A Heuristic Approach and Analysis of The Travelling Salesman Problem

Tyler Conger, CS 570, tconger1@crimson.ua.edu

I. Abstract

***Abstract*—In this paper, I will be covering and diving into the travelling salesman problem both in understanding the purposes and use cases of the travelling salesman problem as it comes to real world uses as well as understanding the problem more completely. The travelling salesman problem is a classic problem that exists within the field of computer science. It has a large degree of relevance within the field of P vs NP problems (problems solvable in polynomial time versus nondeterministic polynomial time). Understanding the travelling salesman problem is important to understanding the field of P vs NP as well as variations of the NP problems. As the travelling salesman problem is considered unsolvable, a variety of algorithms will be implemented including a brute force algorithm that will attempt to check every possible solution, a commonly used approach in the nearest neighbor algorithm, and finally a new implementation approach that represents a heuristic approach to the problem with the goal of finding faster and more accurate solutions to the problem. The overall goal of this self-described heuristic algorithm is to compete within the class period in an attempt to discover the various positives and negatives of different approaches and techniques at solving the problem and finding solutions.**

II. Introduction

First the introduction of the problem. This travelling salesman problem has become a very important question to researchers as it can be applied to many different sciences and areas of study. It originally was introduced by mathematicians in the 1800s. As can be seen this is a very old problem that has seen many iterations of attempts to solve and discover various techniques used within the way of solving the problem. In the classic explanation of the problem a salesman is given a list of various cities across the country that he must visit, the goal of this salesman is to visit each of these cities only one time and to return to his original starting city all while making sure to take the shortest route between them as to save as much time and money as possible.

This problem is an extremely difficult problem to reach a true solution to because as each destination can be visited from any other point within the graph the number of potential routes is extremely high and grows extremely quickly as the number of cities or destinations grows. The large growth of this problem makes it extremely difficult to solve and confirm that the absolutely correct solution has been reached. Because we usually cannot compute every single possible path it is difficult to know if we have selected the ’correct’ path, even if we found a very efficient path a potentially more efficient one exists.

This makes this problem be described as one that is NP-Hard. This means that this problem is difficult to find a complete solution to. The NP stands for nondeterministic polynomial time, meaning that it cannot be determined how long a solution would take to create. The hard means that it is equally as difficult as all other problems within the NP-Hard space, as such if an algorithm is found to solve the travelling salesman problem, then there would exist and algorithm to solve all other problems within the NP-Hard space of problems. In the figure below we can see what a solution to the travelling salesman problem may look like. A scattering of nodes with edges drawn between them representing the path that was taken between each individual node.

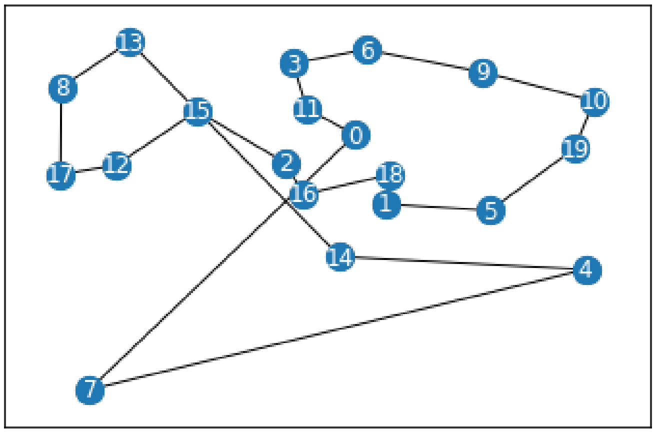


Figure 1. A common graph showcasing the travelling salesman problem.

As can be seen in the figure above. Each of the vertices in the image is representative of a city that must be visited. And the edges are representative of the path that is taken between each city or node. Even in this simple iteration it may be seen some simple ways that distance could be improved, and each city could still be reached. It is important to note in the traveling salesman problem each of the nodes are fully connected, meaning that each node is connected to every other node in the graph. This means that traversal between any two nodes within the graph is possible. However, this requirement of the graph to connect with every node allows for the various large crosses of the graph appear where large distances are traveled to reach a singular point, like reaching point #7, show potential improvements. As such this contributes to the extreme difficulty to developer an algorithm that is able to solve the travelling salesman problem with a great deal of efficiency.

III. Setup

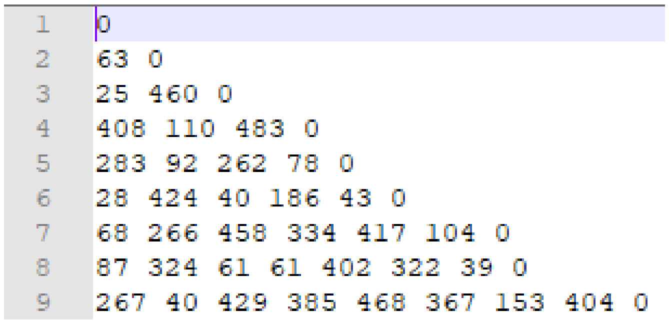
Before creating and solving any iterations of the travelling salesman problem we must first build some simple helper functions that will allow us to save solutions, create graphs, save graphs, display graphs, and read incoming graph files. These pieces will be important basics that will be used by each of the various functions.

The first of these is the creation of graphs. Depending on the outlook of the problem there are two ways to look at the different graphs that are created. The first is considering each edge of the graph as a ‘cost’ and not necessarily a distance. The best way to describe this is the distance between Phoenix and Los Angels is static, but the time versus cost of each way to travel between the two differs. Consider three different methods first being driving which may take 7 or so hours, the second being taking a train which may take 8-9 and the third being a flight between the two cities which may take 3 hours. Now each of these solutions involves a different ‘cost’ as the time cost is different on each as well as the material cost. A plane ticket may be much more expensive then renting a car and driving which may be more expensive than taking an Amtrak train between the two. As such this cost per time should be considered in our graph. As such each edge is not representative of a distance between two points but a cost to travel between them. This is the original methodology that will be followed when creating graphical representations of the cities.

The next thought process in terms of graph creation is the opposite, illustrating only the distance between the two cities and determing the distance potentially in number of pixels or otherwise between the two points and calcuating the eculidian distance between the two. This may be more representative of a situation in which traveling by a consistent means is necessary with less adaptation to outside and various problems that also exsist within the space of the travelling salesman problem.

For consideration of all created algorithms they will work on either methodology of graph creation will work accordingly. For my testing purposes I believe creation of the more realistic graph with more randomized weights to be better and will build with that in mind.

The first step is to create a graph is determing how large the graph should be, problems of up to around 30,000 points should be solvable with our created algorithms. Each point is connected to every other point and has a weight determined by the cost to travel between them. As the graph is generated a diagonal matrix is formed showing the distance between each point. This can be seen in the below figure numbered Figure 2. While a square matrix is acceptable a triangular matrix works the same as both the top and bottom of the triangular matrix will be mirrored across the diagonal. For the use cases in this project we will be using the triangle matrix associated with the fully connected graph, instead of the square matrix, this is because the triangular matrix possess the same information as the square matrix and can be easily converted between the two. As such either is sufficient for solving the travelling salesman problem the triangular will be used.

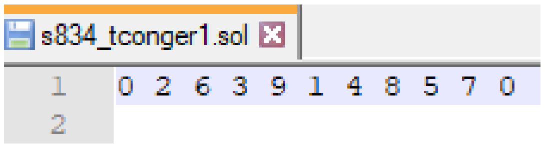


```
0
63 0
25 460 0
408 110 483 0
283 92 262 78 0
28 424 40 186 43 0
68 266 458 334 417 104 0
87 324 61 61 402 322 39 0
267 40 429 385 468 367 153 404 0
```

Figure 2. Textual representation of a fully connected graph generated by the generate_graph function.

