A1. Genetic algorithm (GA) implementation

The algorithm is encapsulated as a class, $GeneticAlgorithm$, to which various parameters are passed: mutation probability (P_m), crossover probability (P_c), tournament size (k), population size (N), number of generations (g), the number of cities n, and the matrix containing the distances between them. The class contains a NumPy matrix, $population$, of shape $(n,\ N)$. Each individual solution is a row of this matrix; all operators function by manipulating it. Cities are internally indexed from 0 to $(n-1)$, instead of 1 to n, as when they are read from file; doing so eliminates off-by-one errors arising from the discrepancy between the indexing of the matrix containing distances between cities, from 0 to $(n-1)$, and the cities themselves. Just prior to writing a tour to file, the index of each city was incremented by 1 to compensate. An object-oriented approach was initially used, whereby a $Population$ class contained a group of $Individuals$, but it was found to be significantly slower, and incur greater memory usage, than an approach based solely on the manipulation of NumPy arrays, so was discarded. A GA must find the right balance between *exploration* of the search space and *exploitation* of areas showing promise, which is governed by *selection pressure*, a vaguely defined notion of the rate at which a population converges. Too high, and the GA does so too early (not enough exploration); too low, and it may never (not enough exploitation). Finding this balance is matter of fine-tuning the parameters mentioned above and choosing the optimum variants of the three operators of a GA: selection, crossover, and mutation.

Selection is "the task of allocating reproductive opportunities to each individual" (Beasley, Bull, and Martin, 1993). The initial selection mechanism implemented was roulette wheel, conceptually the simplest: individuals are allocated slices of a roulette wheel in proportion to their fitness. The wheel is spun N times. The individuals selected are then placed into a mating pool. However, roulette wheel selection performed poorly, as the literature asserts: it tends to result in too high a selection pressure, and therefore premature convergence. Rank selection was implemented next, whereby individuals are accorded a rank, inversely proportional to their fitness, and are allocated a larger slice of the wheel the smaller their rank is. However, this required sorting the population by fitness, an expensive operation requiring the calculation of fitness for each individual, and sorting an array of potentially thousands of individuals, and necessitating using Python lists, as NumPy does not provide the functionality to sort by key (i.e. a lambda returning an individual's fitness). So: tournament selection was settled upon. Goldberg and Deb (1991) found it to be equivalent to rank selection, with suitable parameter adjustment. In tournament selection, k (typically 2) randomly selected individuals compete head-to-head, and the winner is placed into the mating pool. A k of 1 is random selection; a k of N selects only the best individual. The larger k, the greater the selection pressure. Tournament selection has several other benefits: it is efficient to code (requiring only two lines in Python), works on parallel architectures, and allows for selection pressure to be easily adjusted.

The next operator, crossover, facilitates the inheritance of 'characteristics' from parents and is conceptually responsible for exploiting promising regions of a search space. Larrañaga et al. (1999) found genetic edge recombination crossover (ER) to be the best crossover operator, and so this was implemented first. However, ER is based on the number of edges that an unvisited city has, and in the provided samples, the graphs are complete, so ER is little better than random. Partially-mapped crossover (PMX) was chosen next, as the author had used it successfully before. However, OX1, which Larrañaga et al. (1999) found to be second-best in their comparison, outperformed PMX on all city files, was simpler to implement, and ran faster. OX1 constructs an offspring by randomly choosing a subtour from one parent, removing the vertices contained within that subtour from the other parent, then inserting the subtour in a random position in the parent. $P_c = 0.7$ is a typical value.

Finally, mutation. Mutation is designed to overcomes the problem of a whole population sharing the same gene, and thus serves to encourage exploration of the entire search space (Mitchell, 2002). The displacement mutation (DM) operator was chosen as Larrañaga et al. (1999) found it to be one of the best mutation operators, alongside ISM (insertion mutation) and IVM (inversion mutation). All are

very similar: ISM removes only one vertex, whereas DM removes a subtour; IVM inserts the subtour reversed, and DM as it was removed. Sastry, Goldberg and Kendall (2005) advise a P_m of 0.05, based on De Jong's results (De Jong, 1975).

