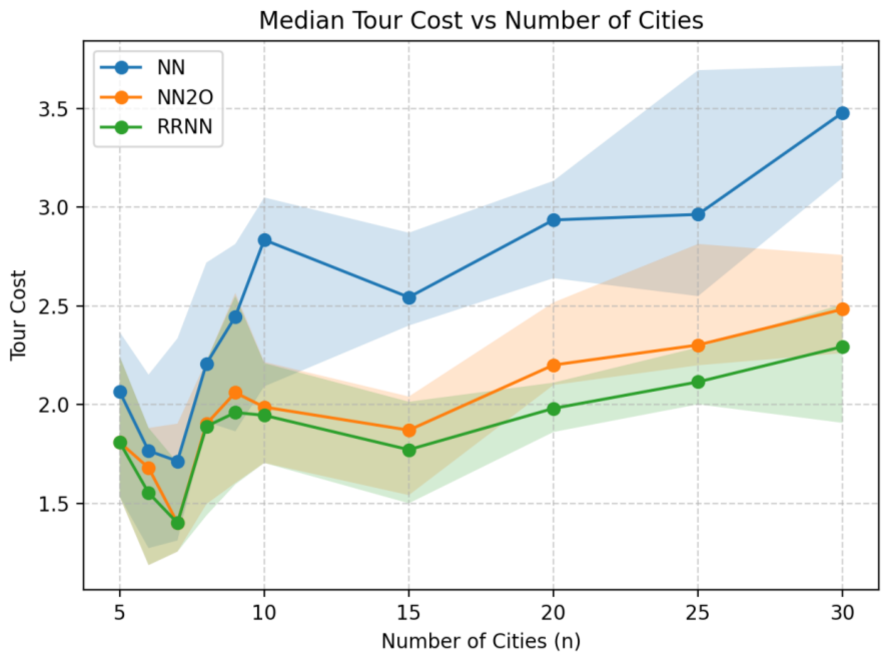
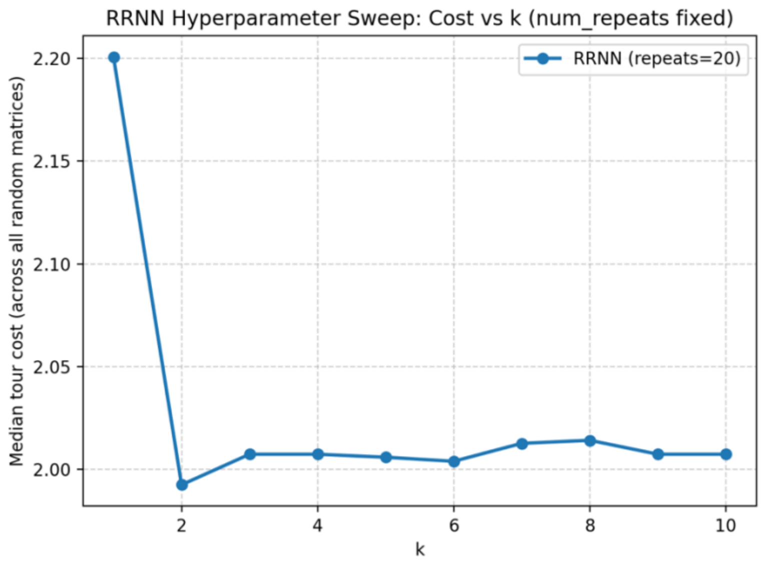
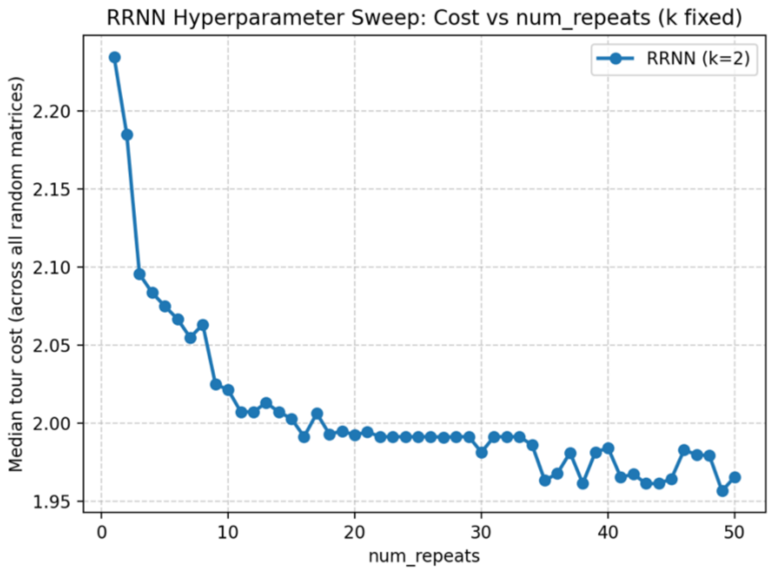
Nearest neighbor (NN), nearest neighbor with 2-opt refinement (NN2O), and randomized repeated nearest neighbor (RRNN) algorithms do not ensure optimality. They take a heuristic approach instead of A*-style exhaustive search methods. NN greedily picks its closest next city, which is usually locally optimal but produces globally suboptimal tours. NN2O is an improvement upon this by adding local 2-opt swapping, yet it is still subject to getting trapped in a locally optimal but globally suboptimal tour. RRNN further reduces this risk by adding randomness and repeating many times, which allows it to find shorter tours much more often but yet again doesn't guarantee its optimal solution. As the figure shows, NN consistently produces the highest median tour cost, NN2O performs significantly better, and RRNN achieves the lowest costs among the three.



