Optimizing Garbage Pick-up in Wahoo, NE

Andrew Pace
apace@unomaha.edu

Follow this and additional works at: https://digitalcommons.unomaha.edu/university_honors_program

Part of the Operational Research Commons

Recommended Citation
Pace, Andrew, "Optimizing Garbage Pick-up in Wahoo, NE" (2021). Theses/Capstones/Creative Projects. 136.
https://digitalcommons.unomaha.edu/university_honors_program/136

This Dissertation/Thesis is brought to you for free and open access by the University Honors Program at DigitalCommons@UNO. It has been accepted for inclusion in Theses/Capstones/Creative Projects by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.
Optimizing Garbage Pick-up in Wahoo, NE
An Application of the Capacitated Vehicle Routing Problem and Simulated Annealing

Andrew Pace
University Honors Program: Capstone
Project Advisor: Fabio Vitor
May 2021
# Table of Contents

Abstract...................................................................................................................................................... pg. 3

1. Introduction............................................................................................................................................... pg. 4

2. Development............................................................................................................................................ pg. 5

   2.1 Mathematical Model............................................................................................................................. pg. 5

      2.1.1 Parameters.................................................................................................................................. pg. 6
      2.1.2 Decision Variables....................................................................................................................... pg. 6
      2.1.3 Objective Function..................................................................................................................... pg. 7
      2.1.4 Constraints.................................................................................................................................. pg. 8

   2.2 Distance Matrix Generator................................................................................................................ pg. 8

      2.2.1 Addresses.................................................................................................................................... pg. 9
      2.2.2 Geocoding.................................................................................................................................... pg. 9
      2.2.3 Distance Matrix.......................................................................................................................... pg. 9
      2.2.4 Generator.................................................................................................................................... pg. 10

   2.3 Heuristic Algorithm............................................................................................................................ pg. 10

      2.3.1 Pseudo Code............................................................................................................................... pg. 11
      2.3.2 Initialization............................................................................................................................... pg. 12
      2.3.3 Transformations......................................................................................................................... pg. 15

         2.3.3.1 Move ....................................................................................................................................... pg. 15
         2.3.3.2 Replace Highest Average..................................................................................................... pg. 17

      2.3.4 Simulated Annealing.................................................................................................................... pg. 18

3 Results ...................................................................................................................................................... pg. 22

4 Conclusion................................................................................................................................................ pg. 24

5 Future Work ............................................................................................................................................. pg. 25

References..................................................................................................................................................... pg. 26
Abstract

This project seeks to optimize the garbage pickup routes of Roadrunner Transportation in Wahoo, Nebraska using operations research and optimization tools. Currently, the company has no formal procedure for determining these routes. Roadrunner Transportation would like to expand their operation and to do so in the most efficient manner. To determine efficient routes, the Capacitated Vehicle Routing Problem (CVRP) is applied, and a solution to this problem is obtained by implementing a simulated annealing heuristic algorithm. Results show that a total distance of approximately 70 miles can be obtained by this implementation using a small subset of the data provided by Roadrunner Transportation. By determining the optimized set of routes, Roadrunner Transportation will save on operating costs. These savings may then be used to grow their operation to provide more and better services to their customers.
1. **Introduction**

Roadrunner Transportation LLC is responsible for a portion of waste removal in Wahoo, Nebraska. The company would like to find whether their current set of routes are inefficient and if they are spending more resources on removal than is necessary. Furthermore, Roadrunner Transportation would like to expand their services to provide more and better services for customers, and any resources saved by the development and implementation of an optimized set of routes can then be reinvested into this expansion. Ideally, the company would like to determine optimal routes so they can invest with foresight.

The primary objective of this project is to determine an improved solution for Roadrunner’s current fleet. Peripheral objectives include determining if their fleet is of optimal size for their current level of operation and to find an optimal route based on their current number of customers and the capacity of their vehicles.

Roadrunner Transportation’s problem is modeled in this project using the Capacitated Vehicle Routing Problem (CVRP). Provided with a fleet of vehicles, a set of customers (nodes), and their demand, the objective of the CVRP is to minimize the total cost of delivering to every customer and satisfying demand. This is performed by determining the optimal set of routes to be traversed by the fleet of vehicles while preventing each vehicle’s capacity from being exceeded. For this application of the CVRP, some assumptions are made:

- Each vehicle is identical with respect to its capacity;
- The weight of garbage being picked up at each stop is the same;
- Vehicles are not allowed to return to a node once it has been visited.

