Austin Youngren

CS 320-001

Lab 8: Genetic Algorithm for TSP

4/19/2024

Algorithm Issues & Approach

For the creation of my genetic algorithm for TSP I implemented five subroutines to aid in code cleanliness, bug detection, readability, and to reduce repetitive code. These subroutines are check_cases(), get_minimum(), parent_selection(), reproduction(), and get_next_gen(). When starting this assignment, I found myself overwhelmed due to the additional files provided. This made the start of the assignment one of the more difficult parts of my implementation process. So, I decided to create the simpler part of the assignment, which was developing a function that checked if the algorithm input parameters were valid, leading to the creation of check_cases().

Afterwards, I took the time to play with the files given to get a better understanding of the population data structure, followed by studying, and heavily utilizing, the lecture slides from both lectures for brainstorming possibilities. Give this information, I decided to pursue and utilize the tournament selection method, discussed in Dr. Partridge's lecture, for parent selection.

As I approached the need to creating a function for crossover, I became overwhelmed once again, partially due to indecision in implementing Professor Whitley's crossover method, or the method discussed in the previously mentioned lecture slides. To help in deciding, I attempted to implement both to see which one was more intuitive for me; Dr Partridge's explanation was, to my surprise, the more intuitive option.

When I began to receive output, after working through many errors, my code was running extremely slow once I hit generation 20, which did not seem right. By debugging with print statements, I found that my population was almost doubling for each new generation. This was startling for me because I took measures to ensure my population size did not stray too far from the initial population size through each new generation. The nature of the issue showed me that the bug lived in the reproduction() subroutine. After minimal searching, I found that I had added the reproductive parents to the next generation list in two separate areas of the function, which was an easy fix. Though the issue was fixed, this situation instigated a paranoid concern for the algorithm run time needed to pass the Zybook grader. Due to this concern, I took steps to reduce the complexity of my implementation by integrating a dictionary to track each path's calculated cost. This ensured that all possible paths were only calculated once and greatly enhanced the speed of my implementation.

Algorithm Implementation

When executing my implementation, ga_tsp() starts by calling the subroutine check_cases(). check_cases() takes the same input parameters as the core function of the algorithm, ga_tsp(), which are the variables 'initial_population', 'distances', and 'generations.' The function checks to see if any of these variables hold the value None and checks to see if 'generations' is less than or equal to 0. If any of these conditions hold true, the function will return the Boolean value of True which signals ga_tsp() to return the value of None to the caller. Otherwise, check_cases() will return the Boolean value of False, which allows ga_tsp() to continue the execution of the algorithm. The idea behind this subroutine was to reduce the amount of code, for readability, in ga_tsp().

After the check, within ga_tsp(), to avoid the possibility of future bugs caused by accidental changes to the input parameter, and to reduce the variable name's text space, I assign a deep copy of initial_population to the variable 'population.' I then create an empty dictionary for storing the costs of paths to reduce the amount of times the cost function is called in the subroutine get_minimum() before entering the for-loop. This loop iterates for the number of generations specified by the parameter 'generations.' For each iteration, the subroutine parent_selection(), which uses tournament selection for choosing which parents that will be used for creation of the next generation, is called.

The parent_selection() subroutine, takes the parameters ‘population,’ ‘distances,’ and calculated_cost. In this function, I create the variable ‘k’, that represents the number of randomly sampled parents taken from the population; in this case, I use a sample value of five. Then, the variable, num_parents, is assigned a value to represent the number of parents needed from the current population for child creation. Here, this number will always be half the population size. The following conditional ensures this number is always even, ensuring that each parent has a partner during the creation of a new generation.

Before entering the while-loop, an empty list named ‘parents’ is created to store the winners of the tournament selection of parents in the population. The while-loop iterates until the list, ‘parents,’ holds the number of winners equal to that of num_parents. For each iteration, the algorithm randomly samples 5 contenders from the population, then sends them to the subroutine get_minimum(), along with the ‘distances’ and ‘calculated_cost’ variables. get_minimum() then returns the contender with the shortest path and assigns it to the variable ‘winner.’ ‘winner’ is then appended to ‘parents’. Once enough winners are collected to meet the while-loop exit condition, the list ‘parents’ is returned to ga_tsp().

The get_minimum() subroutine mainly was created to reduce repeated code and ensure code cleanliness. It takes 3 input parameters, ‘collection’, ‘distance’, and calculated cost. The parameter, ‘collection,’ represents any form of population (contenders or population), allowing the subroutine to find the minimum path of any list that can associate to the ‘distance’ list variable. The subroutine iterates through every path in a collection and calculates the path’s cost, via the provided cost function. Each path’s cost calculation is stored in the dictionary, calculated_cost, where a path serves as the key to its associated cost value. This method of storage serves to reduce computational complexity by allowing the algorithm to only calculate the cost of each possible path in a population, across all generations, at most once. Each cost is then compared to the variable, min_cost, which serves to track the minimal path across a collection. Once every path has been analyzed, the algorithm exits the for-loop and returns the smallest path, min_path, to the caller function.

Once the algorithm has returned to ga_tsp() from parent_selection(), it immediately passes the returned parents to the subroutine get_next_gen(). This subroutine’s only parameter is a list parents retrieved from parent_selection(). In this subroutine, I create an empty list named next_gen and then assign the number of parents to the variable reproduction_rate before entering a for-loop. The for-loop will iterate for every pair of parents in the ‘parents’ list. In a single iteration, the pair of parents are passed to the crossover function, reproduction(), which returns two children. Both pairs of children and parents are then added the next_gen list. Once the loop has finished this process for every pair of parents, the subroutine returns next_gen to ga_tsp().

The subroutine, reproduction(), takes two parents as parameters and assumes that both parents are of the same length. The length of a single parent is assigned to the variable ‘n,’ which is used to randomly generate a start and end index to determine which paths will be passed down to the corresponding child (i.e., parent1 to child1, parent2 to child 2). Before this inheritance takes place, two children of length ‘n’ are created, where each element/gene holds the value -1. Then, using the randomly generated indices, we transfer the genes of the respective parent, from indices ‘start’ to ‘end,’ to the children. Afterwards, the child indices, c1_i & c2_i, are assigned to the element that follows the index, ‘end’. The ‘% n’ ensures that this will be the starting index of the child if ‘end’ is the last element. The for-loops are then utilized to place the remaining unused genes of the opposite parent (i.e., parent2 to child1, parent1 to child 2) in all element spaces that hold the value -1. Both children are then returned to the subroutine next_gen() as tuples.

When returning to ga_tsp() from get_next_gen(), the next generation is assigned the current population. These two lines could be condensed into one, but I left them as is to help with code readability. The process of parent selection and reproduction will repeat for each generation for the specified number of generations before exiting the loop. Once exited, we pass the current population to get_minimum() in order to find the best current path. The returned path is then assigned to the variable best_path, which is then returned to the algorithm’s caller function.