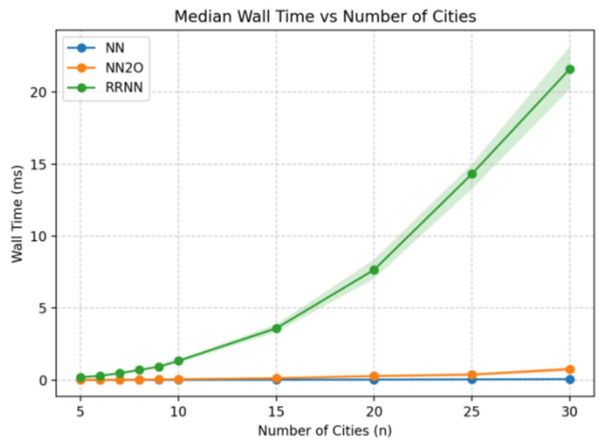
For the randomized repeated nearest neighbor (RRNN) algorithm, I chose k = 2 because it produced the lowest median tour cost across all trials most consistently. As indicated in the figure, performance significantly declines at k=1, where the algorithm is too restricted in its refinement to create lower-cost tours. It is actually higher-cost tours that are generated. Setting k higher than 2 does not lead to significant improvements in solution quality, as the median cost levels off and fluctuates only slightly. Larger k also inflates runtime without adding tangible benefits, making them inefficient for this problem size. Therefore, k=2 provides a good compromise by achieving virtually minimal cost while maintaining computational efficiency of the algorithm.

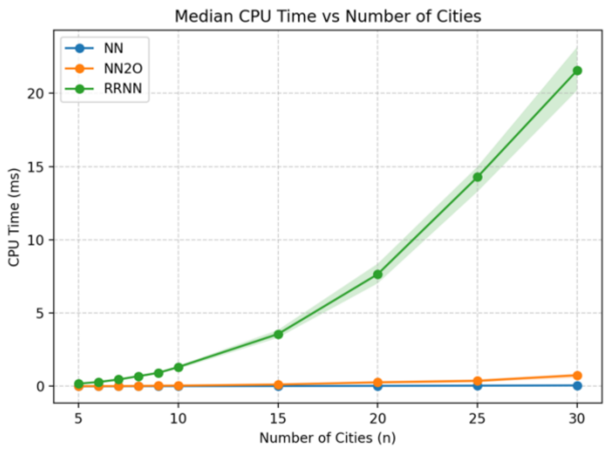


For the RRNN algorithm, I also selected num_repeats = 20 because this is roughly the point where additional repeats only provide marginal gains in tour cost while runtime is still growing linearly. As the figure illustrates, cost drops drastically when num_repeats is increased from very small numbers but by about 20 repeats the curve is flattening, implying diminishing returns in solution quality. Selecting a larger number of repeats would make accuracy improve somewhat but comes at the cost of much longer runtimes with little real-world benefit. By choosing num_repeats = 20, the algorithm gets nearly minimal-cost solutions while computation remains tractable and balances accuracy and efficiency well.



The runtime results show a clear tradeoff between speed and accuracy across the three algorithms. NN is clearly the quickest of all, with nearly zero CPU and wall time even as cities ascend. Yet this speed is paid in terms of inaccuracy, as NN is always the most costly of all three methods tours. NN2O is a compromise: it is slightly slower than simple NN but a lot more efficient, since its cost is significantly reduced by the local improvement step. RRNN has the best tour cost overall, beating NN and NN2O in terms of accuracy to a considerable extent. However, this improvement comes with a steep increase in runtime, which grows quickly as the number of cities increases because of the repeated trials. Therefore, the results show that NN is least efficient yet quickest, whereas RRNN is most efficient yet slowest of all, and that NN2O is a compromise of time and cost.





The asymptotic complexities of the algorithms explain the runtime trends seen in the figures. NN takes O(n^2) because it repeatedly picks the closest unvisited city among each of n cities. This moderate complexity matches the nearly flat runtime plot observed even for larger problem sizes. NN2O adds a refinement step of a local search that examines all candidate edge interchanges possible (specifically, all edge pairs that do not share a common neighbor). This results in an O(n^2) cost for the initial NN tour plus an additional O(n^2) or higher factor depending on the number of refinements, leading to higher runtimes than NN but still controllable growth, as indicated by the plots. RRNN repeats the NN process by a number of repeats r, which results in a runtime of O(r·n^2). This is why RRNN's runtime increases steeply as both n and r increase, making it the most computationally costly of the three. These asymptotic rates are reflected in the plots: NN grows almost flat, NN2O increases moderately, and RRNN exhibits the steepest curve due to its multiplicative repetition.