As can be seen when calling the function generate_and_write_graph function and passing in the size a graph is generated and then written to a file. The file naming convention follows Size<distance_travelled>.graph . Where the distance_travelled is the length of the path that was taken in finding the solution. Now that we have the ability to generate graphs of any size we can begin creating our early methods of deriving solutions to the travelling salesman problem.

It is also important to understand how a solution is formated for consistance across the board. Solution files are names s[distance_traveled]_<user_ID>.sol . As such my solutions would look something like s4535_tconger1.sol . Where in this case 4535 is the distance of the cycle or path that was found. Then the contents of the file are the order the nodes appear in followed by a space, appearing as in the figure seen below.



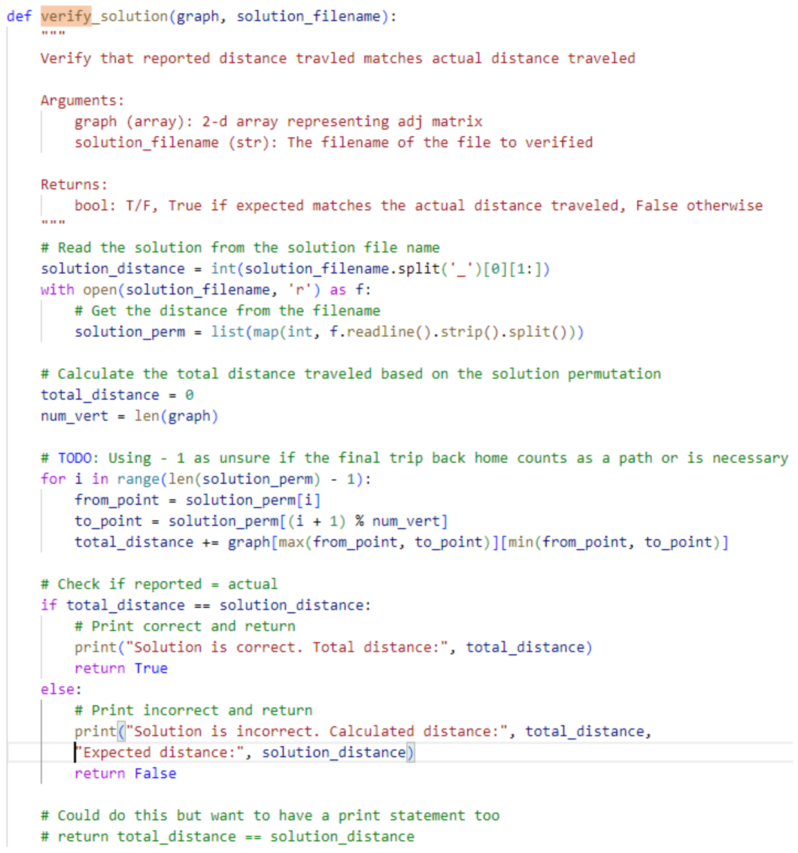
s834_tconger1.sol

```
0 2 6 3 9 1 4 8 5 7 0
```

Figure 3. The solution file showcasing node order and file name

It is important to see that each node is visited one time, with the exception of the first node and that is shown at both the beginning and end of the file, as the cycle must be completed. As can be seen this graph was of size ten, with nodes that are numbered zero through nine.

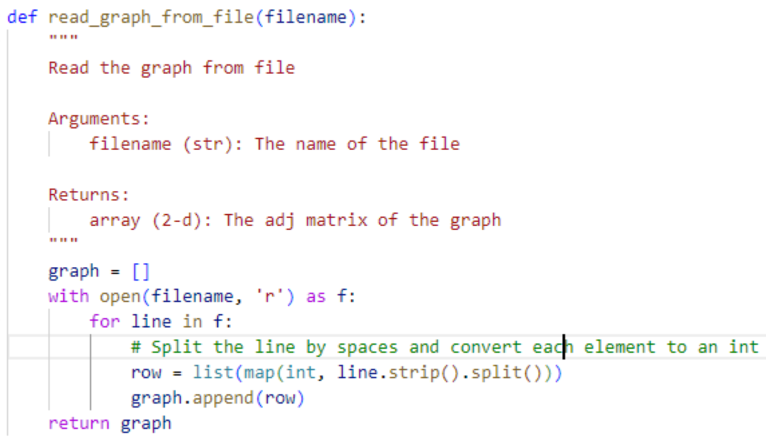
Finally it is necessary to have a function that will verify these solutions we have created, verifying that the solution is correct and the actual distance of the path matches the reported distance of the path. The implementation of this function can be seen in the figure below.



```python
def verify_solution(graph, solution_filename):
    """
    Verify that reported distance travled matches actual distance traveled

    Arguments:
        graph (array): 2-d array representing adj matrix
        solution_filename (str): The filename of the file to verified

    Returns:
        bool: T/F, True if expected matches the actual distance traveled, False otherwise
    """
    # Read the solution from the solution file name
    solution_distance = int(solution_filename.split('_')[0][1:])
    with open(solution_filename, 'r') as f:
        # Get the distance from the filename
        solution_perm = list(map(int, f.readline().strip().split()))

    # Calculate the total distance traveled based on the solution permutation
    total_distance = 0
    num_vert = len(graph)

    # TODO: Using - 1 as unsure if the final trip back home counts as a path or is necessary
    for i in range(len(solution_perm) - 1):
        from_point = solution_perm[i]
        to_point = solution_perm[(i + 1) % num_vert]
        total_distance += graph[max(from_point, to_point)][min(from_point, to_point)]

    # Check if reported = actual
    if total_distance == solution_distance:
        # Print correct and return
        print("Solution is correct. Total distance:", total_distance)
        return True
    else:
        # Print incorrect and return
        print("Solution is incorrect. Calculated distance:", total_distance,
        "Expected distance:", solution_distance)
        return False

    # Could do this but want to have a print statement too
    # return total_distance == solution_distance
```

Figure 4. Function for verifying solve files that the reported distance is the actual distance travelled.

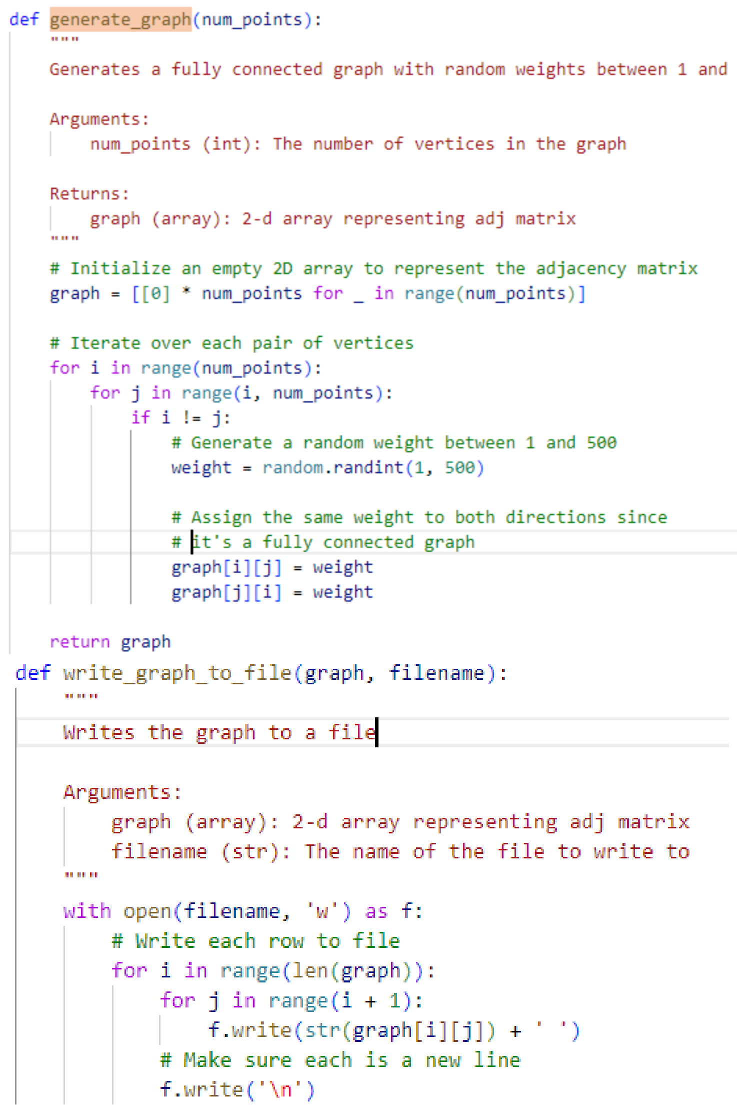
This function works to verify that the solution is correct, by taking the reported distance in the file name and retracing the distance travelled to verify that the distance travelled is the same as the reported distance. Within this setup step it is also required to have functions to read the graph and save it to a variable, the code of which can be seen in the figure below.



```python
def read_graph_from_file(filename):
    """
    Read the graph from file

    Arguments:
        filename (str): The name of the file

    Returns:
        array (2-d): The adj matrix of the graph
    """
    graph = []
    with open(filename, 'r') as f:
        for line in f:
            # Split the line by spaces and convert each element to an int
            row = list(map(int, line.strip().split()))
            graph.append(row)
    return graph
```