The CVRP is known to be NP-hard. That is, given the combinatorial nature of the problem, finding an optimal solution may require a prohibitive amount of time. Due to this limitation,
determining an optimal solution is often forgone in favor of finding a sufficiently good solution through the implementation of a heuristic algorithm. A simulated annealing heuristic algorithm is implemented in this project to find such a solution. This algorithm functions by comparing solutions using a local neighborhood search and keeping the best solutions over the course of many iterations. To prevent itself from becoming stuck in a local optimum, worse solutions are compared to the Boltzmann distribution (see a description of the Boltzmann distribution in Section 2.3.4). Results obtained by implementing this simulated annealing heuristic algorithm on a small subset of data containing 24 customers show a total distance of approximately 70 miles. This project is implemented in Python® 3.9.5.

The remainder of this report is organized as follows. Section 2 discusses the development of this project. This includes the modeling of Roadrunner Transportation’s problem as a CVRP, the generation of distance matrices, the implementation of the simulated annealing heuristic algorithm. Section 3 reports the results obtained with the implementation. Sections 4 concludes the project and Section 5 discuss potential ideas for future work.

2. Development

Three key components were developed and/or implemented for applying the CVRP to waste removal in Wahoo, Nebraska: a mathematical model, a distance matrix generator, and a simulated annealing heuristic algorithm. All three are described in the following sections.

2.1 Mathematical Model

An optimization model is a mathematical structure where choices are first parameterized then represented by decision variables. These decision variables are then used to construct an objective
function which is to be optimized. The set of feasible solutions is then restricted by constraints which specify which types of choices can and cannot be made. The three components of a CVRP optimization model are described in the next sections.

2.1.1 Parameters

Parameters are the data pertaining to the problem. In this model, there are four parameters: the nodes (customers) which trucks travel to and from, the distances between each pair of nodes, the demand of each customer, and the capacity of the trucks. Initially, the addresses of houses to be picked up from were considered as nodes, but it soon became evident that this would be infeasible since multiple stops were often on the same street. It was then considered that the intersections themselves could serve as nodes. This was much more feasible as the streets between intersections could naturally be understood as edges between these nodes. Each edge could then be weighted proportionate to the number of customers on it.

Parameters

- Set of nodes is defined by \( S = \{0,1,...,122\} \).
- Distance between each pair of nodes is defined by \( c_{ij} \forall i, j \in S \).
- Demand along each set of nodes is defined as \( b_{ij} \forall i, j \in S \setminus \{0\} \).
- Capacity of each vehicle is defined as \( d = 10,000 \).

2.1.2 Decision Variables

Decision Variables represent the choices that need to be made concerning the allocation of resources. These may take several forms. In this model, binary, integer, and real variables are utilized. Binary variables serve effectively as an on/off switch, represented as either a 1 or a 0.
Integer variables serve as whole number values as is the case when considering whole units of resources. Real variables allow for the consideration of values between integers, that is, values with decimal units. Since the demand and capacity is measured in weight, it is imperative that non-integer values be accounted for.

In this model, the road between any two intersections \(i\) and \(j\) is understood as an edge and if that road is selected as an element for a path, the variable assigned to it, defined as \(x_{ij}\), is designated as a 1; otherwise as a 0. The number of vehicles is accounted for with an integer variable \(p\). Finally, the current load within each truck is tracked with variable \(u\) while the truck’s route is also being created.

**Decision Variables**

- Edge selected (binary) is defined as \(x_{ij} \in \{0,1\}\) \(\forall i,j \in S\).
- Number of vehicles (integer) is defined as \(p \in \mathbb{Z}_+\).
- Load of the truck (real) after visiting each customer is defined by \(u_i \in \mathbb{R}_+\) \(\forall i \in S \setminus \{0\}\).