One modification tested was the implementation a greedy 'nearest neighbour' algorithm for generating the initial GA population. Although this improved the final result, the cost to runtime was prohibitive. Additionally, although this approach ensures the initial genetic material is of high quality, it is very homogenous: there was little further development of the population. This modification was therefore discarded.

A2. GA results

City file	Best tour length	Mean k	Mean P_c	Mean P_m	Mean generation
NEWAISearchfile012	56	5.6	0.45	0.45	49
NEWAISearchfile017	1444	6.1	0.49	0.51	333
NEWAISearchfile021	2549	6.4	0.38	0.48	296
NEWAISearchfile026	1473	6.2	0.40	0.38	407
NEWAISearchfile042	1071	6.8	0.40	0.36	797
NEWAISearchfile048	12806	7.2	0.35	0.44	885
NEWAISearchfile058	25928	7.2	0.49	0.33	934
NEWAISearchfile175	21827	7.9	0.60	0.26	980
NEWAISearchfile180	1950	8.3	0.35	0.41	980
NEWAISearchfile535	50006	8.2	0.55	0.33	1992

These results are the best-ever generated by any variation of GA. Typical parameters for N and g were ≈ 1000 and ≈ 4000 respectively and took several hours to run. The mean parameters listed are not necessarily those used to achieve these lengths; rather, they are the values that generated a tour with a length within 5% of the best tour generated by *any* combination of parameters tested iteratively, as detailed below.

A3. GA experimentation

To test the GA, and fine-tune parameters, iterative tests were run, varying the parameters k, P_m, and P_c while holding N constant at 100 and g at 1000. These tests form the basis for the following. It was found that the greater n, the greater the optimum value of k was, ranging from $k = 5$ at $n = 12$ to $k = 8$ at $n = 535$. The larger n, the factorially greater the search space ($n!$), and so the greater the selection pressure needed to induce meaningful convergence, such that population actually improves. For general usage, a k between 6 and 7 is recommended. It is likely, however, that the optimum value of k is also influenced by N; it is believed that a larger N will again require a greater selection pressure, and so a greater k.

The optimum value of P_c found to lie in the range $[0.35, 0.6]$, with larger values of n favouring larger values of P_c, though not as strongly as the correlation found between k and n. For general usage, a value of $P_c = 0.45$ is recommended. This is believed to be a consequence of the need for greater selection pressure as the search space increases: the exploitation of fit individuals becomes more important. This led to the prediction that an inverse relationship between n and P_m, as between n and P_c, would exist, and this was indeed confirmed: the greater n, the smaller the optimum value for P_m, at $P_m = 0.5$ for $n = 12$ and $P_m = 0.25$ for $n = 535$. This suggests that P_m should be determined relative to n, as Ochoa, Inman, and Buxton (2000) note: "it has been argued that optimal per-locus mutation rates in GAs are proportional to selection pressures and the reciprocal of genotype length". The larger the search space, the greater the proportion of lower-quality solutions, and therefore the less advantageous it is for the GA to explore those solutions, rather than exploiting the existing population. The opposite is true for P_c, which explains the reversed correlation observed above.

The final parameters to consider are N and g. Increasing N to thousands of individuals ensures the population is genetically diverse, and therefore unlikely to converge, but performing operations on such a large number is very time-consuming, and likely the render the GA unusable for real-life applications. 200 was found to be acceptable. The same logic determines the value of g, with the caveat that large values of g can waste computation time if the population has converged. The smaller the diversity of the population, the more likely this is; g should therefore be set as some non-linear function of *both* n and N. 1000 was found to be a suitable compromise, although with large n best results were frequently obtained after ≈ 950 generations, suggesting the algorithm could have benefited from a greater runtime.