Figure 5. Function for reading the graph from a file and returning the graph.

As can be seen this function quickly reads the file and creates the two-dimensional array that is used to define a graph. This is especially useful for pre-generated graphs such as the ones that will be provided within scope of the competition. We will also need some other helper functions including an aforementioned generate_graph function to create graphs, and a function to write these generated graphs to file, which can be seen in the figure below.



```
def generate_graph(num_points):
    """
    Generates a fully connected graph with random weights between 1 and 500

    Arguments:
        num_points (int): The number of vertices in the graph

    Returns:
        graph (array): 2-d array representing adj matrix
    """
    # Initialize an empty 2D array to represent the adjacency matrix
    graph = [[0] * num_points for _ in range(num_points)]

    # Iterate over each pair of vertices
    for i in range(num_points):
        for j in range(i, num_points):
            if i != j:
                # Generate a random weight between 1 and 500
                weight = random.randint(1, 500)

                # Assign the same weight to both directions since
                # it's a fully connected graph
                graph[i][j] = weight
                graph[j][i] = weight

    return graph
def write_graph_to_file(graph, filename):
    """
    Writes the graph to a file

    Arguments:
        graph (array): 2-d array representing adj matrix
        filename (str): The name of the file to write to
    """
    with open(filename, 'w') as f:
        # Write each row to file
        for i in range(len(graph)):
            for j in range(i + 1):
                f.write(str(graph[i][j]) + ' ')
            # Make sure each is a new line
            f.write('\n')
```

Figure 6. Function generating graph and saving a graph to a file

As can be seen these functions are necessary to support all necessary functionality and are widely used throughout the rest of the TSP implementation, and will be necessary for ability to compete within the class competition. It is important to understand the guidelines to this project as well as the associated competition guidelines.

IV. Competition Guidelines

Before outlining the various algorithms that will be implemented it is important to understand the limitations and expectations of the competition portion. For the competition we are expected to create a heuristic algorithm to solve the TSP. The goal of this heuristic will be to find the shortest path that satisfies the requirements of the TSP on a few various graph sizes. This will have to be done within the allotted time frame of the class period, which is 50 minutes, as such having an algorithm that is concise enough to run within a short time frame will be requisite. It's expected that some of the class period will be used for setup and submission, so the algorithm should be able to complete all of the graphs within a time frame of about 30-minutes.

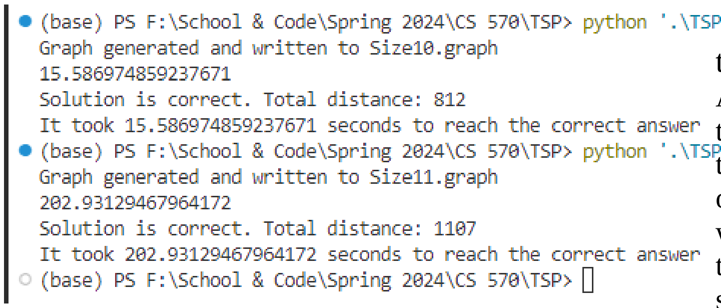
The heuristic should be able to solve 5 different graph sizes. One of size 250, 1,000, 5,000, 15,000, and 30,000. As each of these graph sizes will need to be solved it is both important to find a faster path as well as being able to complete each graph size in the allotted time. As such the algorithms that are used should focus on both the size of the graph and the speed at which computation is possible.

For this competition I plan on using Python as my selected language, while this is a "slower" language than other languages like C or Assembly. I believe that using it in this situation will be sufficient, especially considering the code will already be running on my slower laptop as such competing with faster devices will be difficult even in a more suited language like C. Also, I am much more familiar with Python as a language along with the libraries and resources available in Python allowing these to be used in solving the problem as well. Overall, for this task Python is a fair choice of language for implementation of the three separate solutions along with each of the various helper functions that are necessary.

V. Brute Force Solution

The first solution created is a brute force solution. This means that this attempt will find all possible solutions to the problem and then save the shortest path. While this sounds like the best way to solve this problem, as the problem expands so quickly this method begins to take longer and longer until eventually taking longer than we would be able to run the program feasibly and thus becomes a poor at reaching the solution.

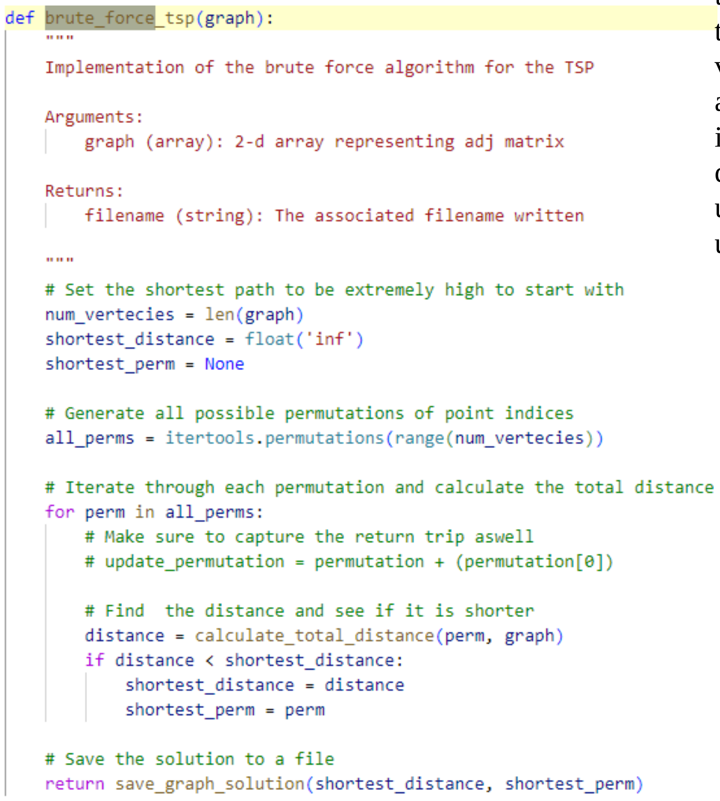
For this solution we first generate every permutation of the nodes order and calculate the distance of that path, and as we do that, we are able to find the smallest solution. Because we are trying so many solutions it takes a great deal of time to complete calculations. This can be seen in attached figure below.



```
● (base) PS F:\School & Code\Spring 2024\CS 570\TSP> python '.\TSP Code.py'
  Graph generated and written to Size10.graph
  15.586974859237671
  Solution is correct. Total distance: 812
  It took 15.586974859237671 seconds to reach the correct answer
● (base) PS F:\School & Code\Spring 2024\CS 570\TSP> python '.\TSP Code.py'
  Graph generated and written to Size11.graph
  202.93129467964172
  Solution is correct. Total distance: 1107
  It took 202.93129467964172 seconds to reach the correct answer
○ (base) PS F:\School & Code\Spring 2024\CS 570\TSP> 
```

Figure 7. Depicting run times for brute force method on graphs of size 10 and then a graph of size 11.

