Open in Colab

In [414]:

```
#@title Genetic Algorithm
crossover_rate = 0.9 #@param {type:"slider", min:0, max:1, step:0.01}
mutation_rate = 0.1 #@param {type:"slider", min:0, max:1, step:0.01}
crossover_method = 'Partially Mapped Crossover' #@param ['Order crossover', 'Maximal Pres
ervative Crossover', 'Partially Mapped Crossover']
mutation_method = 'Swap' #@param ['Swap', 'Insertion', 'Displacement']
threshold_votes_woc = 0.1 #@param {type:"slider", min:0, max:1, step:0.01}
experts_woc = 0.05 #@param {type:"slider", min:0, max:1, step:0.01}
vehicles_total = 22 #@param {type:"integer"}
```

In [429]:

```
from collections import defaultdict
import matplotlib.pyplot as plt
from operator import itemgetter
from itertools import cycle
from random import randrange, sample, random, randint
from statistics import mean,stdev
import networkx as nx
from math import ceil
import numpy as np
import re
import os

class Graph:
    '''
    Graph class that accpets file path as an argument
    '''

    def __init__(self, path):
        self.MUTATION_RATE = mutation_rate
        self.CROSSOVER_RATE = crossover_rate
        self.CROSSOVER_METHOD = crossover_method
        self.MUTATION_METHOD = mutation_method
        # WOC threshold inidicates the minimum votes to choose before using greedy algor
ithm
        # to complete the tour
        self.THRESHOLD_WOC = threshold_votes_woc
        # Percentage expert to vote
        self.EXPERTS_WOC = experts_woc
        # Select 70% of rouets with the minimum cost and 30 randomly
        # Total selection is 100% from population
        self.ELIT_PERCENT = 30
        self.RANDOM_PERCENT = 70
        # Total number of vehicles in DEPOT
        self.VEHILES_NUM = vehicles_total
        # Coordinates of all nodes in the graph
        self._coordinates = self.read_tsp_file(path)
        # Depot node is 1 by default
        self.DEPOT = 1
        # List of all nodes except DEPOT
        self._customers = [key for key in self._coordinates.keys() if key != self.DEPOT]
        # Total number of nodes
        self._size = len(self._coordinates.keys())
        # Distances between each nodes in the graph
        self._distances = self.compute_distances()
        # File index to generate a file name of the plot images
        self._file_index = 0
        # Direcotry to keep all plots
        self._GRAPH_DIR = 'graphs/'
        self.reset_file_index()

        # Mapping dictionary to crossover and mutation methods
        self.__crossovers = {
```

```
            'Order crossover': 'order_crossover',
            'Maximal Preservative Crossover': 'maximal_preservative_crossover',
            'Partially Mapped Crossover': 'partially_mapped_crossover'
            }

        self.__mutations = {
            'Swap': 'swap_mutation',
            'Insertion': 'insertion_mutation',
            'Displacement': 'displacement_mutation'
        }

    def reset_file_index(self):
        '''
        This method is reseting the index of the file name when generating graphs and rem
oving all plots
        '''
        self._file_index = 0

        full_path = os.path.abspath(self._GRAPH_DIR)

        for file in os.listdir(self._GRAPH_DIR):
            if file == '.keep':
                continue
            os.remove(os.path.join(full_path, file))
        return


    def read_tsp_file(self, path):
        """"
        Read TSP file
        """
        city_list = []

        with open(path, 'r') as f:
            _data = re.findall(r'\n([0-9].*)', f.read())

        # Split each line by space and convert to float
        for line in _data:
            city_list.append(tuple(map(float, line.split())))
        return { int(vertex): (x,y) for vertex, x, y in city_list }


    def euclidean_distance(self, nodeA, nodeB):
        """
        Formula to calculate the Euclidean distance
        """
        # Math library
        import math

        x1 = self._coordinates[nodeA][0]
        y1 = self._coordinates[nodeA][1]
        x2 = self._coordinates[nodeB][0]
        y2 = self._coordinates[nodeB][1]
        return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)


    def compute_distances(self):
        """
        Compute distance method invokes at initialization and calculates the cost of each
edge in the graph
        """
        self.distances = defaultdict(dict)
        for nodeA in self._coordinates.keys():
            for nodeB in self._coordinates.keys():
                if nodeA != nodeB and nodeB not in self.distances[nodeA]:
                    distance = self.euclidean_distance(nodeA, nodeB)
                    self.distances[nodeA][nodeB] = round(distance, 2)
                    self.distances[nodeB][nodeA] = round(distance, 2)
        return self.distances


    def divide_into_routes(self,tours):
```

```
    # Split tours into routes
    tours = np.array_split(tours, self.VEHILES_NUM)
    # Convert numpy array into list
    return [route.tolist() for route in tours]


  def total_distance(self, tours, mean_val = True):
    """
    Calculate total distance by passing completed solution with multiple
    tours, where number of tours equal to total available vehicles
    For example: [[1, 2, 3], [4, 1]]
    Retrun the tour with the max cost, we want to improve the solution
    where the max cost tour has the lowest cost
    """
    # Split tours into routes
    tours = self.divide_into_routes(tours)
    # print(f'Divide tour: {tours}')
    total_distance = []
    for single_route in tours:
      # Check if tour contains nodes
      if len(single_route) == 0:
        total_distance.append(0.00)
        continue

      # Reset distance after each tour calculation
      distance = 0
      for i in range(1, len(single_route)):
        distance += self._distances[single_route[i-1]][single_route[i]]

      # Add cost from Depot to first node
      distance += self._distances[self.DEPOT][single_route[0]]
      # Add cost from last node to Depot
      distance += self._distances[single_route[-1]][self.DEPOT]
      total_distance.append(round(distance, 2))
    # print(f'Total distance:{tours} - {total_distance} - {max(total_distance)}')
    return  round(sum(total_distance),2) if mean_val else total_distance



  def create_route(self):
    '''
    Create route method create a random route from the city list
    '''
    # Generate a random list of customers
    route = sample(self._customers, len(self._customers))
    # Geneate a list of chunk length
    # Return only the first two groups, because np.array_split return additional an emp
ty group
    #chunk_length = np.random.multinomial(self._size - 1, np.ones(self.VEHILES_NUM)/sel
f.VEHILES_NUM, size=1)[0][:self.VEHILES_NUM -1]
    # Split the random customer list into groups.
    # where number of gorups corresponds to the total number of vehicles available\
    #np.cumsum of [2,3,4] return [2,5,9]
    #chromosome = np.array_split(random_customers, np.cumsum(chunk_length))
    #!chromosome = np.array_split(random_customers, self.VEHILES_NUM)
    # Convert inner numpy arrays to lists
    #!return [route.tolist() for route in chromosome]
    return route


  def create_init_population(self, population_size):
    '''
    Create initial population method accepts one argument, which is the size
    of random population to create and return N number of random routes
    '''
    population = []

    for i in range(0, population_size):
      population.append(self.create_route())
    return population
```

```python
    def rank_routes(self, population):
      '''
      Accepts population list and returns normalized weights
      between 0 and 1
      '''

      fitness_score = {}

      for i, tour in enumerate(population):
        # Since we are interested in lowest distnace
        # we need to devide 1 by total distance
        # and get the tour from chromosom with max cost
        # print(self.total_distance(tour))

        fitness_score[i] =  1 / (self.total_distance(tour) + 1)
      # Calculate total fitness sctore for all routes
      total_score = sum(fitness_score.values())

      # Normalize the score and sort it from lowest to highest
      for k, v in fitness_score.items():
        fitness_score[k] = round(fitness_score[k] / total_score, 4)
      return sorted(fitness_score.items(), key = itemgetter(1), reverse=True)


    def pick_tour(self, fintess_score ):
      '''
      Choose tour based on it is total cost
      '''
      index = 0
      random_number = random()

      while random_number > 0:
        random_number -= fintess_score[index][1]
        index += 1
        # Pick up the last element if index is greater then total
        if index >= len(fintess_score):
          break
      return fintess_score[index-1]


    def selection(self, fitness):
      '''
      Selection methods accepts list of touple with index and normalized costs\
      and returns a list of indexes of selected routs
      '''
      # ELIT_PERCENT = 30
      # RANDOM_PERCENT = 70
      elit_index = ceil((len(fitness)*self.ELIT_PERCENT)/100)
      result = [tour[0] for tour in fitness[:elit_index]]

      # Generate population based on RANDOM_PERCENT value
      for i in range(ceil((len(fitness)*self.RANDOM_PERCENT)/100)):
        result.append(graph.pick_tour(fitness)[0])
      return result

    def mating_pool(self, population, selection_results):
      '''
      Mating pool methods accpets the following argiments:
      - population which is a list of tours
      - selection results whihc is an index of all selected routs in the tour
      '''
      return [population[i] for i in selection_results]


    # CROSSOVER METHODS
    def order_crossover(self, p1, p2):
      '''
      Use order crossover algorithm to swap cities
      '''
      # Start point in cut
      s = randint(1, len(p1) - 2)
      # End point in cut
```

```python
        e = randint(s + 1, len(p1))

        c1, c2, p1, p2 = p1[s:e],\
                         p2[s:e],\
                         p1[e:] + p1[:e],\
                         p2[e:] + p2[:e]

        c1.extend([i for i in p2 if i not in c1])
        c2.extend([i for i in p1 if i not in c2])
        return c1, c2


    def maximal_preservative_crossover(self, p1, p2):
        '''
        Use maximal preservative algorithm to swap cities
        '''
        # Find how many elemets to swap
        c = len(p1) // 2
        # Find starting point
        r = randrange(len(p1) + 1)
        # Extend the list, which will allow to move back to index 0
        c1, c2 = (p1 * 2)[r:r + c], (p2 * 2)[r:r + c]

        c1.extend([i for i in p2 if i not in c1])
        c2.extend([i for i in p1 if i not in c2])
        return c1,c2


    def partial_mapping(self, p, c, s, e):
        '''
        Method accepts 4 argumets: p - parent, c - child, s - start index
        e - end index,  and open 40% of genes in child
        '''
        # Genearte a set of all parent indices
        p_index = set(range(len(p)))
        # Generate a set of cut indeces
        cut_index = set(range(s,e))
        # Find differnce of indices
        left_index = list(p_index - cut_index)
        # Find 40% of indices to open
        total_open = ceil(len(left_index) * 0.4)
        # Get Random indices
        random_index = sample(left_index, total_open)
        # Open indices in child tour
        for i in random_index:
          if p[i] not in c:
            c[i] = p[i]
        return c


    def partially_mapped_crossover(self, p1, p2):
        # Find start and end index
        s = randint(1, len(p1) - 2)
        e = randint(s + 1, len(p1))
        # print(f'Start-End: {s},{e}')

        # Swap cut part between parents
        c1, c2 = [0] * len(p1), [0] * len(p2)
        c2[s:e], c1[s:e] = p1[s:e], p2[s:e]

        # Open partial genes
        c1 = self.partial_mapping(p1, c1, s, e)
        c2 = self.partial_mapping(p2, c2, s, e)

        # Complete tour
        for i in p2:
          if i not in c1:
            c1[c1.index(0)] = i

        for i in p1:
          if i not in c2:
            c2[c2.index(0)] = i
```

```
        return c1, c2


    # MUTATION ALGORITHMS
    def swap_mutation(self, tour):
        '''
        Swap mutation method accepts one argument, which is a generated
        tour by selection.

        To perform swap mutation select two alleles at random and
        swap their positions. It preserves most of the adjacency
        information but links broken disrupts order more
        '''
        i, j = sample(range(len(tour)),2)
        tour[i], tour[j] = tour[j], tour[i]
        return tour


    def insertion_mutation(self, tour):
        '''
        Insertion mutation method accepts one argument, which is a generated
        tour by selection.

        It is used in Permutation encoding. First of all, pick two allele values
        at random. Then move the second allele to follow the first, shifting the
        rest along to accommodate. Note that this preserves most of the order
        and the adjacency information
        '''
        random_city, random_position = sample(range(len(tour)),2)
        city = tour.pop(random_city)
        tour.insert(random_position, city)
        return tour


    def displacement_mutation(self, tour):
        '''
        Displacement mutation method accepts one argument, which is a generated
        tour by selection.

        Displacement mutation inserts a random string of cities in another
        random place.
        '''

        # Find start and end index
        s = randint(1, len(tour) - 2)
        e = randint(s + 1, len(tour))

        random_position = randint(0, len(tour))
        substring = tour[s:e]
        tour = tour[:s] + tour[e:]
        return tour[:random_position] + substring + tour[random_position:]


    def create_new_generation(self, current_generation, woc = False):
        # Find the name of crossover and mutation methods to use
        crossover = self.__crossovers[self.CROSSOVER_METHOD]
        mutation = self.__mutations[self.MUTATION_METHOD]
        # Create a list of indices from population with normalized weights
        fitness = self.rank_routes(current_generation)
        best_route = current_generation[fitness[0][0]]

        # SELECTION STEP
        # Select method chooses 100% of current population when 30% is routs with
        # best costs and 70% is random with high probabilty of best routes
        selection_result = self.selection(fitness)
        # Generate a mating pool from current population based on indices from
        # selection result