**2.1.3 Objective Function**

An objective function considers the provided parameters and decision variables and organizes them into a single mathematical function. The goal is to determine the maximum (or minimum depending on the desired output) value along this function. For this problem, the goal is to minimize the total cumulative distance travelled by all the garbage trucks in Roadrunner Transportation’s fleet. The function for this will be the sum of all selected edges times their corresponding distance (1). That is, if the distance from node \(i\) to node \(j\), as denoted by \(c_{ij}\), is traversed, \(x_{ij}\) equals 1 and thus that distance is added to the cumulative total. Otherwise, \(x_{ij}\) equals 0 and the distance is not added to the cumulative total.
Objective Function

- \( \min z = \sum_{i \in S} \sum_{j \in S} c_{ij} x_{ij} \). \hspace{1cm} (1) \]

2.1.4 Constraints

The objective function is then subject to a series of constraints which provide rules that the objective function must adhere. For this problem, the objective function must consider that each node must be visited only once \((2)\), and each node must be departed from only once \((3)\). Additionally, the depot must be the first and last node within each route \((4)\)(\(5)\). Finally, the capacity of each truck must not be exceeded within its route and the routes must be connected \((6)\)(\(7)\).

Constraints

- \( \sum_{i \in S} x_{ij} = 1 \quad \forall j \in S \setminus \{0\} \) \hspace{1cm} (2)
- \( \sum_{j \in S} x_{ij} = 1 \quad \forall i \in S \setminus \{0\} \) \hspace{1cm} (3)
- \( \sum_{i \in S} x_{i0} = p \) \hspace{1cm} (4)
- \( \sum_{j \in S} x_{0j} = p \) \hspace{1cm} (5)
- \( u_j - u_i \geq b_{ij} - d(1 - x_{ij}) \quad \forall i, j \in S \setminus \{0\}, \ i \neq j, \ b_{ij} + b_{i+1,j+1} \leq d \) \hspace{1cm} (6)
- \( 0 \leq u_i \leq d - b_{ij} \quad \forall i, j \in S \setminus \{0\} \) \hspace{1cm} (7)

2.2 Distance Matrix Generator

A distance matrix lists values in a 2-dimensional array. The distance matrix corresponds to the parameter \( c_{ij} \). That is, the distance between each pair of customers \( i \) and \( j \). To generate a distance matrix, the OpenStreetMap\textsuperscript{®} API and Microsoft Bing\textsuperscript{®} API are used. Intersection addresses are imputed and from this, the geocodes are retrieved from the OpenStreetMap\textsuperscript{®} API. These geocodes are then provided to the Microsoft Bing\textsuperscript{®} API and used to determine the distance
between each intersection and these values are then arranged into a distance matrix. This portion was developed and provided by Isidore Sossa [2]. There are four parts to the generation of this distance matrix, each of which is coded in Python: addresses, geocoding, distance matrix, and generator.

2.2.1 Addresses

The first part interprets the given addresses as they appear in string form and appropriately formats them. This is accomplished by designating a class, Address, which has the following attributes: street, city, state, country, zip code, latitude, longitude, and info. Each of these attributes are then given basic functions which will facilitate the retrieval of desired information from the provided addresses. Similar functions are designated for the retrieval of the latitude and longitude coordinates which correspond to the given addresses. These coordinates are kept pairwise as strings to facilitate their later use when plotting on Microsoft Bing® maps.

2.2.2 Geocoding

The second part in the process of generating a distance matrix is the retrieval of the geocodes themselves. A class, Geocoding, is created with two primary attributes: get_geocode and get_geocodes. These two functions both retrieve geocode pairs with the simple distinction that get_geocode takes a single address as input and get_geocodes takes a list of addresses.

2.2.3 Distance Matrix

Expanding on the infrastructure established by the Address and Geocoding classes, a third class DistanceMatrix is created which allows for the construction of distance matrices and duration matrices. The first attribute under this new class is get_matrix. This attribute takes a dictionary of
matrix locations and a dictionary of geocodes to return distance and duration matrices. The next attribute of this class `DistanceMatrix` is `request_matrix`. This function takes a list of geocodes, an API key, the method of travel, and size as arguments and requests the distance and time values associated with travelling between any pair of nodes. The function then returns a distance matrix, duration matrix, and the status code. If the status code returned is 200, the request is successful.