As can clearly be seen, the first graph takes 16 seconds to complete while the second one takes over 200 seconds, just from adding one additional node. As more and more nodes are added the time to completely brute force the graph increases exponentially. While on smaller graphs this may be a valid solution it quickly becomes too difficult and is no longer a viable way of solving the problem. Looking at the code of the brute force solution we can see that all possible permutations of the list are created, creating a very large number of possible paths that need to be traced. Then each of these potential permutations are iterated through, with the distance calculated on each individual path.



```python
def brute_force_tsp(graph):
    """
    Implementation of the brute force algorithm for the TSP

    Arguments:
        graph (array): 2-d array representing adj matrix

    Returns:
        filename (string): The associated filename written

    """
    # Set the shortest path to be extremely high to start with
    num_vertecies = len(graph)
    shortest_distance = float('inf')
    shortest_perm = None

    # Generate all possible permutations of point indices
    all_perms = itertools.permutations(range(num_vertecies))

    # Iterate through each permutation and calculate the total distance
    for perm in all_perms:
        # Make sure to capture the return trip aswell
        # update_permutation = permutation + (permutation[0])

        # Find  the distance and see if it is shorter
        distance = calculate_total_distance(perm, graph)
        if distance < shortest_distance:
            shortest_distance = distance
            shortest_perm = perm

    # Save the solution to a file
    return save_graph_solution(shortest_distance, shortest_perm)
```

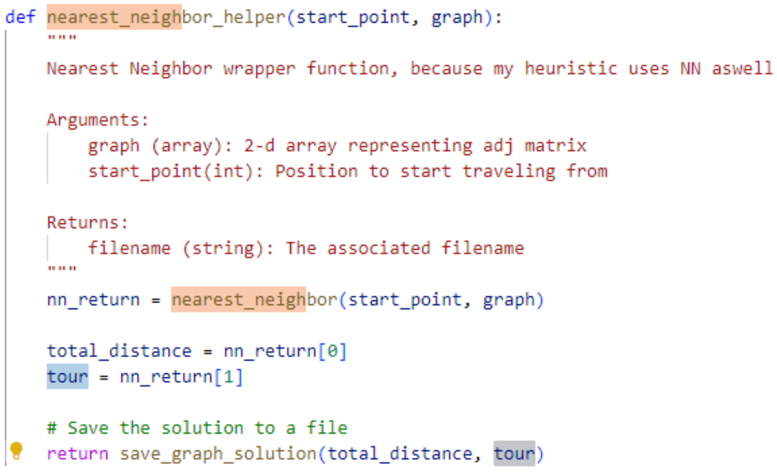
Figure 8. The code for the brute force algorithm implemented in Python

This results in a $O(N^2)$ runtime due to the loop and the runtime of calculate_total_distance being O(N). As this algorithm takes a long time to run it is important that we look to other more heuristic methods of solving the problem. Even while this code could have potential optimization points, like stopping the calculation once we realize that it will not be the smallest or by saving the totals of certain branches or sections of the tree, this solution will still continue to struggle especially as graph sizes grow quickly. Because of this continued exponential growth using a more heuristic approach is necessary, even if this approach may give us a non-perfect answer.

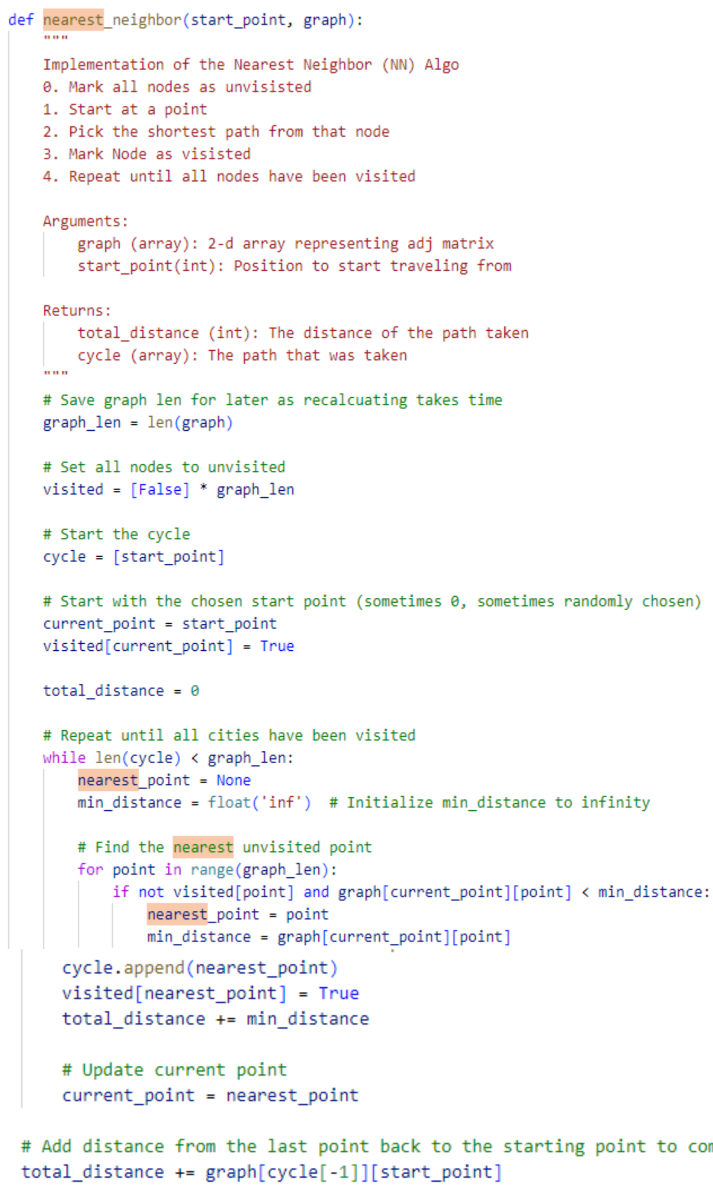
VI. Nearest Neighbor Method

The Nearest Neighbor algorithm is an algorithm that was developed to reach a solution to the TSP. The process by which the nearest neighbor algorithm works is by starting at a random node, and traveling the closest unvisited node until every node has been visited. It can be summarized in these steps.

