AI Search Report

Introduction:

The Travelling Salesman Problem (TSP) is a problem where a traveling salesman must find the shortest route to travel to each city on his list while only stopping at each city exactly once, returning home at the end. The reason why TSP is an important problem in computing is since it is NP-hard; there is no known algorithm that can find the optimal tour length within polynomial time. This means that it provides a good testbed to evaluate the effectiveness of search algorithms.

The algorithms below will treat TSP as an optimisation problem where the goal will be to find the state with the shortest/optimal tour length. A state in our case will be an array of all cities in the order visited, where each city except for the first one appears in the list exactly once, and the last city being the same as the start one. Both algorithms are examples of local search algorithms since they search around the local neighbourhood of a state and over time converge towards the global optimal solution. They do not concern themselves with the path to the result.

The libraries that were imported are math and random. There is also "CityReader" which is the py file containing the city data reader that uses numpy.

File Reader:

This function is used by both algorithms to read the city data and must be in the same directory as them to function.

The file reader returns a 2-tuple – toursize n and a numpy matrix with all the distance data. A numpy matrix was chosen to store the data because they are easy to index to get both the cost between cities and the cities themselves. Additionally, the in-built printing function for numpy matrices provides a very clean visualisation of the matrix for bug-fixing purposes.

Algorithm A: Simulated Annealing (SA)

Implementation:

My implementation of SA accepts 3 parameters, the filename pointing to the file with the relevant city data, constant cool which corresponds to β and initial temperature T(0). The temperature schedule is defined as T(t) = T(t-1) / β where β is a constant and t is the iterative time.

After initialising E (difference of quality between current state and successor state) and T (current temperature), the algorithm builds an initial state randomly using a secondary function "buildState", which returns the cost of the state as well. This function takes the city data and toursize n as parameters and uses them to create a random valid state. Throughout execution the 2-tuple "best" is used to store the tour and cost of the state with the least cost ever found in the search.

The algorithm then enters an infinite while loop where it will check to see if the temperature has cooled (T <= 1) and if true return "best". Otherwise, the algorithm will generate a successor state by applying the transition function onto the current state. The transition function is simple, it takes the city data matrix and a copy of the current state and modifies this state by swapping two random cities around and then recalculating the cost. It returns the new current state (successor) and the associated cost. The algorithm then computes the difference in quality between this successor and the current state by subtracting the tour length of the successor from the tour length of the current state.

If the successor is better than the current one, it replaces it. Furthermore, if it is also better than the current best state, that will also be replaced by the successor. If the successor is not better than the current state then the probability function $e^{\Delta E/T}$ is used where ΔE is the difference in quality of the current state and potential successor (note that this is always negative). This helps us escape local minimums. At the end of the loop the temperature T is cooled with accordance to the formula T = T / cool.

Best Results:

Toursize(n)	12	17	21	26	42	48	58	175	180	535
Tourlength	56	1444	2549	1479	1090	13823	25395	22477	2970	58308

The above results were found using a random state as the initial state.

Experimentation:

When I initially implemented this algorithm, I adopted a cooling schedule that was dependent on the difference ΔE of the successor state. T = T / abs(E) was the first version of this. The idea behind this was that the larger the change in the quality, the closer to the optimum we would get, hence by cooling more with a larger E, we would save on computation time. If E was 0, then the algorithm would not cool for that iteration. In practice, this cooled the algorithm way too fast so the it did not have the time it needed to get anywhere. For the smaller values of n (12-17) the cost of the initial random state was halved but for the larger ones it was almost useless. I then changed this formula to T = T / (1 + abs(cool * E) where cool <= 1 (very small e.g. 0.00002). The results were much better for most files. However, it only worked well for city files where the distances between the cities were similar. The files where these differences were large lead to early termination and bad results. To improve consistency, I elected to remove E and instead use T = T / β where β >= 1 (but only just e.g. 1.00002), this began to achieve the best results for quite some time, so I was happy to leave it like this.