2.2.4 Generator

Finally, the three previous classes are utilized together to generate distance matrices. First these classes are imported to make use of their attributes. A list of addresses is then provided which will run through the program. The OpenStreetMap® API key that allows for the retrieval of these geocodes is imported and the number of successful and failed geocode retrievals are reported alongside the API call status. Three new variables, `distance_map`, `duration_map`, and `response` are defined with respect to the geocode dictionary keys, the API key, mode of travel (in this case driving), and the size of the matrix. The distance and duration matrices are then retrieved by calling the `get_matrix` function from the `DistanceMatrix` file. These matrices are then returned as data frames. The desired matrix is then exported to a Microsoft Excel® spreadsheet which is imported into the heuristic algorithm.

2.3 Heuristic Algorithm

The heuristic method implemented in this project is called simulated annealing. This method attempts to model itself after the technique of annealing used in metallurgy. In this process, solids are heated past their melting point which allows the material’s atoms to move freely and thus the material’s state is subject to seemingly random change. As it cools, the atoms stick
together and this will often result in atomic structures which contain less internal energy than before and thus, form a higher quality material. By varying the rates of cooling, different properties can be observed. This process can be mathematically modeled and applied to problems that seek to minimize the total cost associated with visiting a series of nodes, such as with the CVRP.

2.3.1 Pseudocode

The implementation developed for this project is based on the work of H. Harmanani, D. Azar, N. Heal, and W. Keirouz titled “A Simulated Annealing Algorithm for The Capacitated Vehicle Routing Problem” [3]. Figure 1 shows the pseudocode for generating an initial solution to the problem while Figure 2 presents the simulated annealing pseudocode. This implementation was coded in Python® 3.9.5.

**Source:** A Simulated Annealing Algorithm for The Capacitated Vehicle Routing Problem [3].

![Initial Solution Pseudocode](image)

This method has several steps. First, a random feasible solution is generated. Next, a second random feasible solution is generated and compared to the first. If the new solution is better than the previous, the new solution replaces the old as the current best solution. Otherwise, the new solution is compared with the Boltzmann distribution. If the required conditions are met, the new solution is kept even though it is worse. This allows for a greater likelihood of approaching the global optimum and serves to prevent returning a local optimum. This distribution requires a
temperature value $T$. This value is determined arbitrarily at the beginning and is then updated after each iteration. The greater the difference between the two solutions, the less likely the new solution is accepted. This process is then repeated until either the maximum number of iterations is reached, the temperature value nears zero, or the current solution fails to change after too many iterations.

*Source: A Simulated Annealing Algorithm for The Capacitated Vehicle Routing Problem [3].*

```plaintext
Annealing(CVRP) {
    $S_0 = \text{Initial solution}$
    $\alpha = 0.99$  // Temperature reduction multiplier
    $\beta = 1.05$  // Iteration multiplier
    $M_0 = 5$  // Time until next parameter update
    BestS = Best solution
    $T = 5000$
    CurrentS = $S_0$
    CurrentCost = Cost(CurrentS)
    BestCost = Cost(BestS)
    Time = 0
    do {
        $M = M_0$
        do {
            NewS = Neighbor(CurrentS);
            NewCost = Cost(NewS)
            $\Delta_{Cost} = \text{NewCost} - \text{CurrentCost}$
            if ($\Delta_{Cost} < 0$)
                CurrentS = NewS
                CurrentCost = Cost(CurrentS);
            if (NewCost < BestCost) then
                BestS = NewS
                BestCost = Cost(BestS)
            else if (Random < $e^{-\frac{\Delta_{Cost}}{T}}$) then
                CurrentS = NewS
                CurrentCost = Cost(CurrentS);
            $M = M - 1$
        } while ($M \geq 1$)
        Time = Time + $M_0$
        $T = \alpha * T$
        $M_0 = \beta * M_0$
    } while (Time > MaxTime and $T > 0.001$)
    Return(BestS);
}
```