        mating_pool = self.mating_pool(current_generation, selection_result)

        next_generation = []

        # CROSSOVER STEP
```

```
      for pair in zip(mating_pool[::2], mating_pool[1::2]):
          next_generation.extend(getattr(self, crossover)(*pair)\
                                 if random() < self.CROSSOVER_RATE else pair)
      # MUTATION STEP
      next_generation = [getattr(self, mutation)(tour)\
                         if random() < self.MUTATION_RATE else tour\
                         for tour in next_generation]


      # Order the generation according to the fitness value
      next_generation = sorted(next_generation, key=self.total_distance)

      # print(f'Next generation:')
      # print(*next_generation, sep='\n')

      # Find best tour by using WOC algorithms with specified expert percentage from popu
lation
      if woc:
        woc_tour = self.woc_algorithm(next_generation[:int(self.EXPERTS_WOC * len(next_g
eneration))])
        # print(f'WOC TOUR: {woc_tour}')
        # print(f'WOC Total cost: {sum(self.total_distance(woc_tour, mean_val=False))}')
        # print(f'WOC MEAN cost: {mean(self.total_distance(woc_tour, mean_val=False))}')
        # print(f'GA TOUR: {next_generation[0]}')
        # print(f'GA Total cost: {sum(self.total_distance(next_generation[0], mean_val=F
alse))}')
        # print(f'GA MEAN cost: {mean(self.total_distance(next_generation[0], mean_val=F
alse))}\n')

        if woc == 'include':
          # Replace the last tour in next generation with WOC tour
          next_generation[-1] = woc_tour
          # Next generation needs to be sorted again after WOC tour was added
          next_generation = sorted(next_generation, key=self.total_distance)

        # Build result with next generation
        result = {'next_generation': next_generation,
                  'genetic_output': {'best_route': self.divide_into_routes(next_generati
on[0]),
                                      'total_cost': self.total_distance(next_generation[
0], mean_val=False)},
                  'woc_output': {'best': self.divide_into_routes(woc_tour),
                                 'total_cost': self.total_distance(woc_tour, mean_val=Fa
lse)}
          }
      else:
          result = {'next_generation': next_generation,
                    'genetic_output': {'best_route': self.divide_into_routes(next_genera
tion[0]),
                                       'total_cost': self.total_distance(next_generation
[0], mean_val=False)}
          }

      # best_route = next_generation[0]
      # total_cost = self.total_distance(best_route)
      # print(f'Best Tour: {best_route}')
      # print(f'Best Tour cost: {total_cost}')
      # print(f'WCO tour: {woc_tour}')
      # print(f'WCO cost: {self.total_distance(woc_tour)}\n')


      # return next_generation, best_route, total_cost
      return result


    def greedy_algorithm(self, path):
      '''
      Gread algortithm find the the next city in the best based on min cost.
      Accepts:
      - path which is incomplete path
      '''
      # Find all missing nodes but set difference
```

```
        missing_nodes = list(set(self._customers) - set(path))
        # Set last node in path as next element
        nxt_elmt = path[-1]

        while len(missing_nodes) != 0 :
          # Run while loop till all nodes are added to the path
          # Find the cost of missing nodes
          cost = {k:v for k, v in self._distances[nxt_elmt].items() if k in missing_nodes}
          # Get next element with min cost
          nxt_elmt = min(cost.items(), key = itemgetter(1))[0]
          # Append node to the path
          path.append(nxt_elmt)
          # Remove next element from missing nodes
          missing_nodes.remove(nxt_elmt)
          # print(f'Cost: {cost},\n Next element: {nxt_elmt}')
          # print(f'Path: {path}')
          # print(f'Missing nodes: {missing_nodes}\n')
        return path


      def woc_find_next_node(self, matrix, next_element, path, row = [], column = []):
          '''
          Wco_find_next_node method lookes for the next node in
          matric's row and columns based on max votes count

          '''
          # This function is using global variables
          # global lmatrix, next_element, row_index, column_index, path
          # local lmatrix, next_element, row_index, column_index, path

          # Check if row or column is set to True
          if len(row) == 0 and len(column) == 0:
            return False

          # Based on passed value, create iterator row or column
          node_list = row if len(row) != 0 else column

          # Node list contains indecis with the max value which also represents next node
          for node in node_list:
            if len(row) != 0:
              matrix[next_element, node] = 0
            else:
              matrix[node, next_element] = 0
            if node not in path and node != 0:
              # If node is not in the path that append it to the path and use it to find
              # the next node based on the votes count
              # print(f'Node {node} NOT in the path row')
              path.append(node)
              next_element = node
              # print(f'Path: {path}, Next element: {next_element}')
              break
            # Node is already in the path continue
            # print(f'Path: {path}, Node {node} in the path row')
            continue
          return True


      def woc_algorithm(self, population):
        '''
        WCO algorithm finds best path based on total edge counts from
        population.
        It accepts one argumet population which is population of experts
        '''

        # Create a matrix with zeros where, where the frist column and row are not in used
        # Should not add one, since one node is Depot
        matrix = np.zeros(shape=(self._size + 1, self._size + 1))
        # print(population)

        # Populate the matrix with edge counts
        for path in population:
          for i in range(0,len(path) -1):
```

```python
            # Populate upper and lower triangle sides
            matrix[path[i],path[i + 1]] = matrix[path[i],path[i + 1]] + 1
            matrix[path[i + 1],path[i]] = matrix[path[i + 1],path[i]] + 1
            # print(f'{path[i]} - {path[i + 1]}')

        # matrix[2,2] = 10
        # print(matrix)
        # print(np.tril(matrix, -1))
        # Use lower triangle side since the upper side is symetrical
        # The one side is used to save memory
        matrix = np.tril(matrix, -1)
        # Only get from from matrix that does not exceed max votes threshold
        # Multiple the maximam votes in matrix with set threshold
        max_votes_threshold = int(np.amax(matrix) * self.THRESHOLD_WOC)
        # print(f'MAX VOTES THRESHOLD: {max_votes_threshold}')
        # Find the first edge with the highest votes
        path = [city[0] for city in np.where(matrix == np.amax(matrix))]
        # print(path)
        # Mark edge with 0 votes in matrix
        matrix[path[0], path[1]] = 0
        # Start with second element in the path
        next_element = path[-1]

        while len(path) < len(population[0]):
          # Get the row of the node in matrix
          row = matrix[next_element, :]
          # Get the column of the node in matrix
          column = matrix[:, next_element]
          # print(matrix[next_element, :])
          # print(matrix[:, next_element])
          # print(matrix)
          # Find the maximum value in row
          max_row = np.amax(matrix[next_element, :])
          # Find the maximum value in column
          max_column = np.amax(matrix[:, next_element])
          # print(max_row)
          # print(max_column)
          # Find all indecis for the maximum value in row
          row_index = np.where(row == max_row)[0]
          # Find all indecis for the maximum value in column
          column_index = np.where(column == max_column)[0]
          # print(row_index)
          # print(column_index)
          # print(matrix)

          # If maximum value in row and column is less than the threshold
          # use greedy heuristic
          if max_row <= max_votes_threshold and max_column <= max_votes_threshold:
          # if not (int(max_row) or int(max_column)):
            self.greedy_algorithm(path)
            break

          if max_row >= max_column:
            # Find next element in row
            self.woc_find_next_node(matrix, next_element, path, row = row_index)
          else:
            # Find next element in column
            self.woc_find_next_node(matrix, next_element, path, column = column_index)

        return path


    def genetic_algorithm(self, pop_size=10, term_count=100 , woc = False):
        '''
        Genetic algorithm method accepts the following arguments:
        - pop_size is the size of random initial tours, Default value is 100
        - term_count indicates when to terminate an algorithms after not seeing improving
          in total cost
        - woc is boolean variable indicates if woc algorithm needs to be applied
              False - if WOC doesn't need to be applied
              guess - run WOC, but not add the tour to next generation
              include - run WOC and includ tour in next generation
```

```
        Returns:
                best_route - (Completed routes, list of routes cost, mean cost, total co
st, number of generations)
                progress_ga - total cost of all routes only using GA
                progress_woc - total cost of all routes with WOC
        '''
        # Generate initial population
        population = self.create_init_population(pop_size)
        # Instanciate total cost as infinity value
        total_cost = float('inf')
        # Count generation when there is no improvements
        term_index = 0
        # Count total generation
        generations = 0
        # All selected best routes
        best_routes = []
        progress_ga = []
        progress_woc = []


        # Run while loop till there is no improvements in "term_count" times
        while term_index <= term_count:
          generations += 1
          result = self.create_new_generation(population, woc)
          population = result['next_generation']
          progress_ga.append(round(sum(result['genetic_output']['total_cost']),2))
          if woc:
            progress_woc.append(round(sum(result['woc_output']['total_cost']),2))

          if sum(result['genetic_output']['total_cost']) < total_cost:
              best_route = result['genetic_output']['best_route']
              total_cost = sum(result['genetic_output']['total_cost'])

              # Create a list of tuples that contains the best route which is
              # completed cycle, weights for each route, mean weights among all routes,
              # total weights, number of total generations
              best_routes.append(([[self.DEPOT] + route  + [self.DEPOT] for route in best_
route],
                                  result['genetic_output']['total_cost'],
                                  round(mean(result['genetic_output']['total_cost']),2),
                                  round(sum(result['genetic_output']['total_cost']),2),
                                  generations))
              term_index = 0
          else:
            term_index += 1


          # population = result[0]
          # progress.append(result[2])
          # if result[2] < total_cost:
          #     best_route = result[1]
          #     total_cost = result[2]
          #     # Create a list of tuples that contains the best route which is
          #     # completed cycle, total weights and number of total generations
          #     best_routes.append((result[1] + [result[1][0]], result[2], generations))
          #     term_index = 0
          # else:
          #   term_index += 1
        return best_routes, progress_ga, progress_woc


    def draw_plot(self, tour_info, weights=False):
          '''
          Draw plot methods accepts the following argumetns:
          - tour_info variable, which contains a list of tuples of route, route cost and to
tal generations
          - weights, when set to true addes weigths to each graph
          '''
          tours = tour_info[0]
          mean_tour_cost = tour_info[2]
          total_cost = tour_info[3]
          total_generations = tour_info[4]
```

```
        # Create new graph
        G = nx.Graph()
        pos = self._coordinates
        # Image name
        file_name = f'{self._file_index:0{len(str(self._size))}d}'

        # # Add edges to the graph
        tours_edges = {}
        route_indx = 0

        for tour in tours:
          tour_edges = []
          for i in range(len(tour)-1):
              G.add_edge(tour[i], tour[i+1], weight = self._distances[tour[i]][tour[i+1]
])
              tour_edges.append((tour[i], tour[i+1]))
          tours_edges[route_indx] = tour_edges
          route_indx += 1

        # Display weights of each edge
        if weights:
            # Get weight for each edge
            labels = nx.get_edge_attributes(G,'weight')
            # Drow weights to each edge
            nx.draw_networkx_edge_labels(G, pos, edge_labels=labels, font_size=8)

        # Add nodes
        options = {"node_size":200, "alpha": 0.5}
        nx.draw_networkx_nodes(G, pos, nodelist=list(self._customers), node_color="red",
**options)
        nx.draw_networkx_nodes(G, pos, nodelist=[self.DEPOT], node_color="black", **opti
ons)

        # Iterate over all routes and color them
        # Create color iterable
        cycol = cycle('rbgcmyk')
        for i in tours_edges.keys():
          nx.draw_networkx_edges(
              G,
              pos,
              edgelist=tours_edges[i],
              width=0.5,
              alpha=0.5,
              edge_color=next(cycol)
          )

        # Draw lables in the node
        nx.draw_networkx_labels(G,pos, font_size=10)

        # Increase the size of the plots
        fig = plt.gcf()
        fig.set_size_inches(18, 10.5)
        #plt.figure(1,figsize=(50,50))

        plt.axis("off")
        plt.title(f'VRP with Genetic Algorithm and Wisdom of Artificial Crowds'
          + f'\nTotal customer\'s nodes: {graph._size-1}'
          + f'\nTotal vehicles: {graph.VEHILES_NUM}'
          + f'\nMean costs: {mean_tour_cost}'
          + f'\nTotal cost: {total_cost}'
          + f'\nTotal Generations: {total_generations}')
        plt.savefig(f'graphs/project_6_{file_name}.png')
        plt.show()
        # Generate the file name in ascending order
        self._file_index += 1
```