Another important parameter is the initial temperature T. When I first began to improvise, this value was at 100. Later when I was happy with my cooling schedule, I decided to increase it to 1000 to see if results increased amicably. With the schedule T = T / β this did not change the overall running time of the algorithm much due to the proportional nature of the schedule, but I wanted to see how changing the value of the initial temperature would affect the values of the probability function. In fact, larger values of T gave rise to a longer period of larger probabilities early in the search, giving the algorithm more opportunities to escape local minima. It is hard to quantify the improvement that this change brought to results, but the implication is that an improvement should be expected.

I also tampered with the transition function, previously a transition would be a swap between two random cities in the state. I toyed with the idea of involving a heuristic where one of the neighbours that were furthest apart would be picked to be swapped with another random city. The idea being that the costliest connection would be changed since it has the highest chance of seeing an improvement from a logical standpoint. Unfortunately, results arising from this heuristic were woefully bad, so bad in fact, that often the final state would be worse than the initial state that was chosen.

The initial state that the algorithm began with was also of interest to me. At one point I decided to see what would happen if instead to creating a random state, I used a nearest neighbour algorithm to construct the initial state and use SA to improve on it. The results were mixed, in quite a few cases SA failed to improve upon the solution given by the nearest neighbour most likely since it started stuck in a local minimum. However, for other cases the SA was able to achieve results that were better than both itself and nearest neighbour could have achieved than if they were alone.

Algorithm B: Genetic Algorithm (GA)

Implementation:

My GA implementation takes 4 parameters; filename, p_mutate, Pop_size, gen_count. “filename” is the name of the file that contains the city data, p_mutate is the probability that a child is mutated, Pop_size is the number of members in the population and gen_count dictates the number of generations for which the algorithm will run.

The algorithm starts by reading the file then creating an array P and value F. It begins populating P with random states using the same "buildState" algorithm discussed until Pop_size is reached. Each member of P is represented as a 3-tuple containing an array with the path, a float for fitness and an integer for solution cost. The algorithm also keeps track of the value F, which is the total sum of fitness in the population where fitness f of a member is computed using 360 / tourlength. 360 was chosen because

The algorithm then assigns int gen, array newP and float newF, the first being the generation count and the other two being self-explanatory. After which the main while loop is entered. In the while loop we assign X and Y as members of the population that will be picked for breeding and we begin to compute the regions on the roulette wheel which they will occupy.

The roulette wheel "wheel" is a dictionary with key values being 2-tuples containing two floats, regionS and regionE. These denote the start and end points of a region that is mapped to a population member. A for loop runs through the entire population and the regions are assigned to their respective population members using the formula ((members_fitness) * 360) / F. The formula determines the angle of the roulette wheel owned by that member. The regions themselves are back-to-back non-overlapping.

Once the regions have been determined, the algorithm begins to pick the parents X, Y by using random.uniform(0, 360) which tells where the wheel has landed. We then look through the dictionary in a for-loop and see in which range the wheel has landed. This happens in a while loop which keeps going on until both parents were picked, it is made sure that the parents are unique.

The next step is to create children using the "breed" function. This function takes X, Y, the distMat (same as always), inp_size which is the size of the tours and p_mutate. "breed" wraps around two more functions, namely inversion and genChild. These two functions are detailed by (Üçoluk, n.d.) and my algorithms implement the pseudocode displayed there.

Inversion takes the array containing the tour of a parent and creates another array which acts as the chromosome data of that parent. The chromosome data is acquired by iterating through each city of the parent in a for-loop and comparing all previous cities in that array using a nested while loop. Once this has been done for both parents, a standard 1-opt crossover is done around a random index. In my case this value can take the 2nd city to the 3rd to last city. The reason for this is because the last city is simply a repeat of the first but also because splitting from the 1st city or 2nd last city would simply yield the parents themselves. Using this "split" index, we can separate the chromosomes into their prefixes and suffixes and combine prefix of one parent, with the suffix of another to acquire two sets of chromosomes for the children.