First, chose a node randomly, and mark all other nodes as unvisited. Find the nearest node to this node that has not been visited. Travel to the newly selected node. After traveling to the new node, mark it as visited. Then repeat these steps until every single node as been marked as visited and no nodes remain as unvisited. This algorithmic approach to solving the problem can be seen in the figure below where the associated code is displayed. A helper function was also created that is used to return as well as give a start position, this will be useful later when the heuristic method is created.



```python
def nearest_neighbor_helper(start_point, graph):
    """
    Nearest Neighbor wrapper function, because my heuristic uses NN aswell

    Arguments:
        graph (array): 2-d array representing adj matrix
        start_point(int): Position to start traveling from

    Returns:
        filename (string): The associated filename
    """
    nn_return = nearest_neighbor(start_point, graph)

    total_distance = nn_return[0]
    tour = nn_return[1]

    # Save the solution to a file
    return save_graph_solution(total_distance, tour)
```



```python
def nearest_neighbor(start_point, graph):
    """
    Implementation of the Nearest Neighbor (NN) Algo
    0. Mark all nodes as unvisisted
    1. Start at a point
    2. Pick the shortest path from that node
    3. Mark Node as visisted
    4. Repeat until all nodes have been visited

    Arguments:
        graph (array): 2-d array representing adj matrix
        start_point(int): Position to start traveling from

    Returns:
        total_distance (int): The distance of the path taken
        cycle (array): The path that was taken
    """
    # Save graph len for later as recalcuating takes time
    graph_len = len(graph)

    # Set all nodes to unvisited
    visited = [False] * graph_len

    # Start the cycle
    cycle = [start_point]

    # Start with the chosen start point (sometimes 0, sometimes randomly chosen)
    current_point = start_point
    visited[current_point] = True

    total_distance = 0

    # Repeat until all cities have been visited
    while len(cycle) < graph_len:
        nearest_point = None
        min_distance = float('inf')  # Initialize min_distance to infinity

        # Find the nearest unvisited point
        for point in range(graph_len):
            if not visited[point] and graph[current_point][point] < min_distance:
                nearest_point = point
                min_distance = graph[current_point][point]
        cycle.append(nearest_point)
        visited[nearest_point] = True
        total_distance += min_distance

        # Update current point
        current_point = nearest_point

    # Add distance from the last point back to the starting point to complete the trip
    total_distance += graph[cycle[-1]][start_point]
```

Figure 9. The code implementation of the nearest neighbor algorithm.

As can be seen, the nearest neighbor algorithm works in O(N^2) time allowing a heuristic solution to the problem very quickly. However, it may struggle to find a perfect solution. Consider a situation in which all nearby nodes have already been visited and the only node left to visit is across the graph, while this may not be the most optimal path it is the one that nearest neighbor would take causing a large edge of the graph to be added where taking that node earlier would in fact have been more optimal. This methodology to finding a shortest path is better than the simple brute force methodology as that it works much quicker which is important for larger graphs or less computational power, but it may not find the actual shortest path between the nodes. The starting node is also extremely important in the nearest neighbor method, as changing the starting node also would change what nodes are nearest, changing the entire potential path of the algorithm.

As this implementation of the nearest neighbor does run very fast by comparison to the brute force, especially on large graph sizes, it does not always provide the absolute best solutions. Due to the potential to take paths that are not optimal, such as taking a final edge across the entirety of the graph. Because of this nearest neighbor can produce good results but will produce different results based on the starting node chosen, as the path selected will be different. Due to this the creation of a new heuristic method is necessary in solving the problem both quickly and deriving and finding an optimal path.

VII. Self-Created Heuristic

For creating our own heuristic to solve the TSP, first it is important to consider a few potential directions that could be taken to improve on solving the problem.

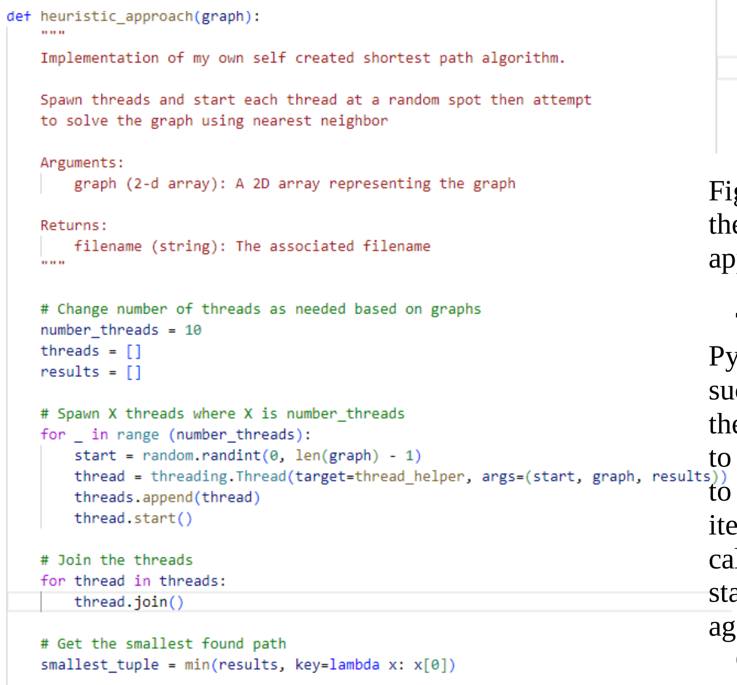
One initial thought I had at a potential algorithm was to divide the graph up into sections, each section would be solved using a nearest neighbor method and then stitched together to form the entire cycle. This way a singular nearest neighbor would not branch the entire length of the graph taking large, disadvantaged paths and instead the largest taken path would be in a singular section. This methodology would be quick but would involve finding the relative location of each group of points to divide them up which may be difficult and intensive to memory usage.

Another potential solution would be to use a genetic approach. At first creating a graph with a method like nearest neighbor, then mutating links selecting the best and continuing to mutate until better links have been selected. While this may be a very good way to find an approximate solution it may take a great deal of fine tuning which could be difficult in a competition setting where there is limited time to run the algorithm multiple times tuning parameters each time.

My thought was similar to the first approach of subdividing the graph and running nearest neighbor, but I quickly discovered I could reach gradually better solutions by running nearest neighbor from different nodes actually continued to yield better solutions. Then if I took a node two hops away from this initial node, I was able to find strong solutions very quickly by running the algorithm again from these nodes. Because of this speed my initial thought is to run an extremely quick algorithm like nearest neighbor many times in an effort find these ‘good’ starting nodes and use those to find shortest path. I think this would be a good method as it should be expandable to any size that the nearest neighbor is able to handle, allowing for handling of extremely large graphs very quickly, where some other types of algorithms would begin to slow at these larger graph sizes.

To quickly run each of these nearest neighbor algorithms in tandem, using threads seemed like a good solution. This would allow for me to utilize all of my computing power even on a smaller machine like my laptop.

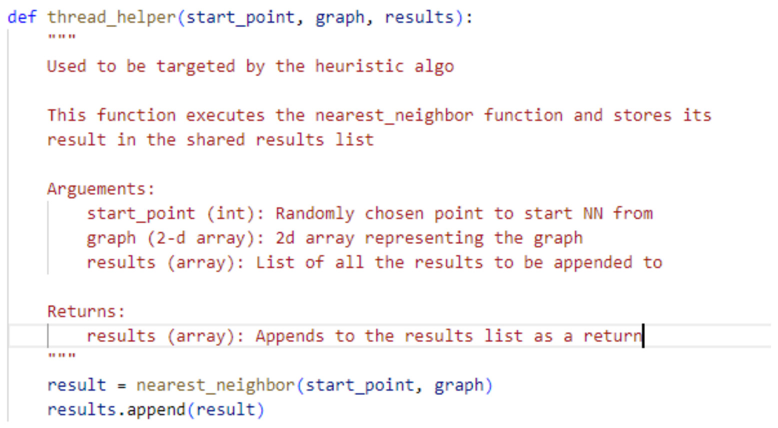
The first step is to run the nearest neighbor algorithm starting at a large variety of randomly selected threads this can be seen in the figure below.



```python
def heuristic_approach(graph):
    """
    Implementation of my own self created shortest path algorithm.

    Spawn threads and start each thread at a random spot then attempt
    to solve the graph using nearest neighbor

    Arguments:
        graph (2-d array): A 2D array representing the graph

    Returns:
        filename (string): The associated filename
    """

    # Change number of threads as needed based on graphs
    number_threads = 10
    threads = []
    results = []

    # Spawn X threads where X is number_threads
    for _ in range (number_threads):
        start = random.randint(0, len(graph) - 1)
        thread = threading.Thread(target=thread_helper, args=(start, graph, results))
        threads.append(thread)
        thread.start()

    # Join the threads
    for thread in threads:
        thread.join()

    # Get the smallest found path
    smallest_tuple = min(results, key=lambda x: x[0])
```

Figure 10. First half of the implementation of the heuristic approach using nearest neighbor algorithm with many threads at a time

As can be seen, the simple idea of spawning a large number of threads and running the nearest neighbor algorithm starting at each of these individual nodes allows for the first half of the code to be run very quickly. The initial starting nodes are chosen randomly to distribute them evenly throughout the graph and will later be refined upon. The number of threads that are used can be modified very easily depending on the size of the problem. I found that on my device using a larger number of threads on smaller problems yielded quick results with a good reduction over time in the length of the path found. In larger graph sizes the number of threads may need to be reduced depending on the computing resources available. The ability to run these threads in parallel allows for a large number of attempts to be run in tandem, which helps to yield quick and precise results. As can be seen the target of these threads is the thread_helper function which can be seen in the figure below, this function is simply used as a wrapper function to the nearest neighbor function as discussed previously.