*Figure 2: Simulated Annealing Pseudocode.*

### 2.3.2 Initialization

Three packages are used in the implementation of the simulated annealing heuristic: *pandas*, to facilitate data frame manipulation; *random*, to randomly generate values when needed; and *math*, to successfully utilize the Boltzmann distribution. Several other values are defined before determining an initial solution. The distance matrix is imported from a Microsoft Excel® spreadsheet and converted into a *pandas* data frame. The number of nodes is determined by counting the number of columns within this data frame. Each node is then assigned a number, and
for simplicity’s sake, this number also corresponds to each column number within the Microsoft Excel® spreadsheet and data frame. An average weight per stop is assigned, as well as the vehicle capacity. The distance matrix used in the description of this implementation consists of 24 locations within Wahoo (customers). Since this is a limited supply of desired nodes, for the sake of demonstration, the vehicle capacity and weight per stop are arbitrarily low. The code for this is displayed in Figure 3.

```python
import pandas as pd
import random
import math

file = "distance_matrix.xlsx"
df = pd.read_excel(file)
number_of_nodes = len(list(df.columns))
nodes = [i for i in range(number_of_nodes)]
weight_per_stop = 10.5
vehicle_cap = 75
```

**Figure 3: Initialization.**

The first function is displayed in Figure 4. This function generates an initial solution which will be used by the following functions and algorithm. This function takes any list of integers (nodes) as an argument and begins by defining variables and establishing lists. These variables consist of the number of vehicles, vehicle routes, route, and solution. The number of vehicles begins at one and the input list is redefined to exclude the first node. Within the context of this code, the first node is intended to be the depot node which every vehicle must depart from. Two lists are created, one to monitor each unique route and another to monitor the set of routes.

The bulk of this function is a while loop. Within this loop, an element is chosen at random from the list of nodes. If the vehicle’s capacity is not exceeded by the addition of this node to its route, the node is appended to the route list and subsequently removed from the list of available nodes. The vehicle’s capacity is adjusted to account for this stop. If the vehicle’s capacity is exceeded by the addition of this node, the current route is appended to the list of routes, the route
list is reset, the element is removed from the nodes list and then appended to a new route. The vehicle capacity is likewise reset, and the number of vehicles is increased by one to account for the additional truck required to run the new route. This loop terminates once there are no remaining nodes left to be assigned to a route. The function then returns an initial list of possible routes. Finally, once all nodes have been assigned to a route, the list of initial routes is displayed.

```python
def initial_sol(sol):
    global vehicle_cap
    num_vehicles = 1
    vehicle_routes = []
    route = [0]
    sol = sol[1:]
    while len(sol) > 0:
        element = random.choice(sol)
        if vehicle_cap - weight_per_stop > 0:
            route.append(element)
            sol.remove(element)
            vehicle_cap = vehicle_cap - weight_per_stop
        else:
            vehicle_routes.append(route)
            sol.remove(element)
            vehicle_cap = 75
            route = [0, element]
            vehicle_cap = vehicle_cap - weight_per_stop
            num_vehicles += 1
    vehicle_cap = 75
    vehicle_routes.append(route)
    print("Vehicle routes: ", vehicle_routes)
    return vehicle_routes
```

Figure 4: Initial Solution.

Another function is needed to calculate the total cumulative distance travelled by the vehicles. Shown in Figure 5, this distance function takes a list of lists, in this case the previously generated routes, as an argument. For each node in a route, the distance between itself and the following node is retrieved from the distance matrix data frame, using the nodes’ numbers as the column and row values. The routes have the distance between each node calculated and added to the cumulative distance, including the distance travelled when returning to the depot. Once every route’s distance has been determined, the total cumulative distance is returned.
2.3.3 **Transformations**

Two transformations are used in this implementation of the simulated annealing algorithm. Both are called *move* and *replace highest average*, respectively. These transformations are the functions used to augment the current solution as the algorithm iterates.

2.3.4.1 **Move**

The first, *move*, splits each route into a series of node pairs. The distance value associated with the distance between the nodes of these pairs are calculated and listed. Each pair is then zipped to its corresponding distance and the three pairs with the shortest distance are put into a list of nodes to be excluded as shown in Figure 6.