INSTRUCTIONS TO RUN:

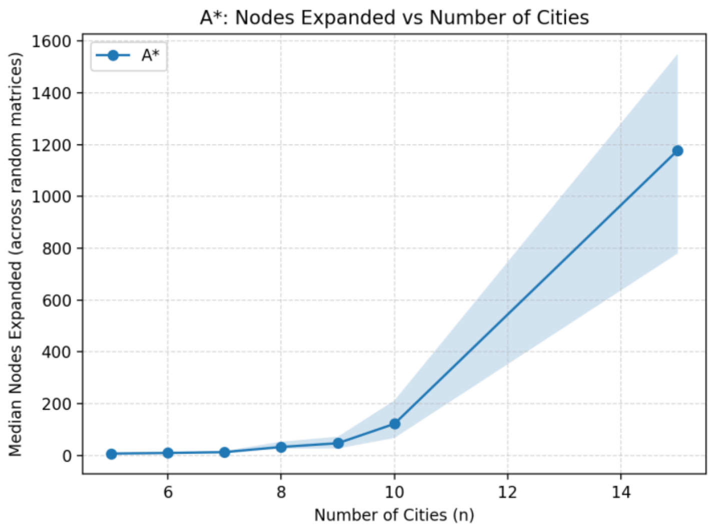
- Make sure that you have Python and NumPy installed
- Open a terminal at the file
- Run the command: python aima_nn_algs.py <matrix txt file>

This will run all 3 (NN, NN2O, and RRNN), and provide you with the path, the cost, and the runtimes

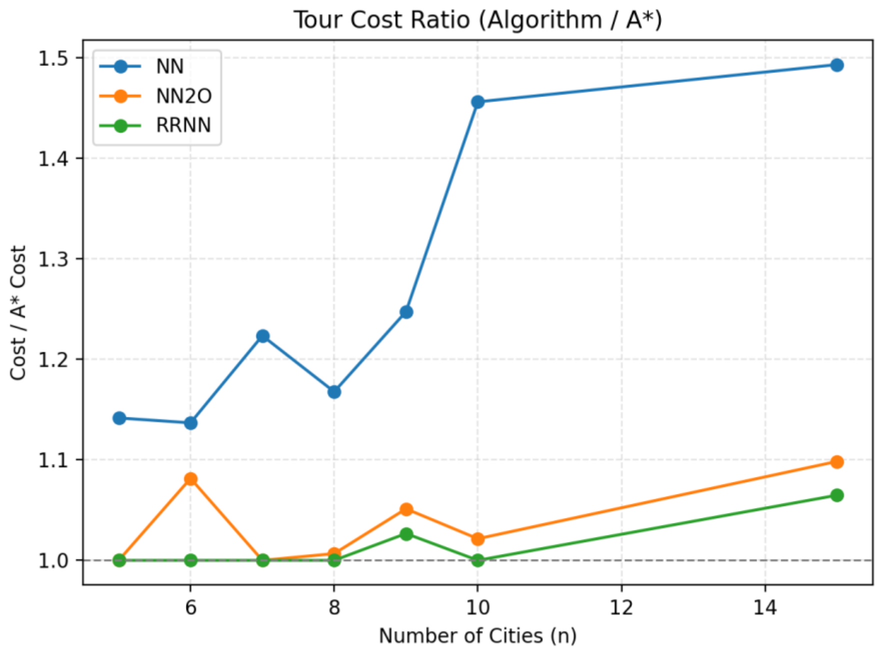
In my version of A*, an A* node is a partial tour: a tuple of visited cities in order (start, …, current). g(n) is path cost of a prefix (the total length of edges in it), and the node's priority in the fringe is determined by f(n) = g(n) + h(n), where h(n) is an admissible MST-based lower bound to complete the tour. Tuples are used for states (because they are hashable) and an unvisited set is constructed as we go. This keeps the code simple, and it is compatible with a priority queue as the fringe, and a set for the closed list. From a software engineering perspective, this requires fast heap operations for the fringe and efficient hashing for duplicate detection.

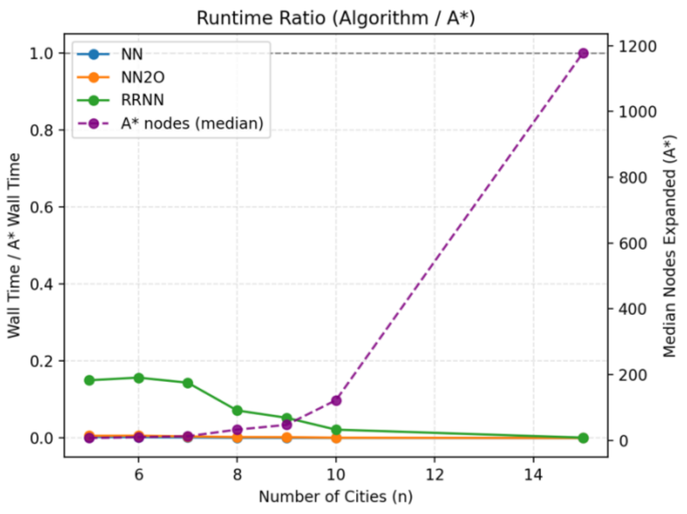
In my tests, I was able to reliably run A* up to matrices of size n = 15. Beyond that, A* became impractical, and I was never able to complete a run on a random matrix of size 20 or higher. The main reason for this was because at each step, the algorithm has to consider every single possible next city, which makes the state space grow extremely fast. This slows down all of the operations. The heuristic calculation (recomputing the MST for each node) also adds cost, but the biggest slowdown comes from the sheer number of possible states. To improve the performance of the algorithm, you could reuse heuristic results for repeated patterns of unvisited cities. You could also use a stronger admissible heuristic than MST, and cut off unpromising branches with a branch-and-bound approach algorithm.