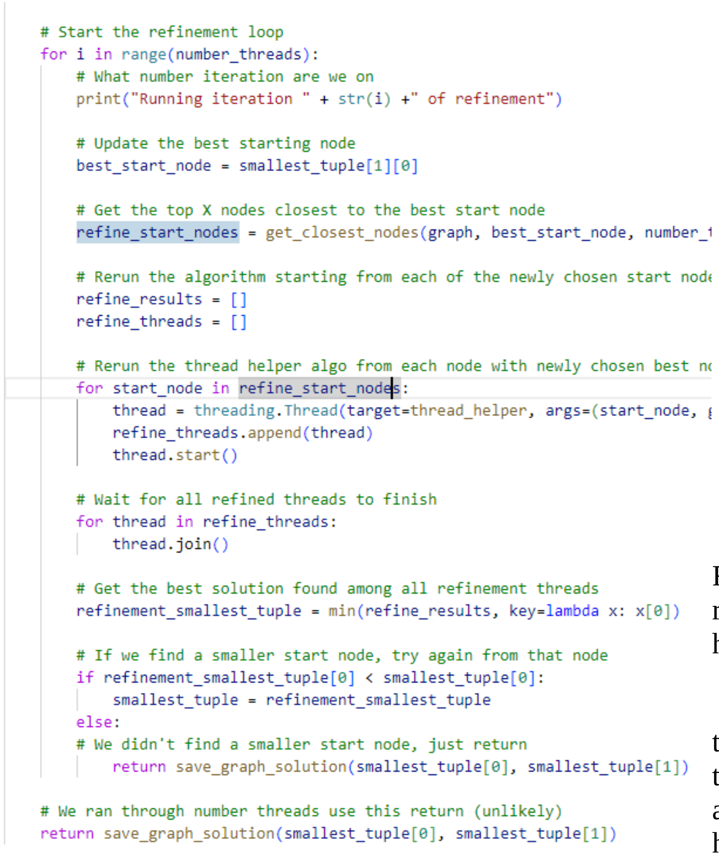
```python
def thread_helper(start_point, graph, results):
    """
    Used to be targeted by the heuristic algo

    This function executes the nearest_neighbor function and stores its
    result in the shared results list

    Arguements:
        start_point (int): Randomly chosen point to start NN from
        graph (2-d array): 2d array representing the graph
        results (array): List of all the results to be appended to

    Returns:
        results (array): Appends to the results list as a return
    """
    result = nearest_neighbor(start_point, graph)
    results.append(result)
```

Fig 11. Thread_helper function which is just used to call the nearest neighbor function and return the results by appending to the list

This function is necessary because the threading in Python needs a function to be targeted by the thread, as such this function just takes in the chosen place to start the nearest neighbor algorithm, the graph (to be passed to nearest neighbor function) and the results, to be added to with the information of the path found in this iteration. Using this targeted function allows for the calling of nearest neighbor function within each thread, starting from each individual start position, saving the aggregated results appropriately.

Once all the threads are created, they are added to an array of threads and then run. Then we join together all the threads together. This thread.join() allows for all the threads to run individually and then the results are saved. All the threads are run and joined together when the final thread is complete, so it may take extra time depending on the slowest thread used. Once all the threads have been run individually, the best result of these threads is taken. After gathering all the results, we find the best start node by getting the minimum distance travelled in the results array. This way we find the best node to start the next iteration of the process at. This can be seen in the continued snippet of the heuristic_approach function code in the block below. Once this new start node has been selected, all the nearest nodes to it are selected and the nearest neighbor algorithm is recalculated starting at each of these nearby nodes, in an effort to narrow down the best starting position for nearest neighbor.

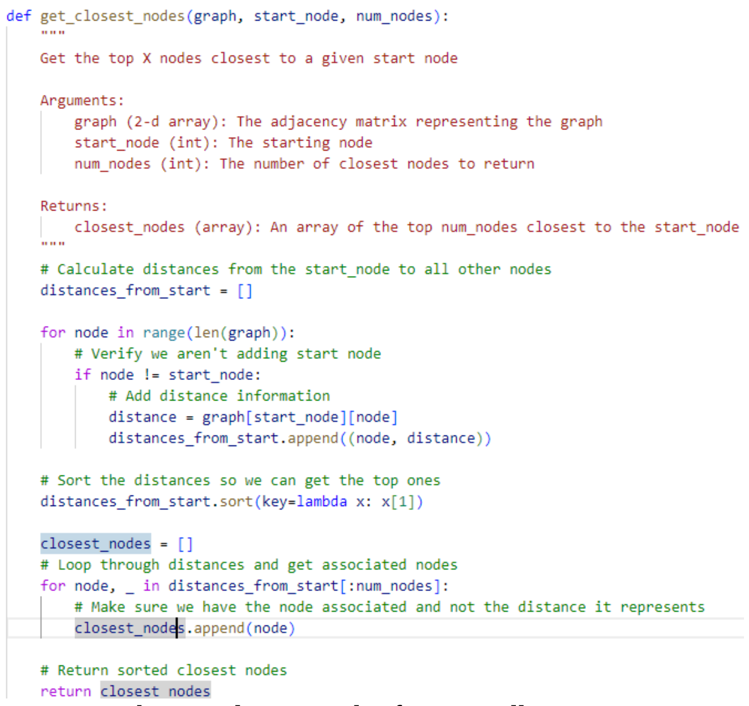
This process is continually repeated with each of these identified best nodes until eventually the code is unable to find a better start position and thus cannot improve any further, depending on the graph this may take only a few steps or many, but usually does not run up till the limit, which is based on the number of threads that were created.



```
# Start the refinement loop
for i in range(number_threads):
    # What number iteration are we on
    print("Running iteration " + str(i) +" of refinement")

    # Update the best starting node
    best_start_node = smallest_tuple[1][0]

    # Get the top X nodes closest to the best start node
    refine_start_nodes = get_closest_nodes(graph, best_start_node, number_t

    # Rerun the algorithm starting from each of the newly chosen start node
    refine_results = []
    refine_threads = []

    # Rerun the thread helper algo from each node with newly chosen best n
    for start_node in refine_start_nodes:
        thread = threading.Thread(target=thread_helper, args=(start_node, g
        refine_threads.append(thread)
        thread.start()

    # Wait for all refined threads to finish
    for thread in refine_threads:
        thread.join()

    # Get the best solution found among all refinement threads
    refinement_smallest_tuple = min(refine_results, key=lambda x: x[0])

    # If we find a smaller start node, try again from that node
    if refinement_smallest_tuple[0] < smallest_tuple[0]:
        smallest_tuple = refinement_smallest_tuple
    else:
    # We didn't find a smaller start node, just return
        return save_graph_solution(smallest_tuple[0], smallest_tuple[1])

# We ran through number threads use this return (unlikely)
return save_graph_solution(smallest_tuple[0], smallest_tuple[1])
```