A subsequent list of nodes which excludes the previous pairs, labeled *beta nodes*, is designated and from *beta nodes*, three nodes are randomly selected and placed into another separate list called *random nodes*. These random nodes are then removed from the original set of routes. From these routes without the excluded nodes, a random route is selected. If the addition of one of these random nodes to this route does not exceed the weight capacity of the vehicle, the node is successfully added. Once the random nodes have all been reassigned, the new rearranged

```python
def distance(routes):
    dist = 0
    for path in routes:
        for i in range(len(path) - 1):
            dist += df[path[i]][path[i + 1]]
        if len(path) > 0:
            dist += df[path[len(path) - 1]][path[0]]
        else:
            pass
    return dist
```

Figure 5: Distance.
routes are returned. Figure 7 displays the second half of the move function. The number of excluded nodes and random nodes selected can be altered to accommodate larger or smaller data sets.

```python
def move(routes):
    pairs = []
    d_list = []
    for path in routes:
        for i in range(len(path) - 1):
            pairs = pairs + [[path[i], path[i + 1]]]
            pairs = pairs + [[path[len(path) - 1], path[0]]]
        for pair in pairs:
            d = df[pair[0]][pair[1]]
            d_list.append(d)
    s = sorted(zip(d_list, pairs))[:3]
    s_prime = list(zip(*s))
    excluded_nodes = []
    for i in s_prime[1]:
        excluded_nodes.append(i[1])
    if int(0) not in excluded_nodes:
        excluded_nodes.append(0)
    else:
        pass
    beta_nodes = [i for i in nodes if i not in excluded_nodes]
    random_nodes = []
    while len(random_nodes) < 3:
        r = random.choice(beta_nodes)
        if r not in random_nodes:
            random_nodes.append(r)
        else:
            pass
    routes_ohne_random_nodes = []
    for path in routes:
        delta = [i for i in path if i not in random_nodes]
        routes_ohne_random_nodes.append(delta)
    r = random.choice(routes_ohne_random_nodes)
    while len(random_nodes) > 0:
        if weight_per_stop * len(r) > vehicle_cap:
            r = random.choice(routes_ohne_random_nodes)
        else:
            x = random_nodes.pop()
            r.append(x)
    rearranged_routes = routes_ohne_random_nodes
    return rearranged_routes
```

Figure 6: Move Function Part 1.

Figure 7: Move Function Part 2.
2.3.4.2 Replace Highest Average

The second transformation, *replace highest average*, shown in Figure 8, works similarly to the *move* transformation. Instead of withholding the node pairs with the shortest distance and rearranging the routes, it will specifically target three-node subsets with the greatest average distance and break these nodes up. This transformation first divides each initial route into a series of sub-routes, each containing three nodes, called vertices.

```python
def replace_highest_average(routes):
    vertices = []
    for path in routes:
        for i in range(len(path) - 1):
            vertices = vertices + [path[i - 1], path[i], path[i + 1]]
        vertices = vertices + [path[len(path) - 2], path[len(path) - 1], path[0]]
        vertices = [i for i in vertices if i[1] != 0]
        d_list = []
        averages = []
        for vertice in vertices:
            d = df[vertice[0]][vertice[1]] + df[vertice[1]][vertice[2]]
            d_list.append(d)
        for i in d_list:
            avg = i / 2
            averages.append(avg)
```

*Figure 8: Replace Highest Average Function Part 1.*

With each vertex constructed, their average distances are calculated, Figure 9. That is, the mean value of the distances from the central node to its neighbors is determined. Each vertex has its distance values pulled from the distance matrix data frame and these values are summed together and added to the list of distances. Then each of these distances is divided by 2 to determine the average of the distances travelled to and from each central node in the vertices.

```python
d_list = []
averages = []
for vertice in vertices:
    d = df[vertice[0]][vertice[1]] + df[vertice[1]][vertice[2]]
    d_list.append(d)
for i in d_list:
    avg = i / 2
    averages.append(avg)
```