B1. Simulated annealing (SA)

The algorithm is again encapsulated in a class $SimulatedAnnealing$, to which various parameters are passed, based on Geltman's (2014) basic implementation. As with the GA, a solution is just a NumPy array containing cities indexed from 0 to $(n-1)$. Part of the attraction of SA is the simplicity of implementation. There are three functions: the function used for the acceptance probability; the function for generating neighbours; and the function determining the temperature schedule. There are two key parameters: T_{min}, the point at which SA terminates; and α, which controls the rate at which the temperature decreases.

The most important of functions was found to be $generate\_neighbour$. Typically, when using SA to 'solve' the TSP, neighbours are calculated by switching two adjacent randomly chosen vertices. This introduces little variation, however. The first modification made was to let the number of vertices 'switched' depend on temperature: the higher the temperature, the more vertices are switched. When the temperature is very high, neighbours are completely new random permutations. As the annealing progresses, and the algorithm tends towards a solution, neighbours vary less and less, until a minimum of two vertices are switched. This mechanism was intended to ensure that SA explored the search space at high temperatures and *exploited* it in later iterations. However, this still led to disappointing results. Further research (Schneider, 2014) revealed an alternative: reversing a random sublist of the tour, such that the length of the sublist is independent of temperature. This simple change greatly improved the performance of the algorithm; it frequently outperformed the GA, with considerably less computation, despite being much simpler.

The second most important was found to be $temperature\_schedule$ function. The function first chosen was the original used by Kirkpatrick et al. (1983) in their seminal paper on SA where, for each iteration, the temperature $T \coloneqq T \cdot \alpha$. In such a schedule, the temperature decreases quickly initially, and then slowly to close to 0. Other cooling schedules considered include those discussed by Nourani and Andresen (1998) but were ultimately dismissed either for complexity of implementation or lack of improvement over Kirkpatrick's original. Determining combinations of T_{min} and α that led to acceptable runtimes took some experimentation. The closer α is to 1, and the closer T_{min} is to 0, the longer the SA takes to terminate. Consistently good results balanced with reasonable runtimes were found with values of $\alpha = 0.99995$ and $T_{min} = 0.0001$. Changing either of these values by one order of magnitude in the appropriate direction resulted in an hours-long runtime. However, a variation of SA (adaptive tabu-), presented by Azizi and Zolfaghari (2004) was later found. Adaptive simulated annealing (ASA) is a subset of SA, whereby parameters are themselves fine-tuned by heuristics as the algorithm runs: a meta-heuristic, of sorts. In Azizi and Zolfaghari's algorithm, the temperature, instead of declining gradually, is maintained dynamically, controlled by the following function:

$$T = T_{min} + \alpha \ln(1 + r)$$

Where T, T_{min} and α remain as in basic SA. r is a counter variable: the number of consecutive upward moves performed by SA, where 'upward' means moving to a solution of higher cost. The rationale behind is this as follows. In early iterations, the likelihood of downwards moves (to lower cost solutions) is high, so a low temperature is desirable. As the search progresses, the chance of getting

trapped in a local maximum increases, and thus a *high* temperature (as controlled by r) is desirable, to give SA the opportunity to explore higher-cost neighbouring solutions, and therefore escape. Azizi and Zolfaghari set T_{min} and α to 1; the same was initially done here. They also introduced a tabu list to the algorithm, consisting of previously-used solutions. This prevents SA returning to local optima that have already been explored. These additional features were found to improve the SA and had the interesting side-effect of requiring run-time to be explicitly set, a further advantage. This parameter was thus added to the $SimulatedAnnealing$ class; it was defaulted to $60s$.

Thirdly, the $accept$ function. This function determines whether a solution should be taken as the new 'best' solution that SA has found. It is key to the determining the efficacy of SA. The more 'lenient' the $accept$ function, the greater the likelihood of SA accepting a bad solution, and thus potentially escaping a local optimum; the less 'lenient', the more the SA explores the optimum in is currently in. It thus has parallels with the selection pressure that characterises a similar trade-off in a GA. The typical $accept$ function (Kirkpatrick et al, 1983) is as follows:

$$accept(new, old) = \begin{cases} 1, & cost_{new} < cost_{old} \\ e^{\frac{cost_{new} - cost_{old}}{T}}, & cost_{new} \geq cost_{old} \end{cases}$$

The likelihood of an inferior solution being chosen decreases as temperature decreases. In effect, the behaviour of the SA shifts from exploration in early iterations to exploitation in later iterations. The function above was the one used.

