time-based and objective-based performance of our implementation.

5.1 How does adding processors affect performance?

The first aspect of our implementation that we want to test is: How much does performance improve by adding more processes for a fixed problem size? In other words, we are testing our algorithm's strong scaling properties.

To test this, we fix $n = 400$, and let p range over the values 1, 2, 4, 8, 16, 32. The number of annealing steps per call k was set to 500, and the maximum allowed annealing steps was set to 1,000,000 (i.e., we allow a maximum of 2,000 annealing calls). The patience of the algorithm is set

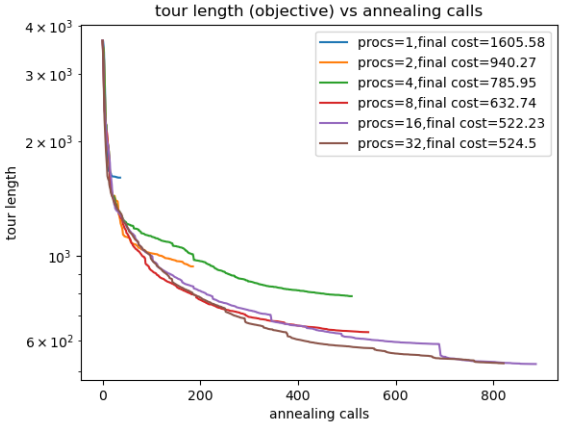

Figure 2: For different numbers of processors, we show the trajectory of objective values (tour lengths) over the course of a run. The cost at which the algorithm terminated is shown in the legend.

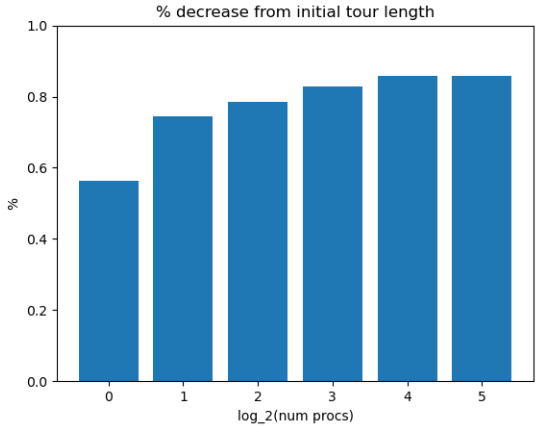

Figure 3: The percent decrease from the initial objective value that runs with different values of p achieve.

to $t = 5$ iterations; if the algorithm does not find a state with a better objective value for t iterations, it terminates.

There are a few interesting findings that we can extract from Figure 2. First, we notice that runs that used more processes were able to run for longer before they stopped making further progress. This empirically shows that being able to search more of the state space on each step allows us to break out of local minima quicker. Second, we see that for the most part, using more processes also leads to better overall performance (measured by tour length at termination). We suspect that this relationship does not hold between the $p = 16$ and $p = 32$ cases due to the relatively small problem size we chose for this experiment.

Another way of understanding the performance

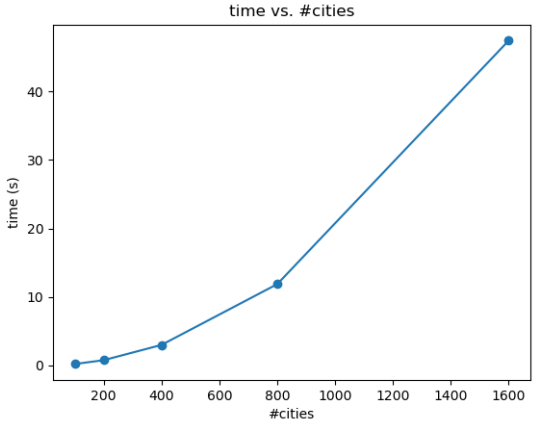

Figure 4: The number of seconds per annealing iteration for increasing problem sizes.

Problem size	Time per annealing step (s)
100	0.197
200	0.754
400	2.987
800	11.863
1600	47.436

Table 1: The number of seconds per iteration for increasing problem sizes.

of the algorithm is by looking at the percent reduction in objective value that each value of p produces. This is shown in Figure 3. Unfortunately, this is the best we can do, since we are not able to compute the shortest tour for a problem of that size.