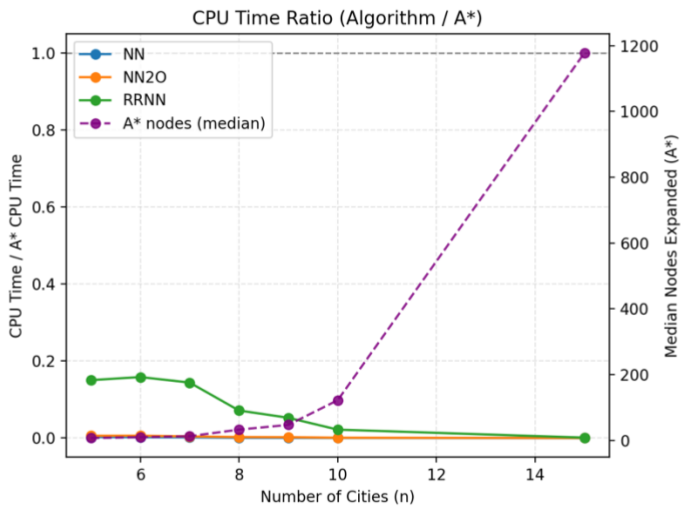
As the number of cities increased, the number of nodes increased exponentially. This is expected, since A*'s state space increases exponentially as you go deeper. Each increase in the number of cities multiplies the number of partial tours that A* has to explore before the heuristic can guarantee that the solution is optimal. This is why as we get past 15 cities, the algorithm starts to stall.



In comparison to the other algorithms we have done so far, A* is far better than NN, NN2O, and RRNN when it comes to the solution quality. No matter what, A* will find the most optimal solution to travel to all the cities and back, while the other algorithms are not always guaranteed to find this solution. The drawback is the amount of time it takes to find the solution. The heuristic methods usually come within about 5–15% of optimal, but they run much faster and use far less CPU time (see the CPU and wall-time ratio plots). NN, NN2O, and RRNN are much quicker than A*, and they are accurate enough. A* is only practical for smaller TSPs or when you absolutely need a guaranteed optimal solution. For a larger, real-world problem, A* would not be feasible. It would make much more sense to use one of the other three algorithms, as they run significantly faster and still produce good enough solutions.







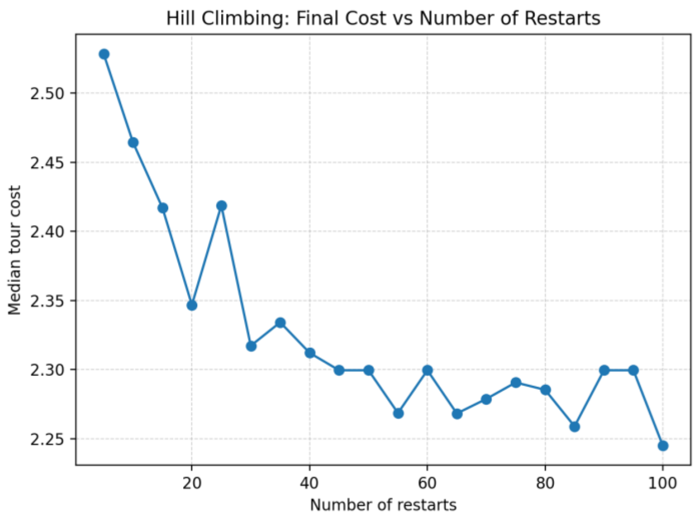
INSTRUCTIONS TO RUN:

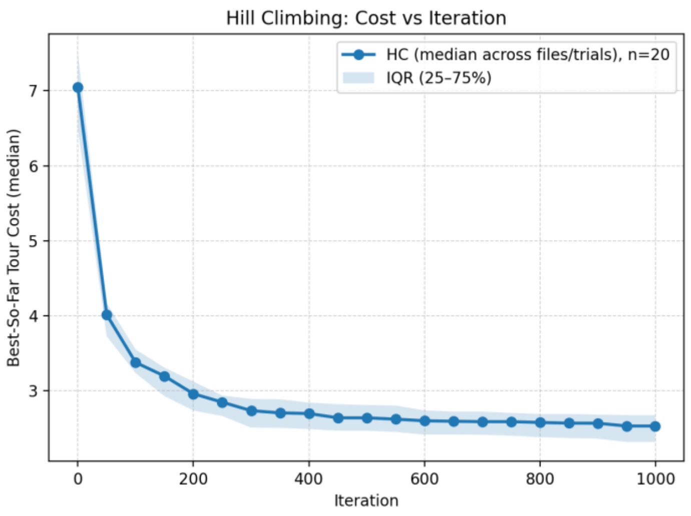
- Make sure that you have Python and NumPy, SciPy (for MST) and the AIMA search module installed
- Open a terminal at the file
- Run the command: python aima_my_tsp.py <matrix txt file>