The function "genChild" simply takes an array with the chromosomes and an array of size inp_size (I chose a parent since it doesn't matter and I can save on memory and time) and uses the chromosome information to generate a valid child tour in that second input array. I used two nested for-loops and a temporary array pos to store the intermediate results. Finally, each child has a p_mutate chance to be mutated by the function mutate.

The mutate function takes an array which corresponds to a tour and swaps two randomly selected cities. It does this by randomly selecting two distinct indices (such that the last city cannot be selected as it is the same as the 1st city). A while loop is used on the second index to make sure it is different. A standard swap is made using a temporary variable and the cities in those indices are swapped. If any of the indices point to the 1st city, then the last one is fixed accordingly.

After the children have been made, they are both added in a small for-loop. Their fitness and length are computed and they are added to newP, the new population. At the end of the big while loop, we always check whether or not the size of newP has reached Pop_size, if no, we continue and select another pair to breed. If yes, we update P and F to their new values and increment the gen counter.

Eventually, the while loop will fall through and we will simply be left to find the member in the final population. This is done using a for loop at the end of the algorithm, we simply compare the tourlengths of each solution and update “bestTour” and “minCost” accordingly. The algorithm returns these two values and terminates.

Best Results:

Toursize(n)	12	17	21	26	42	48	58	175	180	535
Tourlength	56	2210	4758	2208	2515	37543	93832	46635	795480	150975

Whenever “results” are mentioned below they are referring to tourlength in the table.

Experimentation:

The first parameter that I changed around was p_mutate. For all my tests I used pop_size = 10 and gen_count = 300 with n = 12. I first tried the extreme value of 1, this meant that every child would be mutated. It is always hard to quantify results due to the randomness of GA but after 20 or so tests I found that most of the time the results would be around 170 – 200. I then halved p_mutate and after around 20 tests or so found that most results were around 150 – 190 with more results landing at 160. I took it further and made p_mutate = 0.1 which is when the biggest improvement was. The majority results were now around 120 – 160 with some results being at 100 – 110. I kept reducing the value until around 0.008 at which point results had started to dip below 100. These results occurred because the higher the value of p_mutate, the more akin to a random search that GA becomes, since the mutation algorithm swaps at random.

The second parameter was pop_size. Keeping p_mutate at 0.008 and gen_count at 100 I tested the algorithm for values of pop_size between 10 and 100. For n = 12 at 100 the algorithm was able to find the optimal 56 reasonably consistently however the execution time started to become quite long. At the lower end (10), the algorithm ran much faster but the results would hang around 130 – 160, never getting below 100. In the end I elected to change population size depending on the input size, in order to manage execution time with solution quality.

The final parameter that was changed was the gen_count. Pop_size, n and p_mutate were kept at 10, 12 and 0.008 respectively. The values that were tested ranged from 100 to 500. At 500 most results would lie in the range 90 – 130 and at 250 the results were around 100 – 140. For 100 the results ranged from 120 – 200. The reason why the higher values gave similar results is most likely due to the small pop_size, the lack of genetic material meant that they got stuck in local minima. When the pop_size was increased to 30 the range of results became 60 – 100 for both 250 and 500. As such, it can be concluded that the gen_count is only useful as a way of telling the algorithm how much exploring it should do.

I also tinkered around with the breeding function. Before the inversion algorithm I had first implemented a simple 1-opt crossover function where a random split index was chosen to split the parents to their prefixes and suffixes. However a part of this function required for the children to be repaired because of duplicate cities appearing in the children. This took quite a lot of execution time so I chose to implement the inversion method which as described, infers genetic information from the parents and builds the children using that information as opposed to directly using the parents. Although hard to quantify, the new method helped reduce execution time a little, although not as much as I had hoped.

References

Üçoluk, G., n.d. *Genetic Algorithm Solution of the TSP Avoiding Special Crossover and Mutation,* Ankara: Middle East Technical University.