*Figure 9: Replace Highest Average Function Part 2.*

These values are then zipped and sorted, and the three greatest values are returned. The central nodes in the vertices corresponding to the greatest average distances are removed from their original routes. The routes without these nodes and the nodes themselves are stored in separate lists: *excluded vertex points* and *routes ohne vertices*. This is displayed in Figure 10.
As shown in Figure 11, once each of these nodes are removed from their original routes, they are assigned to a random new route if that route can accommodate the additional weight. Once all these nodes are reassigned, the new rearranged routes are returned.

```python
s = sorted(zip(averages, vertices), reverse=True)[:3]
s_prime = list(zip(*s))

excluded_vertex_points = []
for i in s_prime[1]:
    excluded_vertex_points.append(i[1])

routes_ohne_vertices = []
for route in routes:
    for vertex in excluded_vertex_points:
        if vertex in route:
            route.remove(vertex)
    routes_ohne_vertices.append(route)

r = random.choice(routes_ohne_vertices)
while len(excluded_vertex_points) > 0:
    if weight_per_stop * len(r) > vehicle_cap:
        r = random.choice(routes_ohne_vertices)
    else:
        x = excluded_vertex_points.pop()
        r.append(x)
rearranged_routes = routes_ohne_vertices
return rearranged_routes
```

**Figure 11: Replace Highest Average Function Part 4.**

### 2.3.4 Simulated Annealing

These previous functions and transformations serve as the infrastructure of the simulated annealing algorithm. Within the algorithm, these four components work together over a series of iterations to determine better solutions. Before the algorithm initiates, a total of eight values must be defined. The first four values, temperature $T$, $m_0$, $alpha$, and $beta$, work together to determine the stopping criteria of the algorithm. The latter four values, current solution, current distance, best solution, and best distance keep track of the results after each iteration and eventually the best
set of routes and its cumulative distance is returned. Figure 12 shows the initialization of the simulated annealing algorithm.

```python
def annealing_cvrp():
    T = 1000
    m_0 = 5
    alpha = 0.99
    beta = 0.99
    current_sol = initial_sol(nodes)
    current_distance = distance(current_sol)
    best_sol = current_sol
    best_dist = distance(best_sol)
```

**Figure 12: Simulated Annealing Function Part 1.**

Notice that the functions `initial_sol()` and `distance()` are utilized when defining these parameters. These values are to be updated after each iteration of the algorithm. The algorithm itself consists of a while loop nested within another loop as well as nested if/else statements. The first while loop is set to terminate once the temperature variable crosses a predetermined threshold, in this instance, that threshold is the value 0.05. Under this statement, the number of iterations, \( m \), is set equal to the \( m_0 \) value of 5. The second while loop is set to terminate once the number of iterations drops below 0. Within this nested loop, the replace highest average transformation occurs every iteration while the move transformation occurs in 80% of the iterations.

The first operation under the nested while loop is the use of the replace highest average transformation, followed by the calculation of its distance. The variable denoted as `delta distance` calculates the difference between the current distance, as defined outside of the while loops, and the new distance. The current distance is then set equal to the new distance value.

Under the nested while loop, if the value of `delta distance` is less than zero, that is the distance travelled in the new solution is less than the distance travelled under the current solution, the new solution becomes the current solution. Likewise, the new distance becomes the distance of the current solution. Under this `if` statement lies another, nested, `if` statement. If the new distance is less than the best distance, the new solution becomes the best solution, and the best distance
becomes the distance of the new solution. If the new distance is not less than the best distance, then the nested if statement is disregarded. This portion of the simulated annealing algorithm is depicted in Figure 13.

```
while T > 0.05:
    m = m_0
    while m >= 0:
        new_sol = replace_highest_average(current_sol)
        new_dist = distance(new_sol)
        delta_dist = new_dist - current_distance
        current_distance = new_dist
        if delta_dist < 0:
            current_sol = new_sol
            current_distance = distance(current_sol)
        if new_dist < best_dist:
            best_sol = new_sol
            best_dist = distance(best_sol)
```

**Figure 13: Simulated Annealing Function Part 2.**

If the delta distance value is greater than or equal to zero, the above nested if statements are disregarded and the else statement is considered. Within the else statement is where the Boltzmann distribution is utilized. The Boltzmann distribution is expressed as follows [1] where \( p \) equals a uniform random number between 0 and 1:

\[
p \leq e^{-\frac{|f(x) - f(x')|}{T}}
\]  

The numerator value is represented by the variable delta distance in this project. A uniform random number between 0 and 1 is generated and then compared to the value of the Boltzmann distribution. If the random number is less than that of the value of the Boltzmann distribution, the new solution and distance become the current solution and distance. This function serves to prevent the algorithm from becoming stuck in a local optimum while searching for the global optimum.