This will run all A* and provide you with the path, the cost, and the runtimes

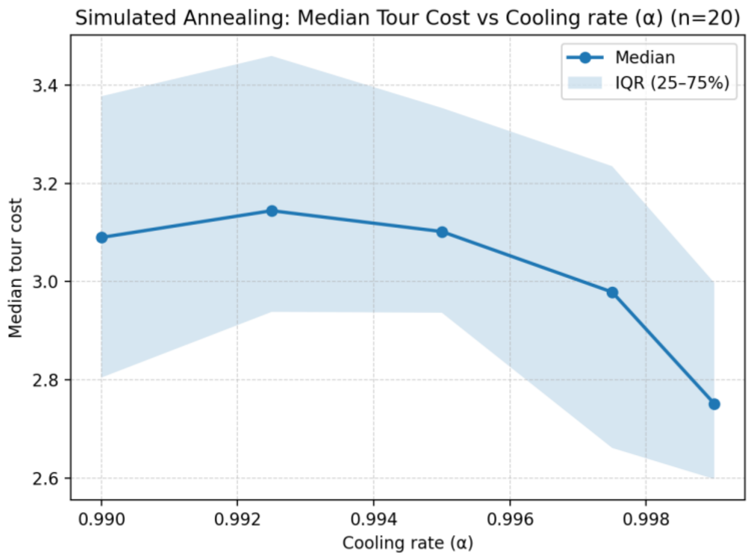
In all three of hill climbing, simulated annealing, and genetic algorithm, solutions to TSP are represented by a path as a permutation of cities in a closed cycle. For hill climbing, a neighbor is a small edit to the tour. In this case, it would be a random swap of two positions (excluding the start/end node). The algorithm moves to the neighbor as the optimal solution only if it is strictly better than the current solution. For simulated annealing, the definition of neighbor is the same, but this algorithm sometimes accepts worse neighbors. The probability of accepting a worse neighbor is $e^{\frac{score' - score}{T}}$, where score' is the new solutions score (or cost), score is the current solution's score, and T is the temperature parameter. The probability of accepting a worse solution decreases exponentially with how much worse it is and increases with the temperature T. Lastly, for generic algorithm, neighbors come from paths evolving. Two parent tours are combined to produce a child. After a crossover, a tour might be randomly mutated, swapping two cities. The algorithm then decides which tours to move to the next generation using elitism. This means that the best solutions are always passed down.

The hill climbing algorithm worked the best with more restarts, and a reasonable cap on iterations. More restarts improved the odds of finding a better solution, but runtime increased linearly. At a certain point, more restarts are were found to be diminishing in return. Looking at the plot charting the final cost of the solution hill climbing produced against the number of restarts, I found this number to be around 50 restarts. For the number of iterations, the algorithm saw the steepest improvement in the first 200 iterations, but not much of a difference from 200 to 1000 iterations. A good middle ground was about 300-400 iterations, where the curve started to round out and the cost wasn't being reduced much further.