Figure 12. Second half of the implementation of the heuristic approach, where after finding the best start position we start from nearby nodes in a refinement effort

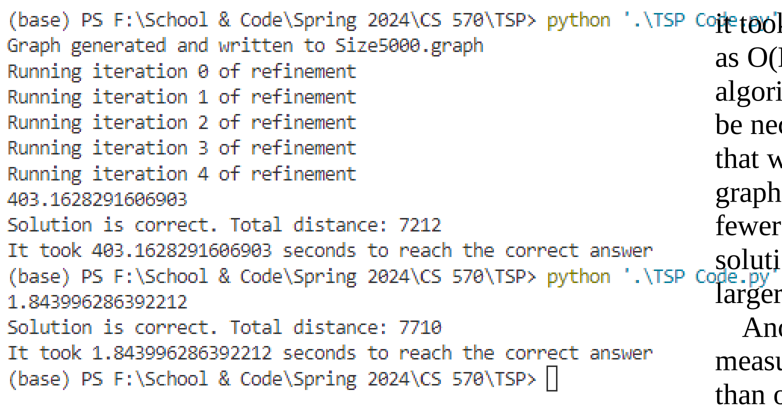
The above snippet of the code will run until near exhaustion, running for at max the same as the number of threads, but rarely actually reaching this maximal. Instead it will run a few times to find nearby nodes that are a better start location and attempt the same process at each of these newly selected nodes. The first step is to get the X closest nodes, where X is the number of threads we are using overall, as we will be running the algorithm again with the X threads. This selection of the closest nodes can be seen in the figure below. Getting the closest nodes is important as it allows us to find other potentially good starting nodes, which may be better, so finding these close nodes to the current good starting position helps to refine the search for a potential best starting node.



```
def get_closest_nodes(graph, start_node, num_nodes):
    """
    Get the top X nodes closest to a given start node

    Arguments:
        graph (2-d array): The adjacency matrix representing the graph
        start_node (int): The starting node
        num_nodes (int): The number of closest nodes to return

    Returns:
        closest_nodes (array): An array of the top num_nodes closest to the start_node
    """
    # Calculate distances from the start_node to all other nodes
    distances_from_start = []

    for node in range(len(graph)):
        # Verify we aren't adding start node
        if node != start_node:
            # Add distance information
            distance = graph[start_node][node]
            distances_from_start.append((node, distance))

    # Sort the distances so we can get the top ones
    distances_from_start.sort(key=lambda x: x[1])

    closest_nodes = []
    # Loop through distances and get associated nodes
    for node, _ in distances_from_start[:num_nodes]:
        # Make sure we have the node associated and not the distance it represents
        closest_nodes.append(node)

    # Return sorted closest nodes
    return closest_nodes
```

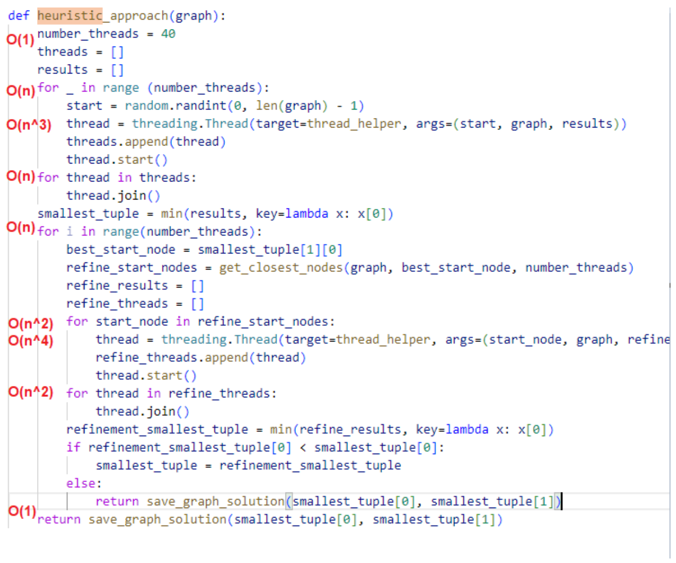
Figure 13. The get_closest_nodes function allowing retrieval of the X closest nodes to be used within the heurisitc approach algorithm.

Once the closest nodes are retrieved, we can create threads for each of these nodes and rerun the process on the newly selected nodes, running the nearest neighbor algorithm from each of these nodes. This is similar to how it was done during the first section of the code sample. Then after the threads we find the smallest path that was found in the refinement step. If it is smaller than the smallest_tuple the algorithm is rerun starting at this newly selected node, again creating threads, and finding the smallest cycle from these newly chosen best nodes. This all in an effort to find the best starting node location for the nearest neighbor algorithm and thus reducing the cycle taking to solve the TSP. After the loop finishes running the results are returned or are returned early if we fail to find a smaller cycle. This information is returned and saved in a file so that we are able to access it for later review. Now we must conduct some analysis on this methodology used to verify it actually yields improved solutions in a decent amount of time. This example runtime on a graph of size 5,000 is shown in the image below. Obviously, it would be expected that running the nearest neighbor algorithm many times would be slower than just a singular run of the algorithm. With each run of the refinement steps, we are running the nearest neighbor algorithm many times, equal to the number of threads, thus a slower run time is to be expected. However, due to the speed at which nearest neighbor algorithm runs it is expected to still be faster than other potential algorithms. The big O analysis of this function will yield an expected runtime of O(N^4), due to the double loop, for iterating over each identified start position, and N^2 from the nearest neighbor algorithm. This big O analysis can be seen in figure 15 below.



```
(base) PS F:\School & Code\Spring 2024\CS 570\TSP> python '.\TSP Code.py'
Graph generated and written to Size5000.graph
Running iteration 0 of refinement
Running iteration 1 of refinement
Running iteration 2 of refinement
Running iteration 3 of refinement
Running iteration 4 of refinement
403.1628291606903
Solution is correct. Total distance: 7212
It took 403.1628291606903 seconds to reach the correct answer
(base) PS F:\School & Code\Spring 2024\CS 570\TSP> python '.\TSP Code.py'
1.843996286392212
Solution is correct. Total distance: 7710
It took 1.843996286392212 seconds to reach the correct answer
(base) PS F:\School & Code\Spring 2024\CS 570\TSP> 
```

Figure 14. A run of the heuristic approach followed by a run of a traditional nearest neighbor algorithm on the same graph. Heuristic approach is run with X threads being 40



```
def heuristic_approach(graph):
O(1)    number_threads = 40
        threads = []
        results = []
O(n)    for _ in range (number_threads):
            start = random.randint(0, len(graph) - 1)
O(n^3)      thread = threading.Thread(target=thread_helper, args=(start, graph, results))
            threads.append(thread)
            thread.start()
O(n)    for thread in threads:
            thread.join()
        smallest_tuple = min(results, key=lambda x: x[0])
O(n)    for i in range(number_threads):
            best_start_node = smallest_tuple[1][0]
            refine_start_nodes = get_closest_nodes(graph, best_start_node, number_threads)
            refine_results = []
            refine_threads = []
O(n^2)      for start_node in refine_start_nodes:
O(n^4)          thread = threading.Thread(target=thread_helper, args=(start_node, graph, refine
                refine_threads.append(thread)
                thread.start()
O(n^2)      for thread in refine_threads:
                thread.join()
            refinement_smallest_tuple = min(refine_results, key=lambda x: x[0])
            if refinement_smallest_tuple[0] < smallest_tuple[0]:
                smallest_tuple = refinement_smallest_tuple
            else:
                return save_graph_solution(smallest_tuple[0], smallest_tuple[1])
O(1)    return save_graph_solution(smallest_tuple[0], smallest_tuple[1])
```