Once past this if/else statement, the number of iterations is adjusted, and the move transformation is considered. Since the move transformation is intended to only be utilized in 80%
of iterations, a similar approach is taken as was in the implementation of the Boltzmann distribution. A uniform random number between 0 and 1 is generated, and if this value is less than 0.8, the program proceeds with the *move* transformation. To implement this, the *move* transformation is performed on the current solution and this value becomes the new solution. Its distance is calculated and becomes the new distance. Like the *replace highest average* transformation, the *delta distance* is calculated, and the current distance becomes the new distance. If the value of the *delta distance* is less than zero, the current solution becomes the new solution, and the current distance is replaced by the distance of the current solution. Then if the new distance is less than the best distance, the best solution becomes the new solution, and the best distance is replaced by the distance of the best solution. If the value of the *delta distance* is greater than zero, again, this value is considered for the Boltzmann distribution. Afterwards, the iterations value *m* is adjusted.

At this point, once the nested while loop is terminated, the temperature value begins to decrease. It is multiplied by the *alpha* value provided at the beginning of the algorithm function. Still inside the initial while loop, the *m_0* value is multiplied by the beta value. Once the value of *T* drops below the threshold of 0.05, the algorithm terminates and reports the best solution found over the course of its iterations. The code from Figure 13 is continued in Figure 14 and concludes the implementation. The next section presents some preliminary results obtained.
3. Results

The simulated annealing algorithm was tested using a distance matrix generated from the code described in Section 2.2. Currently, only 24 of the intersections in Wahoo, Nebraska, are utilized to test the implementation. As this is the case, the results are limited but sufficient to validate the code. As displayed in Figure 15, the algorithm demonstrates the capacity to generate solutions which become increasingly better.

Table 1 displays the best solutions as they are recorded by the simulated annealing heuristic algorithm. Each instance begins with a different initial solution. It is observed that the objective values for each instance consistently decrease. These values are recorded in kilometers and as one can see, the best total distance obtained are approximately 70 miles.
To confirm that the algorithm is capable of handling larger distance matrices, a random matrix generator was utilized to create a 250 x 250 matrix with values ranging between 1 and 100. This matrix was then converted to a data frame and its diagonal values replaced with zeros to simulate a distance matrix. To accommodate this larger matrix, some values within the code were
adjusted. These values include the number of elements targeted by the transformations which increased from 3 to 5, as well as the temperature threshold for termination which was increased from 0.05 to 1 to prevent the Boltzmann function from returning a math bound error. The weight capacity was increased to 200 and the weight per stop was decreased to 10. Figure 16 displays the resulting best objective values as they are calculated over the course of each instance’s iterations.

![Figure 16: Objective Values Using a Large Distance Matrix.](image)

4. Conclusion

Applying the Capacitated Vehicle Routing Problem (CVRP) to garbage removal in Wahoo, Nebraska is an effective approach to modeling and optimizing garbage routes. Geocode coordinates corresponding to addresses can be converted into a distance matrix. The simulated annealing heuristic algorithm interprets this data and returns an optimized solution. Additionally, this implementation is dynamic enough to handle matrices of varying sizes and can account for
vehicles of differing capacity. The results gathered thus far suggest that with further development, this project has the potential to find an efficient set of routes for the garbage pickup in Wahoo.

5. Future Work

To further develop this project, there are two areas which require additional work. With respect to the heuristic algorithm, it is imperative that the edges between nodes are properly weighted. Currently it is assumed that the weight of refuse retrieved from each node is the same. Many stops may be on a single road between two intersections. To account for this, further data may be collected from Roadrunner Transportation to determine the number of stops on any given road. This information can then be used to assign weight values to particular edges which will then be considered when accounting for the capacity of each vehicle. Once this is achieved, the algorithm may run with the vehicles’ real capacity of five tons.

The second area concerns the retrieval of geocode values from the Microsoft Bing® API. Currently, when these values are requested, not every address is recognized, which prevents the acquisition of the required geocodes. This may be improved upon by approximating the addresses of these intersections. Requesting the geocodes of an address which resides at the intersections in question can provide such an approximation.
References