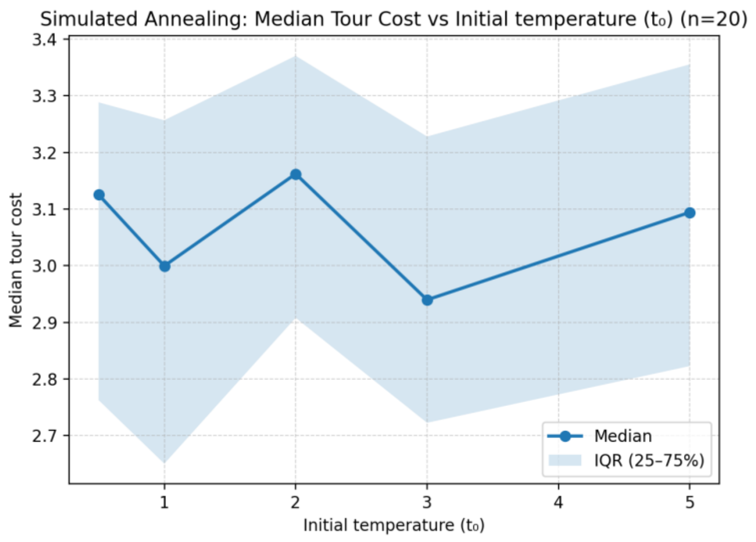




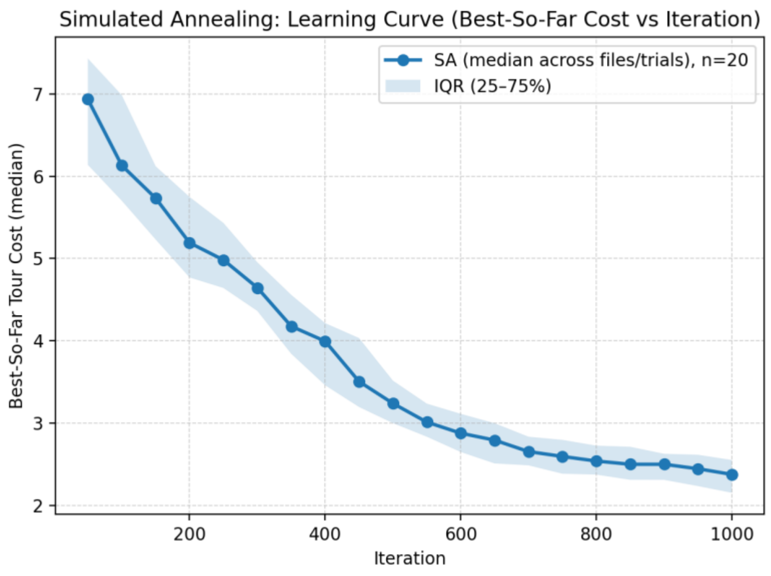
For simulated annealing, the two main hyperparameters I experimented with were the cooling rate (α) and the initial temperature (t_0). The cooling rate controls how quickly the "temperature" drops, which in turn decides how willing the algorithm is to accept worse solutions as it progresses. A slower cooling rate (closer to 1.0, such as α = 0.998) led to better median tour costs because the algorithm had more time to explore and escape local minima. Slower cooling means the algorithm spends more time exploring, so it usually finds better solutions but takes longer to finish. On the other hand, faster cooling makes it run quicker, but the solutions are worse because it stops exploring too soon.



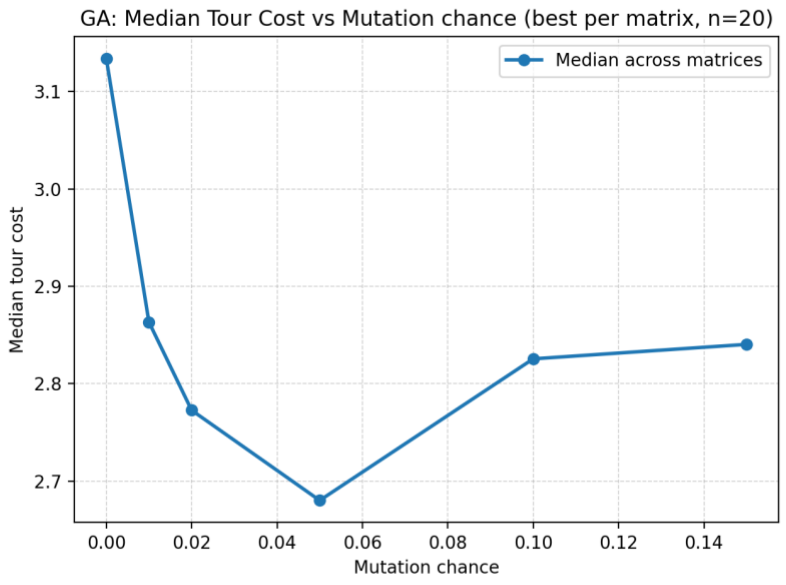
For the initial temperature (t_0), I found that small values worked about as well as larger ones. The difference was not as pronounced as with the cooling rate, but setting t_0 too high can waste time exploring unhelpful random moves, while setting it too low risks the algorithm behaving too much like greedy hill climbing. A moderate starting temperature gave a reasonable balance.



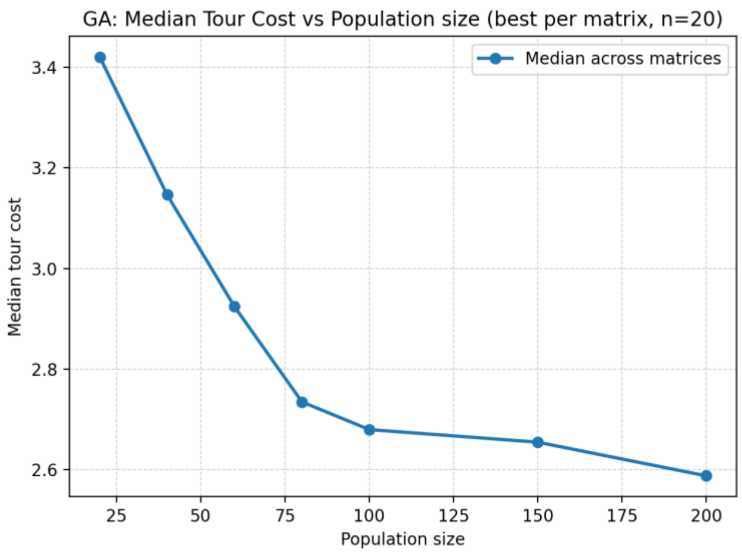
The number of iterations had more of an impact though. The graph shows that most of the improvement happened in the first 600-800 iterations, and then the median cost leveled off. So more iterations than this add more runtime without much of a gain, so the optimal balance was around 600-800 iterations.