To improve the performance of SA, a greedy 'nearest neighbour' algorithm was used to generate the starting route. While this resulted in a small initial hit to run-time, particularly for the larger city files, the greater solution quality ensured it was worthwhile. To further assist the algorithm, the all-time best solution is saved within the SA class, and that solution is returned on completion, rather than the current best; this ensures that, even if the algorithm terminates in a local maximum, the best solution is still used.

B2. SA results

City file	Best tour length	Mean α	Mean T_{min}	Mean time
NEWAISearchfile012	56	0.910	5.0	0.00
NEWAISearchfile017	1444	0.906	5.5	6.90
NEWAISearchfile021	2549	0.910	5.1	8.17
NEWAISearchfile026	1473	0.910	5.0	7.87
NEWAISearchfile042	1055	0.910	3.6	21.77
NEWAISearchfile048	12283	0.918	5.6	25.94
NEWAISearchfile058	25395	0.910	5.0	9.27
NEWAISearchfile175	21415	0.910	5.0	38.78
NEWAISearchfile180	1950	0.910	2.86	0.30
NEWAISearchfile535	48878	0.910	5.0	52.12

These results were generated by greedy adaptive-tabu SA. As the table demonstrates, the mean values of $\alpha = 0.91$ and $T_{min} = 5$ generate results within 5% of the optimum in seconds and were consistent for most city files, with anomalies in T_{min} occurring only for NEWAISearchfile042 and NEWAISearchfile180, likely a consequence of a distance-matrix with an unusually broad range of distances.

However, both of the optimum values recorded lie close to the mean of the values iteratively tested ($\alpha = 0.90$ and $T_{min} = 5.0$), implying that their significance has been misunderstood, a subject for further investigation. Nevertheless, it has been shown that SA achieves significantly better results than a GA, in significantly shorter time, and with markedly less complexity.

C. References

Azizi, N. and Zolfaghari, S. (2004). Adaptive temperature control for simulated annealing: a comparative study. *Computers & Operations Research*, 31(14), pp.2439-2451.

Beasley, D., Bull, D. and Martin, R. (1993). An overview of Genetic Algorithms: Pt1, Fundamentals. *University computing*, 15(2), pp.58-69.

De Jong, K. (1975). *An analysis of the behaviour of a class of genetic adaptive systems*. Ph.D. University of Michigan, Ann Arbor.

Geltman, K. (2014). *Katrina Ellison Geltman*. [online] Katrinaeg.com. Available at: http://katrinaeg.com/simulated-annealing.html.

Goldberg, D. and Deb, K. (1991). A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In: J. Rawlins, ed., *Foundations of Genetic Algorithms*. San Mateo: Morgan Kaufmann, pp.69-94.

Larrañaga, P., Kuijpers, C., Inza, I. and Dizdarevic, S. (1999). Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review*, 13(2), pp.129-170.

Nourani, Y. and Andresen, B. (1998). A comparison of simulated annealing cooling strategies. *Journal of Physics A: Mathematical and General*, 31(41), pp.8373-8385.

Ochoa, G., Inman, H. and Buxton, H. (2000). Optimal mutation rates and selection pressure in genetic algorithms. In: *Genetic and Evolutionary Computation Conference*. Las Vegas, pp.315-322.

Sastry, K., Goldberg, D. and Kendall, G. (2005). Chapter 4 - Genetic Algorithms. In: E. Burke and G. Kendall, ed., *Introductory Tutorials in Optimisation and Decision Support Techniques / Search Methodologies*. Springer US, pp.97-125.

Schneider, T. (2014). *The Traveling Salesman with Simulated Annealing, R, and Shiny*. [online] toddwschneider.com. Available at: http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/.