5.2 How does the running time of the program change as the problem size increases assuming fixed resources?

To test this, we fixed $p = 32$ and let n range over the values 100, 200, 400, 800, 1,600, and 3,200. The parameter k was set to 500 this time as well, but the maximum allowed annealing steps was 100,000, since we do not care as much about convergence as we do about time per iteration. (In the figures below, we do not show the numbers for 3,200 since it timed out on the cluster.)

The results are shown in Figure 4 and Table 1. We see that the relationship between problem size and time is roughly quadratic. That is, each time the problem size doubles, the running time increases by a factor of four. This is expected, since each processor takes k steps each annealing step, and each annealing step is dominated by the construction of transition probabilities, which in

our implementation is linear in the number of nodes in the graph. Since k and n are of similar orders of magnitude in our setup, this would result in a quadratic-looking plot. The running time might also be impacted by communication costs, but since we use the same number of processes for each of these experiments, we do not think that is the cause for the relationship observed here.

6 Discussion

The goal of this project was to was to apply parallel computing principles to the simulating annealing algorithm and then adapt it to TSP, not necessarily to optimize the simulated annealing algorithm itself. In this regard, we have demonstrated proof of concept that the SA algorithm is still effective when run with separate threads on a distributed system.

6.1 SA optimization vs. parallel optimization

In this work, we operated under the assumption that all of our MPI processes (workers) should run the same number of annealing steps and stay in the same β-domain. We also reset all of the workers after a certain number of steps. While there are many avenues to explore in terms of optimizing the SA algorithm, including different schedules for varying β and more sophisticated transition strategies, it seems unlikely that these strategies would be impacted by the introduction of MPI. These changes to the algorithm are only local in the sense that they effect only a single worker's trajectory at a time and not interprocess communication. In order to utilize the benefits of our distributed system, we should strive to seek optimizations that work across the entire network.

One possible improvement could be to change the way the workers interact with each other. Currently, our algorithm resets every process after the root process collects all of the workers' minima and broadcasts the latest global minimum to all other nodes. A better solution might be to keep a certain percentage of nodes working on their own trajectories and reset the rest. This might avoid some of the strange behavior that we showed in our presentation where adding nodes could actually hamper the progress of the algorithm. We believe that this is due to the extra workers finding local minima earlier in their trajectories that beat out the global minimum trajectories in the short term. Given enough time, these trajectories should still converge to the correct solution, but the efficiency sought by adding the extra nodes is reduced.

6.2 Further parallelization using OpenMP

Another set of improvements could be the introduction of OpenMP to parallelize different worker-local aspects of the algorithm.

One such example is the construction of the transition matrix, which profiling results showed dominates the algorithm's runtime. Assuming we had access to multiple CPUs on each node, because probabilities of different transitions can be constructed independently of one another, the matrix construction could be embarassingly parallelized using OpenMP.

Another way we could leverage OpenMP would be in sampling from the transition matrix. Our current implementation uses what is essentially a scan operation over the whole transition matrix. We would have to be more careful here than with the matrix construction, but would still be able to parallelize the sampling procedure.

6.3 Difficulties in Evaluation

While we did test the strong scaling of our problem, we did not test weak scaling properties. One reason for this is the fact that increasing the problem size creates a lot more additional work than adding more processes adds capacity for. We also had some difficulties in evaluating the results definitively as we did not have the ground truth solutions to compare against our implementation. This could have been alleviated somewhat by constructing test cases with simple, known solutions, such as graphs designed with specific underlying spatial structure.

7 Conclusion

Simulated Annealing is a powerful algorithm capable of exploring large discrete state spaces to solve difficult optimization problems. However, its nature as a Monte Carlo Markov Chain Method makes it difficult to parallelize in a straightforward way. We have shown that distributed methods using OpenMPI can prove useful in cutting down the total number of iterations needed to converge to a reasonable solution. Further research in optimizing the network coordination across parallel Markov trajectories presents a significant opportunity for advancing TSP solvers, and for solving other large optimization problems as well.

8 Contributions

The code can be found here: `https://github.com/gindij/nyu_sp23_hpc_simulated_annealing`. Contributions to the project were as follows:

- Brady: initial annealing code, math background about simulated annealing
- Jack: initial TSP-related code, data generation, visualization
- Both: parallelization, debugging, experiments, paper