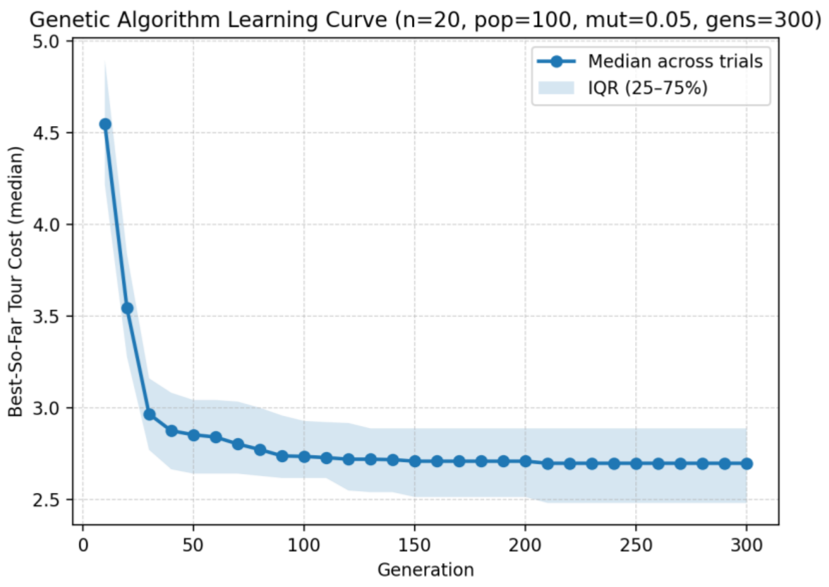
Mutation applies random alterations to solutions (e.g., exchanging two cities). If there is no mutation, the population converges rapidly and may become stranded in suboptimal solutions. If there is too much mutation, the algorithm is nearly random and misses the advantage of crossover. The results of my code show the optimal performance to be near a 5% mutation rate. The rate applies enough randomness to get out of local minima without allowing the crossover operator to further enhance solutions.



The population is the size of candidate tours retained each generation. Small populations (such as 20–40) converge rapidly but don't search thoroughly and converge to poorer solutions. Big populations (100–200) provide better diversity and yield better solutions, however, they require more computation per generation. My graph reveals that making bigger increased the population size always decreased the median tour cost, although with diminishing return beyond 100-150.



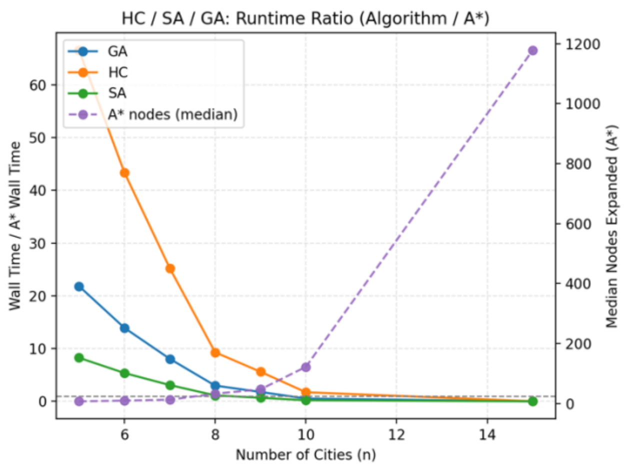
Generations are the algorithm's iterations. Each generation uses crossover, mutation, and selection to make the new population. The earlier generations make the largest improvements (sharp decline in cost), but improvements slow down and the line peters out after about 100–150 generations. Many generations then make fine-tuning of the solution, but with very small improvements relative to the increased runtime.

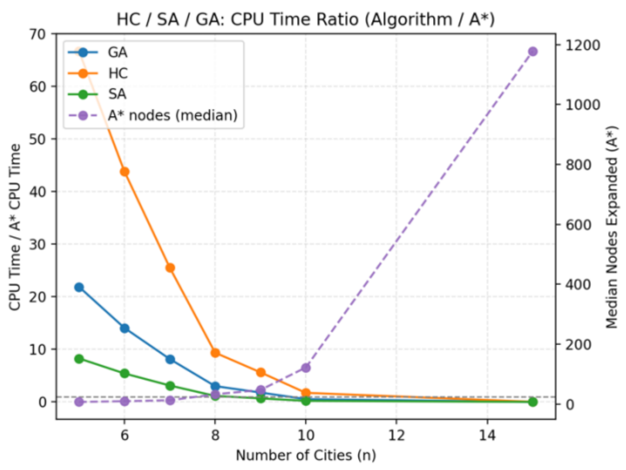


The local search and evolution algorithms (simulated annealing and genetic algorithms) and the hill climbing algorithm found better solutions than the nearest neighbor (NN, NN2O, RRNN) heuristics. The NN algorithms are extremely fast, but they're greedy: they choose the next nearest city without planning ahead, and thus frequently get stuck with a locally good but globally bad tour. Hill climbing is better than NN by starting with a random tour and repeatedly improving it, but this causes local optima getting stuck. Simulated annealing prevents this by sometimes taking worse moves, which allows it to escape local optima and typically achieve better solutions than pure hill climbing. The genetic algorithm performed best of all, as it keeps a diverse population and pools information across multiple tours.

The tradeoff is time. NN and its variations run much faster because they construct only one tour, whereas the other algorithms run longer to search and perfect solutions. The local search and GA algorithms, in my runs, frequently came up with tours that were nearly optimal (lower cost), but took longer to run. This makes NN preferable to quick approximations on large instances, and the other algorithms preferable where solution quality is prioritized over speed.