In [3]:

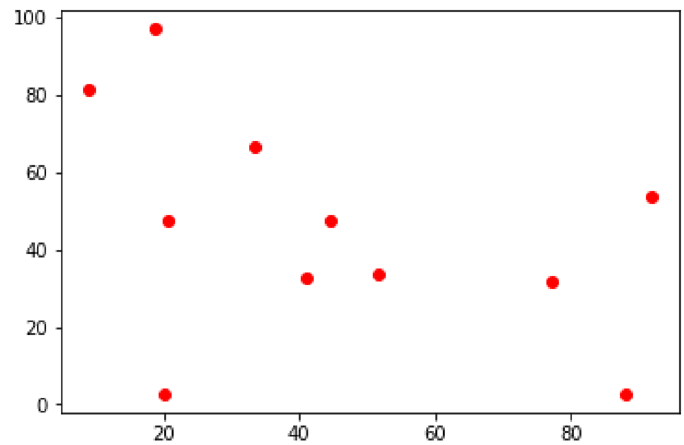
```
# Plot all nodes in the graph
import matplotlib.pyplot as plt
import os
```

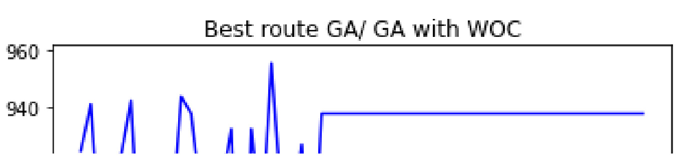
```
import imageio


def plot_nodes(graph):
  x = [coord[0] for coord in graph._coordinates.values()]
  y = [coord[1] for coord in graph._coordinates.values()]

  plt.scatter(x, y, color='red')
  plt.show()

def plot_ga_and_woc(result, name):
  plt.plot(results[1], color='red', label='GA best route')
  plt.plot(results[2], color='blue', label='WOC best route')
  plt.ylabel('Distance')
  plt.xlabel('Generation')
  plt.title('Best route GA/ GA with WOC')
  plt.legend()
  plt.savefig(name)
  plt.show()

def create_gif(gif_name):
  # Generate gif images from png plots
  images_dir = 'graphs'
  image_list = []

  # Generate a list of absolute paths to the png images
  filelist= sorted([os.path.abspath(f'{images_dir}/{file}') for file in os.listdir(image
s_dir) if file.endswith('.png')])
  for file_name in filelist:
      image_list.append(imageio.imread(file_name))

  # Create gif file
  kargs = { 'duration': 0.3 }
  imageio.mimwrite(gif_name , image_list, **kargs)
```

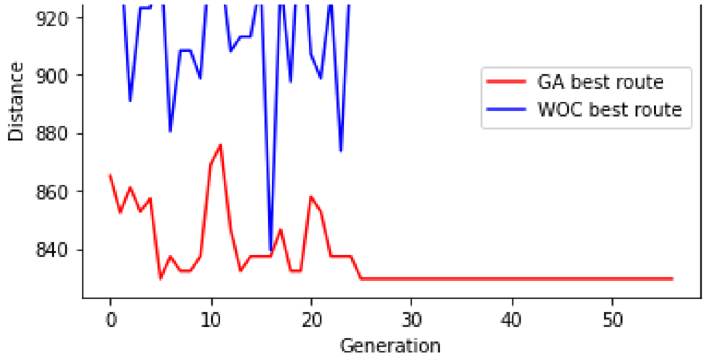
VRP with 1 Depot,10 customers and 5 drivers

In [245]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random11.tsp')
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_10_customers.png')
```








```
CPU times: user 609 ms, sys: 14 ms, total: 623 ms
Wall time: 623 ms
```

Best cost without WOC

In [240]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 165.93 Total: 829.64 Generations: 4
Mean: 165.93 Total: 829.64 Generations: 15
Mean: 165.93 Total: 829.64 Generations: 4
Mean: 165.93 Total: 829.64 Generations: 10
Mean: 165.93 Total: 829.64 Generations: 4
Mean: 165.93 Total: 829.64 Generations: 6
2 loops, best of 3: 236 ms per loop
```