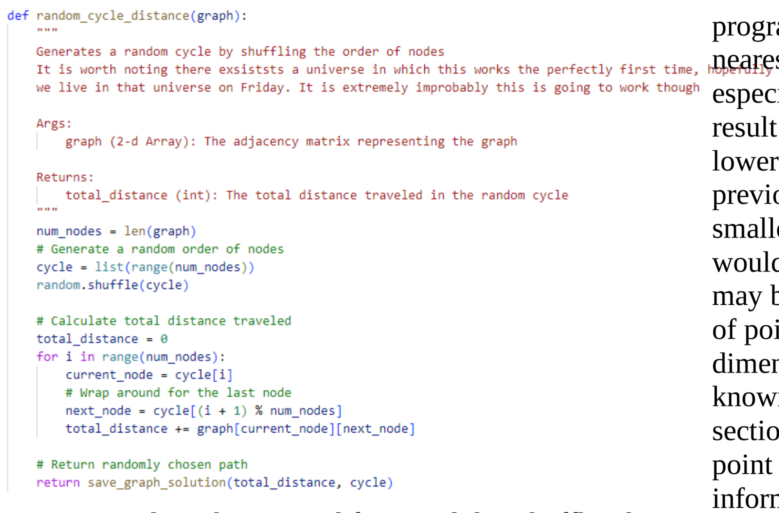
Figure 15. Big O analysis of the heuristic approach algorithm, comments and white space removed for concisenss.

As can be seen in figure 14, only 5 iterations of the refinement step were run, instead of the alloted 40 times based on the number of threads. In the process improving the total distance traveled by nearly 500, a fairly large improvement on a graph size of 5,000. While it did take a good bit of time longer to run the heuristic algorithm it was able to yield a better result and narrowed in on the best answer overtime, showing improvement each iteration. While running this solution does take a great deal of time more, where regular nearest neighbor algorithm took only 1.84 seconds to run the heuristic took nearly 6 and a half minutes. This runtime is entirely depending on the number of threads chosen. In this example, 40 threads were chosen, which means that throughout the process here the nearest neighbor algorithm was run over 240 times. Because of this high number of runs, it is understandable how long it took. We can also see this shown in the big O analysis, as $O(N^4)$ comparative to $O(N^2)$ of the nearest neighbor algorithm. However, on larger graph selections, it may be necessary to tune the number of threads created as that will largely affect the runtime. So with a larger graph like 10,000 it may be better to run with only 20 or fewer threads which will still help to reach a strong solution but reduce the time growth that will occur at larger graph sizes.

Another potential improvement would be to not measure the nearest neighbor for the same node more than once. Adding already searched nodes to a list and then skipping them within the solution would help to reduce the time, as it would be expected that running each iteration will yield different nodes, there may be some overlap causing delay as the same information is calculated multiple times. This improvement could be done by adding each searched node to a list and if an element from that is is to be searched skip over it. Even still the utilization of many threads helps to complete the tasks in parrallel and will change based on various hardware specifications of the system the code is being run on, so it is important for some tuning to take place to properly balance finding the solution and managing the time it takes to do so. This methodology of tuning the number of threads, and thus the iterations that are used will be necessary for the competition in larger graph sizes.

VIII. Discussions

After the completion of the competition, new insights were gleaned into how other individuals' algorithms may work and compare to my own. Unfortunately, my algorithms were not able to reach a more efficient solution, but it is still important to understand what potential improvements could be made at each step of the process. One unforeseen issue I ran into was the professor pre-filling the best scores with his own scores. This was an unexpected turn, as it was stated that speed would be an element and thus, I focused my algorithm on running fast in an effort to get on the board first even if it meant being passed later in the competition as a less efficient but more precise algorithm completed running. One piece I even developed is shown in the figure of the function below, a function derived to run extremely quickly but does provide less than desirable answers in terms of shortest path.



```python
def random_cycle_distance(graph):
    """
    Generates a random cycle by shuffling the order of nodes
    It is worth noting there exsiststs a universe in which this works the perfectly first time, hopefully
    we live in that universe on Friday. It is extremely improbably this is going to work though

    Args:
        graph (2-d Array): The adjacency matrix representing the graph

    Returns:
        total_distance (int): The total distance traveled in the random cycle
    """
    num_nodes = len(graph)
    # Generate a random order of nodes
    cycle = list(range(num_nodes))
    random.shuffle(cycle)

    # Calculate total distance traveled
    total_distance = 0
    for i in range(num_nodes):
        current_node = cycle[i]
        # Wrap around for the last node
        next_node = cycle[(i + 1) % num_nodes]
        total_distance += graph[current_node][next_node]

    # Return randomly chosen path
    return save_graph_solution(total_distance, cycle)
```

Figure 16. Algorithm created for speed that shuffles the nodes and reports the distance between them

As can be seen in this function, all that happens is the list of nodes is shuffled and the distance calculated. While it may seem like this is not a useful piece of code, it is helpful in two ways. It will derive an answer on any size graph extremely quickly, faster than nearly any algorithm can run, allowing the user to get an instant start ahead of those who just run their algorithms to begin with and then begin running the heuristic approach algorithm. Also, it is useful in giving an idea of what a potential path size in the graph would be, and thus is a good way to comapre ones own heuristic algorithm to, to see what level of improvement has been achieved between the two.

My initial plan was to run this random_cycle_distance function to achieve a quick and easy distance for the TSP, submit that answer before anyone had a chance to finish their algorithm and then begin running my actual heuristic algorithm, thus allowing me to get an early start in the race, that would eventually be beaten by a slower piece of code.

While running my heuristic algorithm, I was able to near the best answers, but not beat them. This was especially more prominent at larger graph sizes where at size 15,000 I was able to reach an answer of 560851, which while rather large does compete with some better answers. Also discussing with classmates sitting nearby to me, I was able to achieve better solutions than they were on some of the graphs. This information is encouraging, as even if I was not able to beat the best solutions that were found, I was able to still generate relatively‘good' solutions to the problem.

IX. Conclusion

Overall, I think my solution was relatively successful, but there are still a multitude of improvements that could be made to increase the speed and accuracy of the program. I think first of all, making sure not to run the nearest neighbor algorithm on the same point twice, especially if we already know that it is larger than the result would be an easily implemented way to drastically lower time spent searching for solutions. Secondly as previously mentioned, subdividing the graph into smaller graphs, and solving each then reattaching them would be a way to improve the pathfinding ability. This may be a difficult task without knowing the true location of points, such as knowing an X and Y value in a two-dimensional plane. If this kind of information were known, it would be very easy to split the graph into sections. As it stands currently point 1 could be next to point 100 making it difficult to subdivide based on information about the points. Without this basic information it would have to be computed which could take away valuable compute time that could be used in finding solutions. Another solution that was discussed in class was picking starting points at randomly and slowly adding on to those, as to add new points to the edge that most effectively captures them. This way seemed to have strong results while also having a relatively simple approach to the problem.

This competition approach was a very interesting way to understand the problem, as it forced everyone to come up with unique solutions and see what techniques worked well and what techniques were less successful. I believe my solutions placed me somewhere in the middle of the class, as I was able to reach nearer to the top solutions, but not surpass any of them individually. It is worth noting that, working with limited hardware on my laptop and the chosen language of Python may have hindered some of the abilities of my algorithm to compete. I do think the strategy I went in with had potential for success. Overall, I am satisfied with both my solution and my overall placement within the wider class, I believe that in a second round I would be able to further refine my algorithm. It is also worth discussing, how seeing the time growth, especially that with larger problems, shows just how difficult these problems are to actually solve. It is impossible to verify that one solution would be the maximal and 'perfect' solution, and just that it is better or worse than another potential solution. Understanding, the P vs NP problem is useful and an important foundation in computer science.