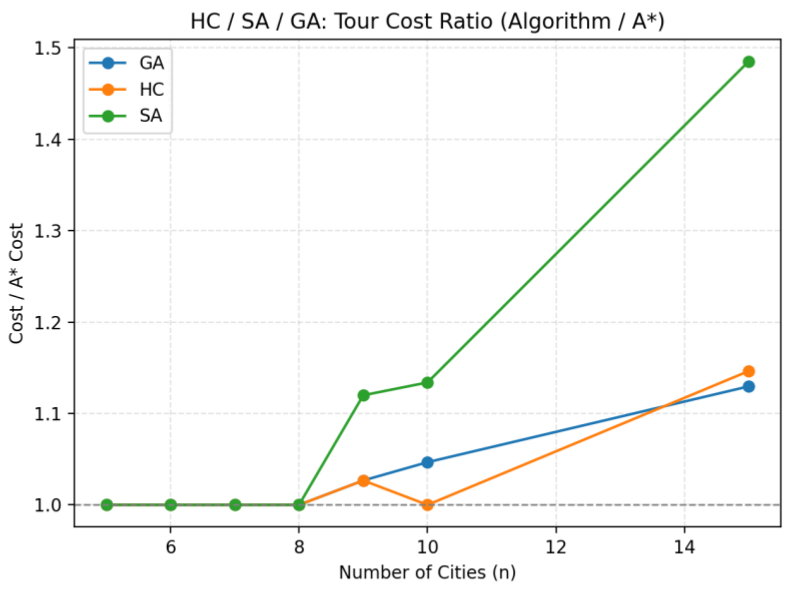
Comparing these algorithms to A*, A* ensures an optimal tour, but its run-time grows rapidly as the number of cities increases. For me, A* became impractical beyond about 15 cities. The plots of runtime ratios explain why: as n grows, the number of nodes A* expands rises enormously, and so both CPU and wall time blow up. In contrast, hill climbing, simulated annealing, and the genetic algorithm scale much better. Their wall time and CPU time relative to A* drop to nearly zero as the number of cities grows. These algorithms are far more realistic for larger TSPs, even though they don't guarantee optimality. They usually find solutions close to optimal (within a small percentage) but much faster.





You would want to use A* if you're solving a small problem and absolutely require the exact best tour (e.g., in cases where even a slight cost difference matters, such as microchip design or a precise logistics problem with very few nodes).

You would use hill climbing, simulated annealing, or GA once the number of cities gets larger, or when runtime and resource limits matter more than provable optimality. In these cases, hill climbing, simulated annealing, and the genetic algorithm give "good enough" solutions quickly, which makes them practical for real-world delivery routing or scheduling with dozens or hundreds of sites, where A* would not be feasible.



Hill climbing, simulated annealing, and genetic algorithms all employ randomness to improve on deterministic algorithms such as nearest neighbor or A*. Nearest neighbor always takes the same path direction, and A* always opens new nodes to expand using the same sequence, and both can get stranded or be impractical. The addition of randomness assists these other algorithms to search wider: hill climbing employs random restarts to prevent being trapped within the same local minimum, simulated annealing sometimes tolerates worse moves to get out of bad regions, and genetic algorithms add diversity by random crossover and mutation. The randomness avoids the search getting too specific and widens the likelihood of discovering better solutions, even though they're not guaranteed to be optimal.

One of the real-world uses of TSP is microchip design, and here wires between components need to be routed with minimum inefficiency. Because circuits that scale small but require perfect precision, even minute inefficiencies make a difference. That's why A* is the optimal choice because the optimal tour is guaranteed and can be applied to small cases where preciseness is paramount.

A use second is last-mile delivery routing, in which trucks or vehicles need to make numerous stops. Dozens or even hundreds of stops make runtime here preferable to optimal perfection. A genetic algorithm performs best here because this algorithm scales well to bigger inputs and gives us near-optimal routes within reasonable runtime.

Another is warehouse order picking, whereby workers will pick products in a predetermined order with the highest speed possible. Because the order flows constantly and solutions should be found rapidly, heavy computation is not possible. A quick heuristic such as hill climbing with restarts is fine, and this will quickly provide decent tour that continues to keep the operations flowing smoothly.

Lastly, airline crew schedules represent a massive and complicated problem that encompasses costs and fairness constraint. The search space is enormous and with many local optima, and hence a greedy or deterministic procedure will get trapped. Here, simulated annealing is optimal, for its willingness to periodically make worse moves can aid getting out of local minima and discovery of quality schedules.

These examples highlight that no single algorithm is universally best, and the choice depends on priorities. When exactness matters and the problem is small, A* is ideal. For larger, real-world cases where speed and scalability are crucial, heuristics and metaheuristics like hill climbing, simulated annealing, or genetic algorithms offer practical, "good enough" solutions.

INSTRUCTIONS TO RUN:

- Make sure that you have Python and NumPy installed
- Open a terminal at the file
- FOR HILL CLIMBING:

```
python aima_hill_climbing.py <matrix_file.txt> <num_restarts> <max_iterations>
```

- FOR SIMULATED ANNEALING:

```
python aima_simulated_annealing.py <matrix txt file> <alpha> <t0> <iterations>
```

- FOR GENETIC ALGORITHM:

```
python aima_genetic.py <matrix_file> <mutation> <pop_size> <generations>
```