Best cost with WOC

In [241]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 165.93 Total: 829.64 Generations: 12
Mean: 165.93 Total: 829.64 Generations: 12
Mean: 165.93 Total: 829.64 Generations: 1
Mean: 165.93 Total: 829.64 Generations: 6
Mean: 165.93 Total: 829.64 Generations: 9
Mean: 165.93 Total: 829.64 Generations: 5
2 loops, best of 3: 337 ms per loop
```

In [242]:

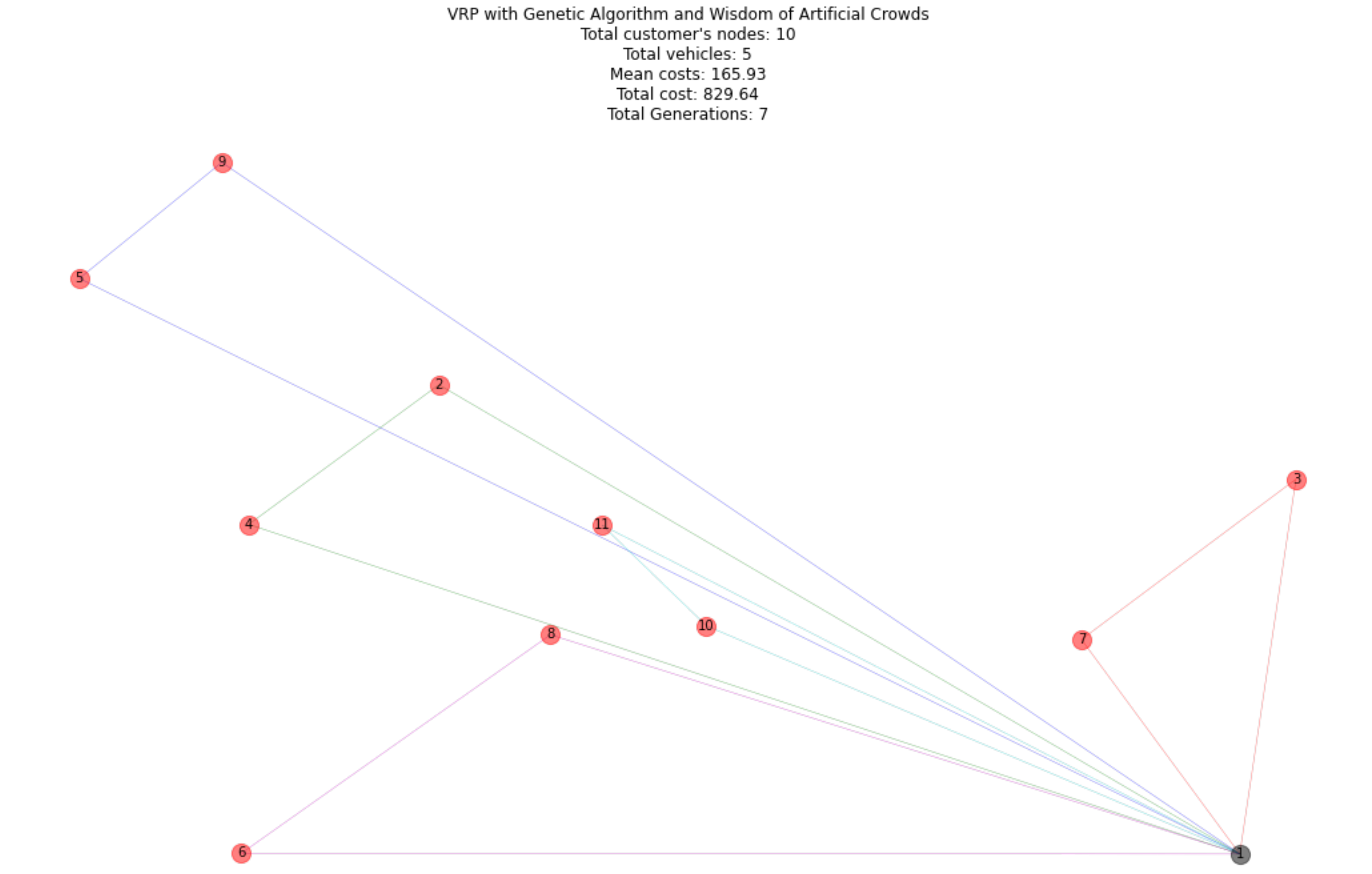
```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 7, 3, 1], [1, 9, 5, 1], [1, 4, 2, 1], [1, 11, 10, 1], [1, 8, 6, 1]]
Total routes: 5
Individual routes cost: [108.78, 246.96, 188.15, 125.72, 160.03]
Mean route cost: 165.93
Total cost: 829.64
Total number generations: 7
```

In [243]:

```
graph.draw_plot(results[0][-1], weights=False)
```

```
graph.draw_plot(results[0][-1], weights=False)
```



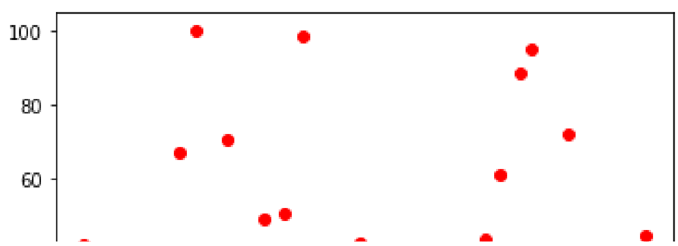
In [ ]:

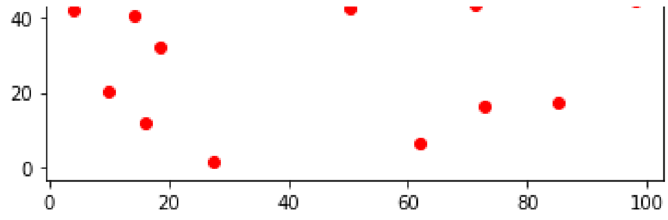
```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)

# Create gif image
create_gif('GA_with_WOC_10.gif')
```

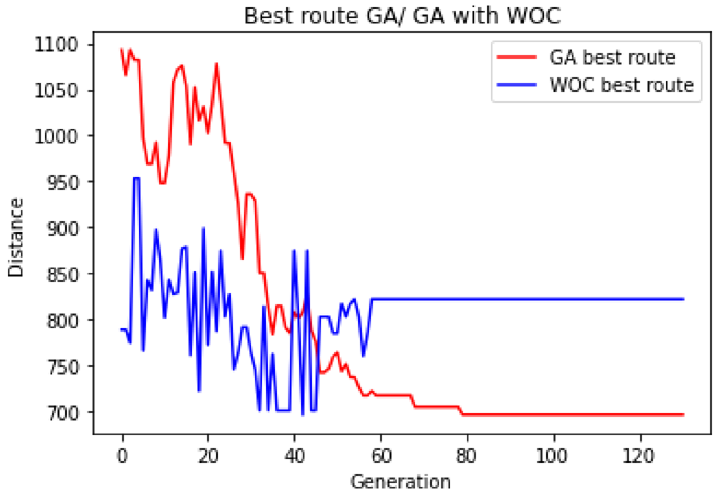
VRP with 1 Depot, 21 customers and 5 drivers

In [276]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random22.tsp')
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_22_customers.png')
```








```
CPU times: user 1.06 s, sys: 4.97 ms, total: 1.07 s
Wall time: 1.07 s
```

Best cost without WOC

In [248]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 133.84 Total: 669.21 Generations: 85
Mean: 135.17 Total: 675.86 Generations: 79
Mean: 131.88 Total: 659.39 Generations: 68
Mean: 135.83 Total: 679.14 Generations: 135
Mean: 138.5 Total: 692.52 Generations: 116
Mean: 133.0 Total: 665.02 Generations: 81
2 loops, best of 3: 665 ms per loop
```

Best cost with WOC

In [249]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 140.22 Total: 701.09 Generations: 8
Mean: 145.65 Total: 728.23 Generations: 59
Mean: 134.1 Total: 670.52 Generations: 72
Mean: 133.43 Total: 667.15 Generations: 62
Mean: 137.0 Total: 684.98 Generations: 24
Mean: 135.3 Total: 676.5 Generations: 185
2 loops, best of 3: 690 ms per loop
```

In [303]:

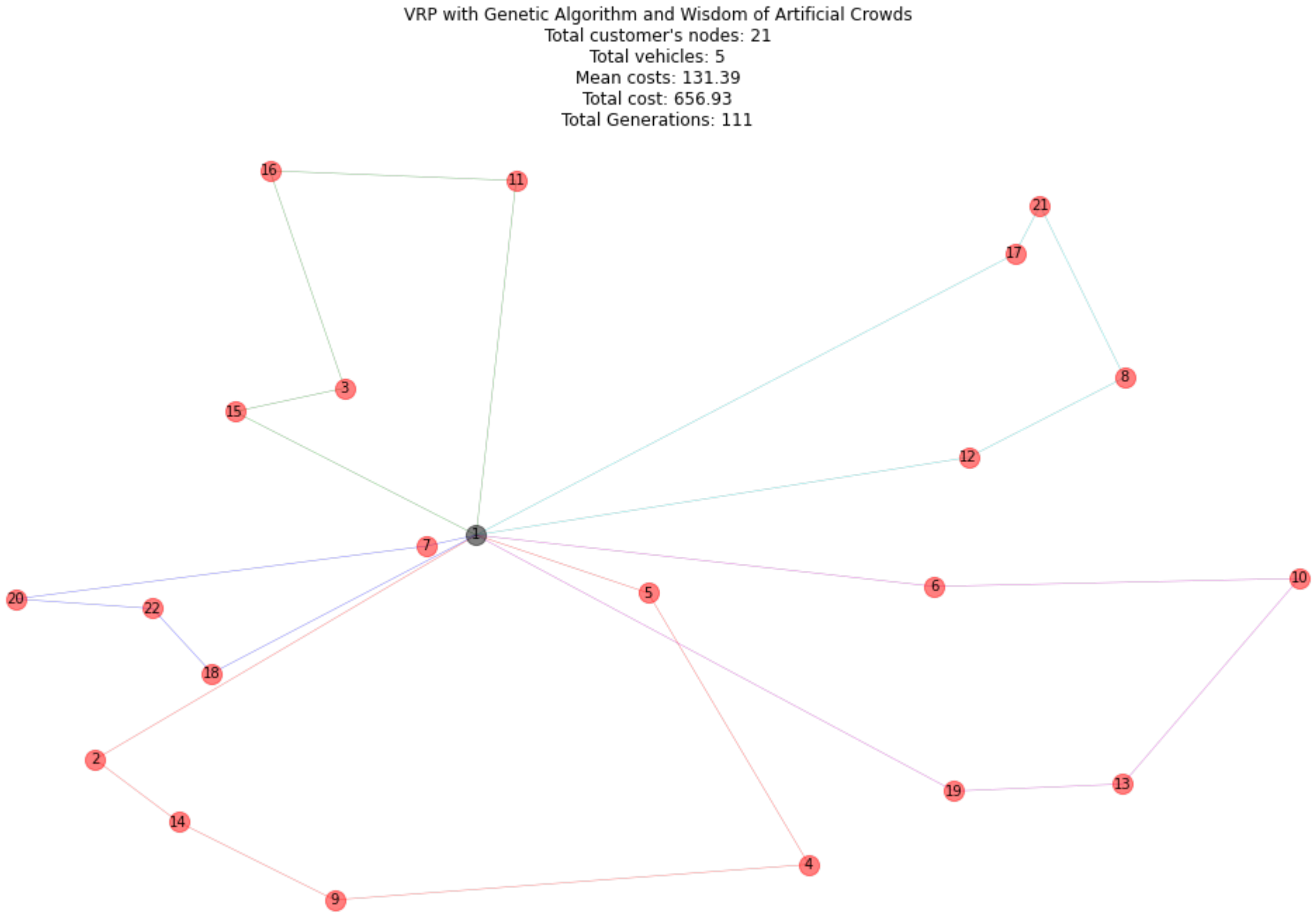
```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
```

```
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 5, 4, 9, 14, 2, 1], [1, 18, 22, 20, 7, 1], [1, 15, 3, 16, 11, 1], [1,
17, 21, 8, 12, 1], [1, 6, 10, 13, 19, 1]]
Total routes: 5
Individual routes cost: [155.28, 81.42, 128.63, 138.62, 152.98]
Mean route cost: 131.39
Total cost: 656.93
Total number generations: 111
```

In [304]:

```
graph.draw_plot(results[0][-1], weights=False)
```



In [ ]:

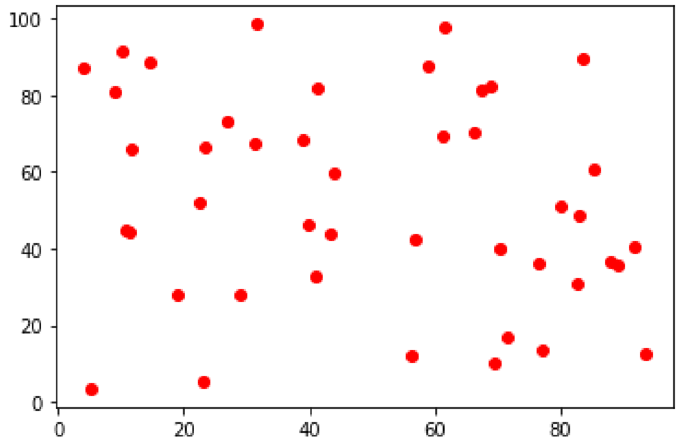
```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)

# Create gif image
create_gif('GA_with_WOC_22.gif')
```

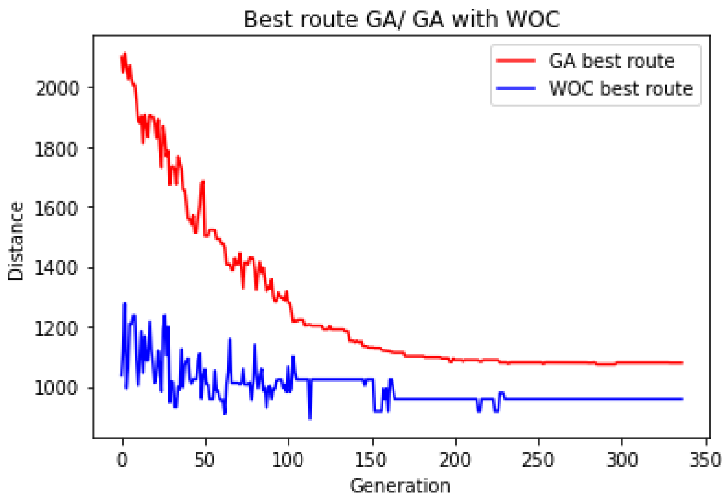
VRP with 1 Depot and 43 customers

In [306]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random44.tsp')
```

```
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_44_customers.png')
```






```
CPU times: user 3.02 s, sys: 13 ms, total: 3.03 s
Wall time: 3.04 s
```

Best cost without WOC

In [255]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 224.12 Total: 1120.59 Generations: 188
Mean: 217.61 Total: 1088.05 Generations: 337
Mean: 219.26 Total: 1096.28 Generations: 116
Mean: 181.06 Total: 905.28 Generations: 309
Mean: 205.57 Total: 1027.83 Generations: 563
Mean: 198.7 Total: 993.52 Generations: 448
2 loops, best of 3: 1.81 s per loop
```

Best cost with WOC

In [139]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 176.81 Total: 884.04 Generations: 3
```

```
Mean: 176.51 Total: 882.53 Generations: 161
Mean: 171.39 Total: 856.96 Generations: 169
Mean: 171.39 Total: 856.96 Generations: 183
Mean: 178.94 Total: 894.7 Generations: 56
Mean: 188.71 Total: 943.54 Generations: 55
2 loops, best of 3: 2.06 s per loop
```

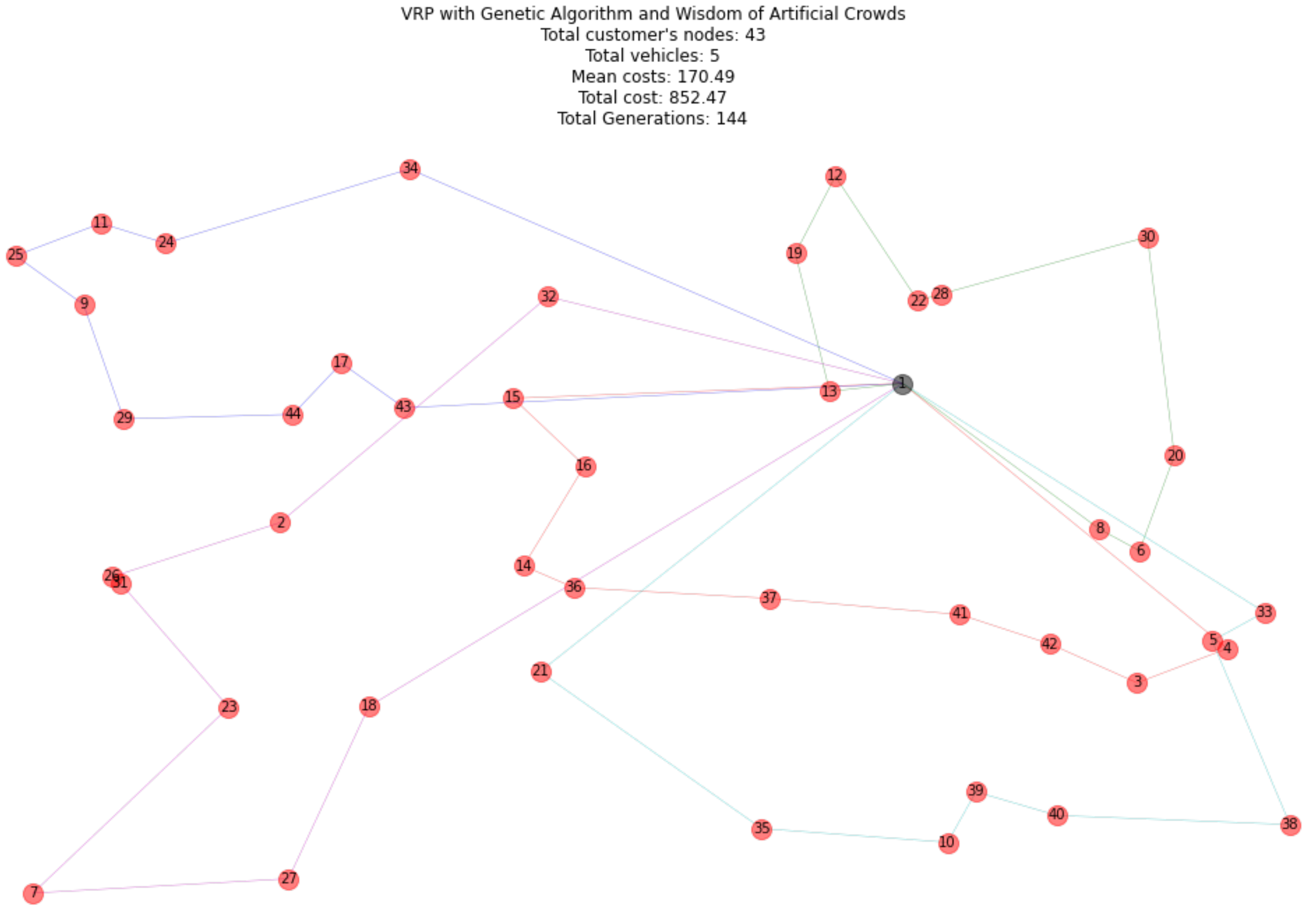
In [331]:

```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 4, 3, 42, 41, 37, 36, 14, 16, 15, 1], [1, 43, 17, 44, 29, 9, 25, 11,
24, 34, 1], [1, 13, 19, 12, 22, 28, 30, 20, 6, 8, 1], [1, 33, 5, 38, 40, 39, 10, 35, 21, 1
], [1, 32, 2, 26, 31, 23, 7, 27, 18, 1]]
Total routes: 5
Individual routes cost: [147.78, 161.79, 138.48, 183.49, 220.93]
Mean route cost: 170.49
Total cost: 852.47
Total number generations: 144
```

In [332]:

```
graph.draw_plot(results[0][-1], weights=False)
```



In [ ]:

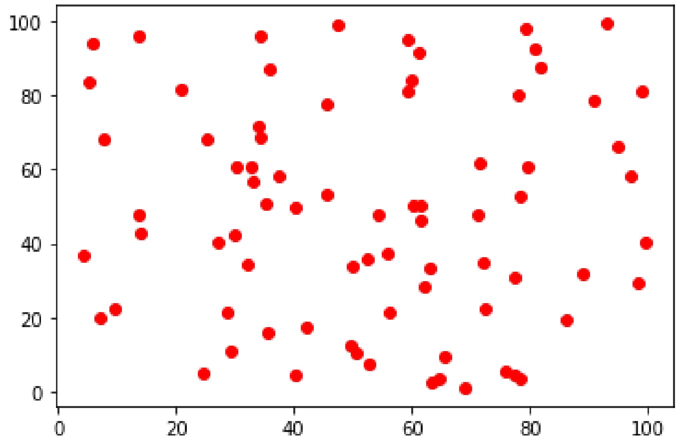
```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)
```

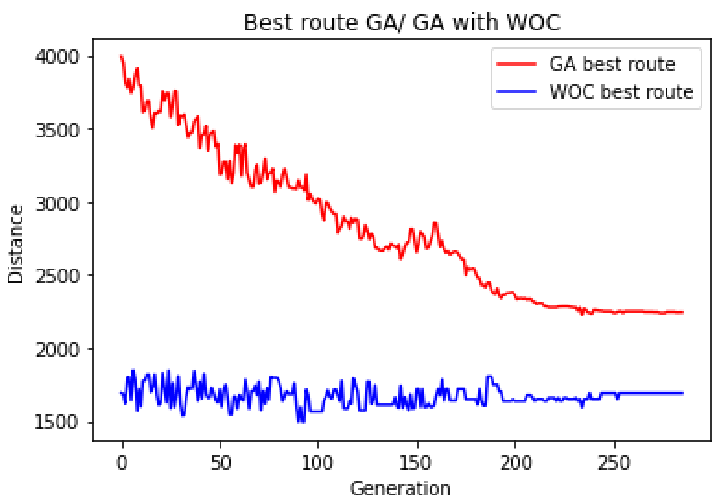
```
# Create gif image
create_gif('GA_with_WOC_44.gif')
```

VRP with 1 Depot,76 customers and 8 drivers

In [349]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random77.tsp')
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_77_customers.png')
```






```
CPU times: user 4.69 s, sys: 11 ms, total: 4.7 s
Wall time: 4.71 s
```

Best cost without WOC

In [350]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 345.4 Total: 2763.17 Generations: 116
Mean: 226.55 Total: 1812.42 Generations: 504
Mean: 243.84 Total: 1950.7 Generations: 514
Mean: 400.12 Total: 3200.95 Generations: 48
```

```
Mean: 327.73 Total: 2621.82 Generations: 135
Mean: 238.21 Total: 1905.64 Generations: 791
2 loops, best of 3: 3.98 s per loop
```

Best cost with WOC

In [351]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 187.45 Total: 1499.61 Generations: 48
Mean: 190.24 Total: 1521.91 Generations: 50
Mean: 192.07 Total: 1536.6 Generations: 12
Mean: 192.1 Total: 1536.81 Generations: 8
Mean: 190.95 Total: 1527.63 Generations: 32
Mean: 185.21 Total: 1481.69 Generations: 112
2 loops, best of 3: 1.34 s per loop
```

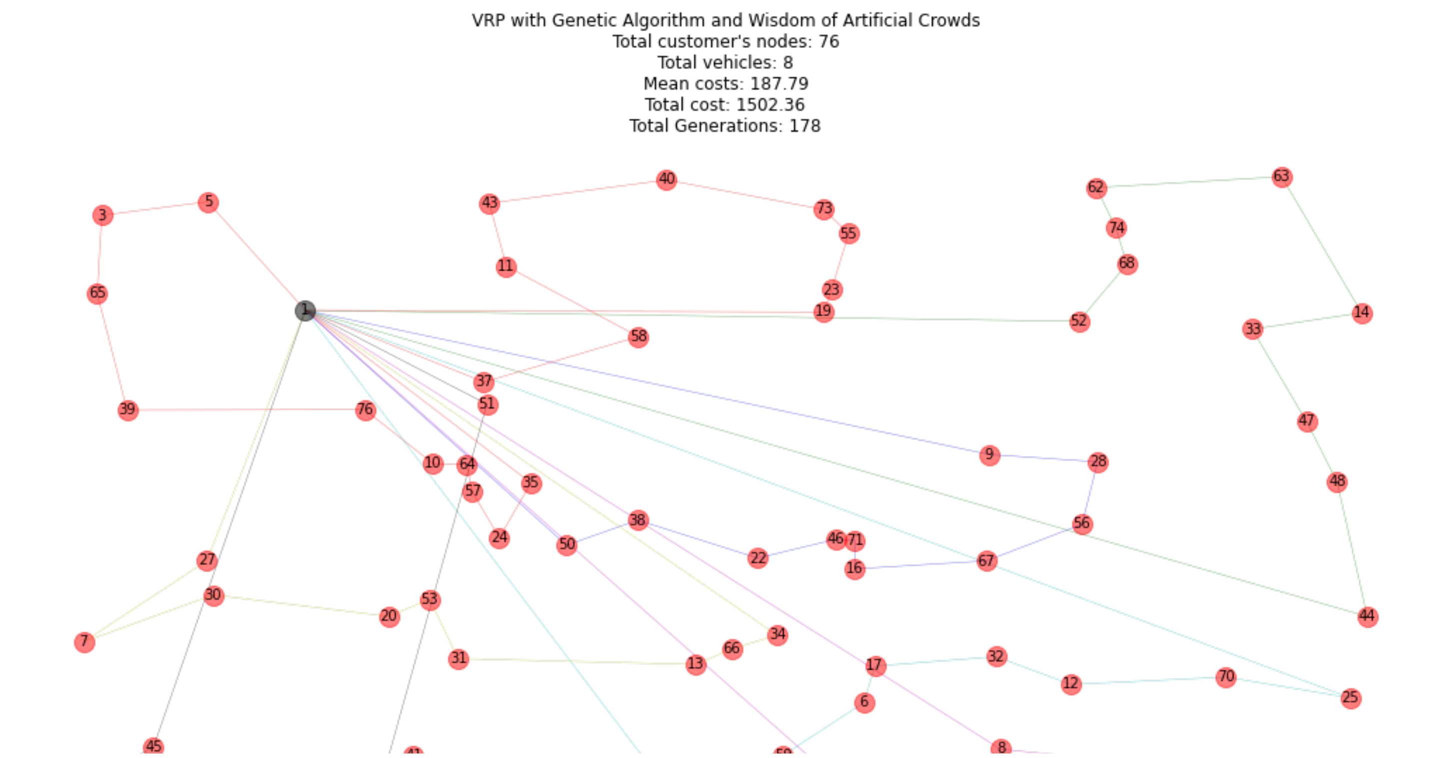
In [377]:

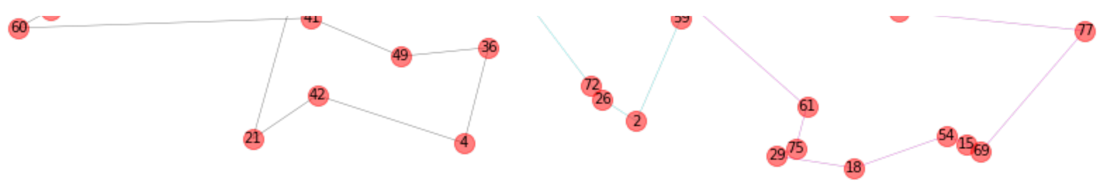
```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 5, 3, 65, 39, 76, 10, 64, 57, 24, 35, 1], [1, 50, 38, 22, 46, 71, 16,
67, 56, 28, 9, 1], [1, 52, 68, 74, 62, 63, 14, 33, 47, 48, 44, 1], [1, 25, 70, 12, 32, 17,
6, 59, 2, 26, 72, 1], [1, 61, 75, 29, 18, 54, 15, 69, 77, 8, 1], [1, 34, 66, 13, 31, 53, 2
0, 30, 7, 27, 1], [1, 45, 60, 41, 49, 36, 4, 42, 21, 51, 1], [1, 37, 58, 11, 43, 40, 73, 5
5, 23, 19, 1]]
Total routes: 8
Individual routes cost: [126.53, 154.67, 246.79, 239.99, 220.4, 166.81, 216.76, 130.41]
Mean route cost: 187.79
Total cost: 1502.36
Total number generations: 178
```

In [378]:

```
graph.draw_plot(results[0][-1], weights=False)
```





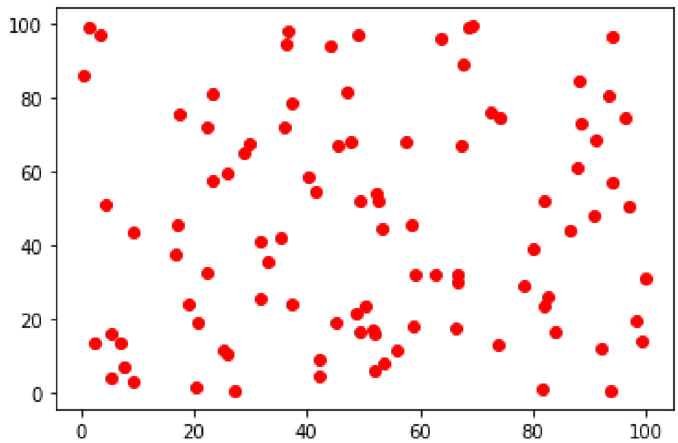
In [ ]:

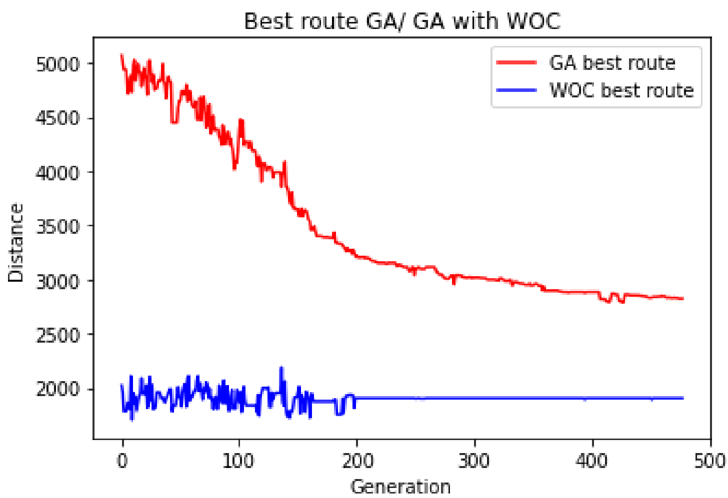
```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)

# Create gif image
create_gif('GA_with_WOC_77.gif')
```

VRP with 1 Depot,96 customers and 10 drivers

In [382]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random97.tsp')
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_97_customers.png')
```





CPU times: user 10.6 s, sys: 12 ms, total: 10.6 s

```
CPU times: user 10.6 s, sys: 12 ms, total: 10.6 s
Wall time: 10.6 s
```

Best cost without WOC

In [383]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 208.45 Total: 2084.49 Generations: 905
Mean: 441.17 Total: 4411.68 Generations: 37
Mean: 290.01 Total: 2900.15 Generations: 477
Mean: 266.72 Total: 2667.18 Generations: 426
Mean: 348.26 Total: 3482.61 Generations: 201
Mean: 256.33 Total: 2563.34 Generations: 510
2 loops, best of 3: 6.52 s per loop
```

Best cost with WOC

In [384]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 173.75 Total: 1737.55 Generations: 177
Mean: 176.0 Total: 1760.01 Generations: 73
Mean: 167.64 Total: 1676.38 Generations: 10
Mean: 175.97 Total: 1759.71 Generations: 82
Mean: 176.6 Total: 1765.96 Generations: 2
Mean: 176.6 Total: 1765.96 Generations: 2
2 loops, best of 3: 1.56 s per loop
```

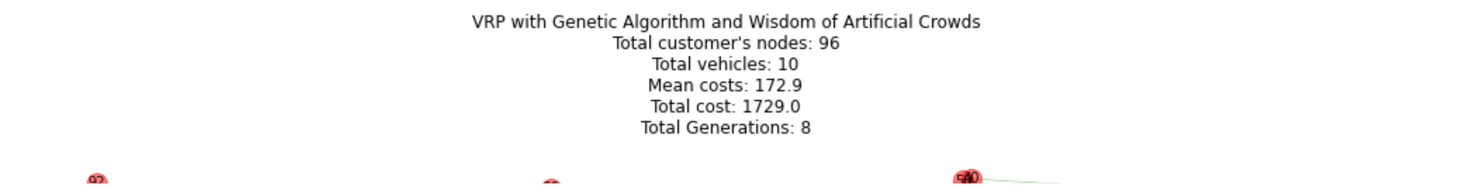
In [411]:

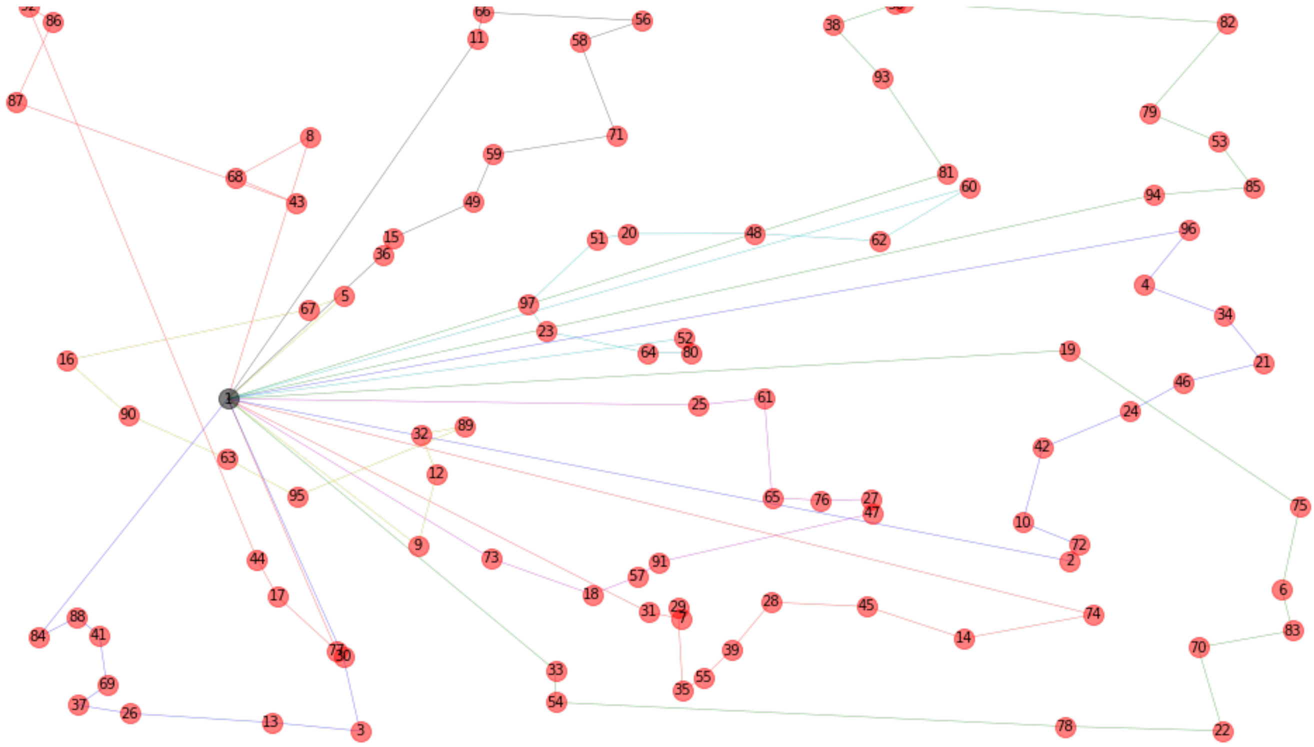
```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 31, 7, 29, 35, 55, 39, 28, 45, 14, 74, 1], [1, 2, 72, 10, 42, 24, 46,
21, 34, 4, 96, 1], [1, 94, 85, 53, 79, 82, 40, 50, 38, 93, 81, 1], [1, 60, 62, 48, 20, 51,
97, 23, 64, 80, 52, 1], [1, 25, 61, 65, 76, 27, 47, 91, 57, 18, 73, 1], [1, 9, 12, 32, 89,
95, 63, 90, 16, 67, 5, 1], [1, 36, 15, 49, 59, 71, 58, 56, 66, 11, 1], [1, 8, 68, 43, 87,
86, 92, 44, 17, 77, 1], [1, 30, 3, 13, 26, 37, 69, 41, 88, 84, 1], [1, 33, 54, 78, 22, 70,
83, 6, 75, 19, 1]]
Total routes: 10
Individual routes cost: [172.76, 208.06, 228.63, 160.52, 128.34, 125.72, 137.05, 216.25,
121.41, 230.26]
Mean route cost: 172.9
Total cost: 1729.0
Total number generations: 8
```

In [412]:

```
graph.draw_plot(results[0][-1], weights=False)
```





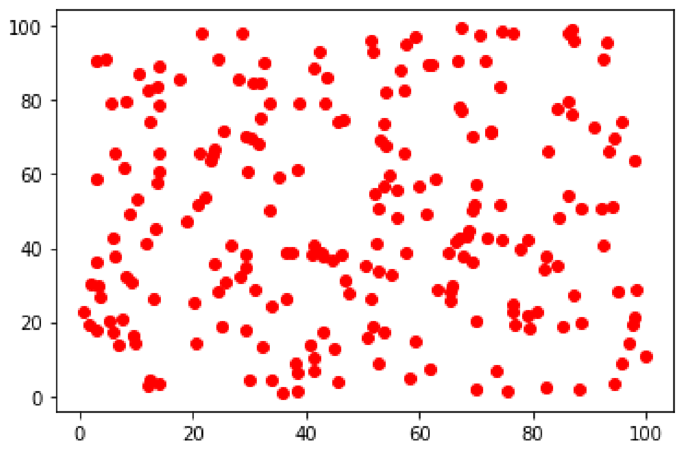
In [ ]:

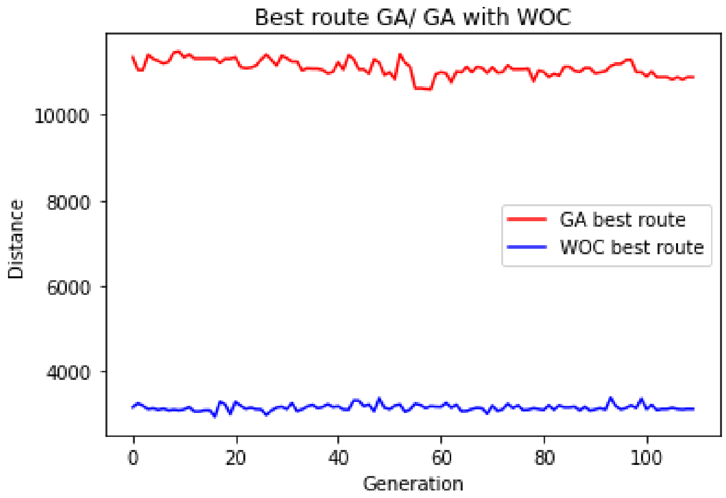
```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)

# Create gif image
create_gif('GA_with_WOC_97.gif')
```

VRP with 1 Depot,221 customers and 22 drivers

In [416]:

```
%%time
# Find besth path by using nearest insertion algorithm
# Instantiate graph
graph = Graph('Random222.tsp')
plot_nodes(graph)

# Run GA with WOC but not include tour to next generation
# with population size 50 and stopping criteria 50 generation without cost improvement
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'guess')
plot_ga_and_woc(results, 'best_routes_ga_and_woc_222_customers.png')
```






```
CPU times: user 12 s, sys: 17.9 ms, total: 12 s
Wall time: 12 s
```

Best cost without WOC

In [417]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = False)
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 456.83 Total: 10050.19 Generations: 145
Mean: 479.32 Total: 10544.99 Generations: 97
Mean: 491.54 Total: 10813.96 Generations: 23
Mean: 472.75 Total: 10400.55 Generations: 145
Mean: 431.93 Total: 9502.49 Generations: 134
Mean: 473.04 Total: 10406.97 Generations: 48
2 loops, best of 3: 7.71 s per loop
```

Best cost with WOC

In [418]:

```
%%timeit -n 2
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'Mean: {results[0][-1][2]} Total: {results[0][-1][3]} Generations: {results[0][-1
][4]}')
```

```
Mean: 138.53 Total: 3047.69 Generations: 3
Mean: 139.39 Total: 3066.65 Generations: 17
Mean: 133.52 Total: 2937.52 Generations: 2
Mean: 136.79 Total: 3009.45 Generations: 2
Mean: 138.73 Total: 3052.12 Generations: 2
Mean: 138.91 Total: 3055.98 Generations: 115
2 loops, best of 3: 6.55 s per loop
```

In [420]:

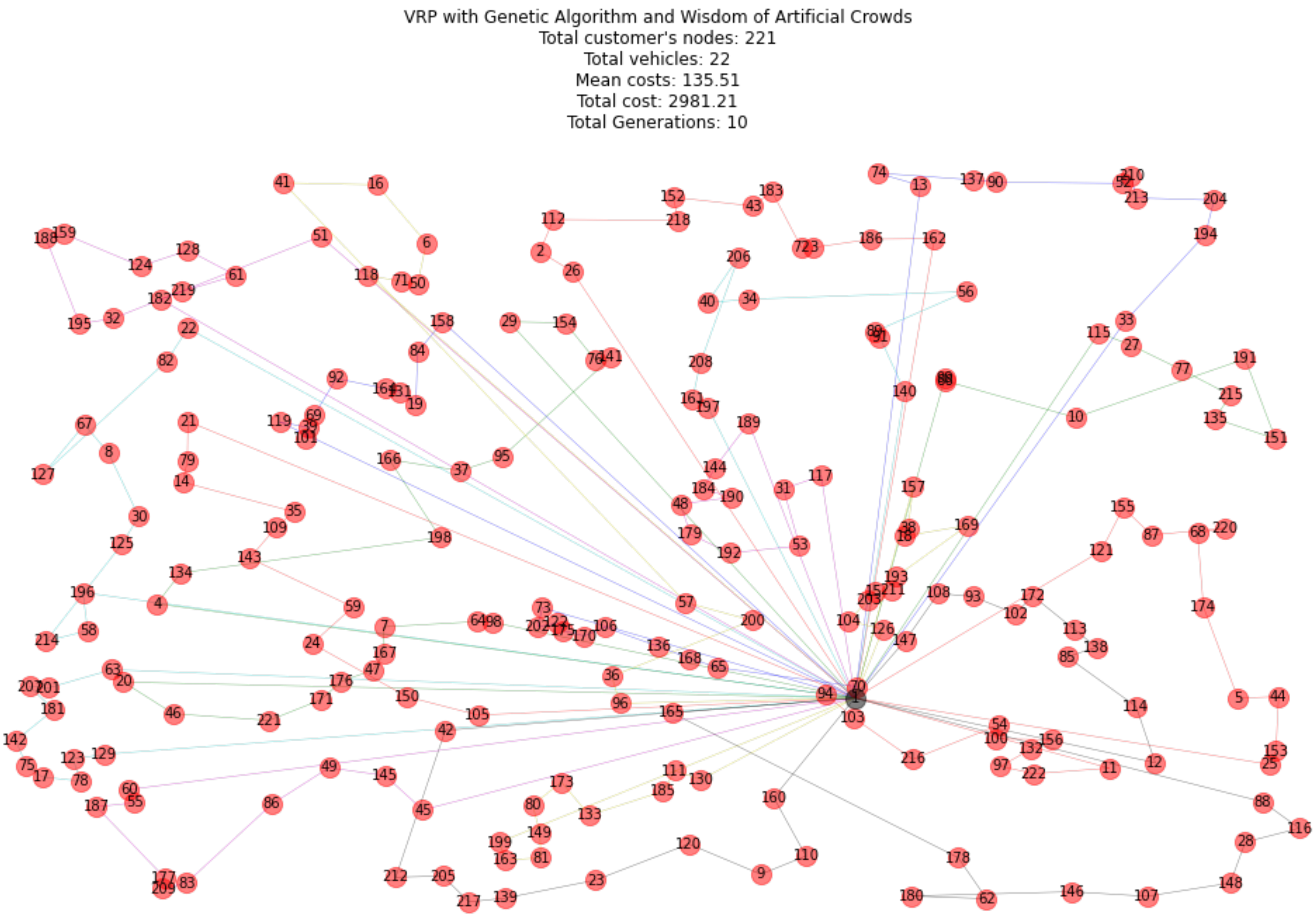
```
results = graph.genetic_algorithm(pop_size=50, term_count=50, woc = 'include')
print(f'The best route: {results[0][-1][0]}',\
      f'Total routes: {len(results[0][-1][0])}',\
      f'Individual routes cost: {results[0][-1][1]}',\
      f'Mean route cost: {results[0][-1][2]}',\
      f'Total cost: {results[0][-1][3]}',\
      f'Total number generations: {results[0][-1][4]}', sep='\n')
```

```
The best route: [[1, 26, 2, 112, 218, 152, 43, 183, 72, 3, 186, 162, 1], [1, 13, 74, 137,
90, 52, 210, 213, 204, 194, 33, 1], [1, 115, 27, 77, 215, 135, 151, 191, 10, 66, 99, 1],
[1, 140, 91, 89, 56, 34, 40, 206, 208, 161, 197, 1], [1, 189, 144, 184, 190, 48, 179, 192
, 53, 31, 117, 1], [1, 157, 38, 18, 169, 193, 211, 15, 203, 104, 126, 1], [1, 147, 108, 9
```

```
3, 102, 172, 113, 138, 85, 114, 12, 1], [1, 11, 222, 97, 132, 156, 100, 54, 216, 103, 94,
1], [1, 70, 65, 168, 136, 106, 170, 175, 122, 202, 73, 1], [1, 98, 64, 7, 167, 47, 176, 1
71, 221, 46, 20, 1], [1, 63, 201, 207, 181, 142, 75, 17, 78, 123, 129, 1], [1, 60, 55, 18
7, 177, 209, 83, 86, 49, 145, 45, 1], [1, 199, 163, 81, 149, 80, 173, 133, 185, 111, 130,
1], [1, 160, 110, 9, 120, 23, 139, 217, 205, 212, 42, 1], [1, 105, 150, 24, 59, 143, 109,
35, 14, 79, 21, 1], [1, 119, 39, 101, 69, 92, 164, 131, 19, 84, 158, 1], [1, 29, 154, 76,
141, 95, 37, 166, 198, 134, 4, 1], [1, 196, 58, 214, 125, 30, 8, 67, 127, 82, 22, 1], [1,
182, 219, 61, 128, 124, 159, 188, 195, 32, 51, 1], [1, 118, 71, 50, 6, 16, 41, 57, 200, 3
6, 96, 1], [1, 165, 178, 62, 180, 146, 107, 148, 28, 116, 88, 1], [1, 25, 153, 44, 5, 174
, 68, 220, 87, 155, 121, 1]]
Total routes: 22
Individual routes cost: [172.0, 175.6, 156.84, 147.77, 109.24, 69.63, 77.25, 63.46, 59.03
, 124.68, 145.62, 142.41, 82.84, 113.06, 150.21, 149.34, 185.64, 203.43, 209.67, 199.64,
129.74, 114.11]
Mean route cost: 135.51
Total cost: 2981.21
Total number generations: 10
```

In [421]:

```
graph.draw_plot(results[0][-1], weights=False)
```



In [ ]:

```
# Plot all paths in nearest insertion algorithm
# Reset index for file name and delete all images from graph directory
graph.reset_file_index()

for i in range(len(results[0])):
    graph.draw_plot(results[0][i], weights=False)

# Create gif image
create_gif('GA_with_WOC_222.gif')
```

In [423]:

```
# # Create pdf from Colab
```

```
# !apt-get install texlive texlive-xetex texlive-latex-extra pandoc > /dev/null 2>&1
# !pip install pypandoc > /dev/null 2>&1
```

In [424]:

```
# from google.colab import drive
# drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

In [446]:

```
!cp drive/My\ Drive/Colab\ Notebooks/project6.ipynb ./
```

In [447]:

```
!jupyter nbconvert --to PDF "project6.ipynb" > /dev/null 2>&1
```

In [448]:

```
!cp * drive/My\ Drive/Project6/
```

```
cp: -r not specified; omitting directory 'drive'
cp: -r not specified; omitting directory 'graphs'
cp: -r not specified; omitting directory 'sample_data'
```