**Genetic Algorithm and the Vehicle Routing Problem**

**Stated Problem:**

The vehicle routing problem (VRP) is an optimization problem with uncertainty (Wang et al., 2017). The goal when solving this problem is to find optimal routes for vehicles visiting multiple locations but it can be complicated by constraints placed on the vehicles or visits to the provided destinations. To find the most optimal solution for the problem each combination of destination visits must be assessed, which is often not practical for the input size. To find a solution that is near optimal in a reasonable amount of time a heuristic solution must be applied to the problem.

**Algorithm Overview:**

For my project I used the genetic algorithm to create a solution to the VRP that can not only be scaled to apply to larger sets of data but can also be customized to meet the requirements presented by unique constraints that arise when the solution is applied to real life situations. The genetic algorithm is a heuristic algorithm that utilizes the theory of natural selection to find the most fit solution to a problem within a given number of generations (Mallawaarachchi, 2017). When applied to the VRP each solution is considered a chromosome, representing the sequence of stops for each vehicle.

The first-generation chromosomes are created by randomly generating a sequence value that represents the truck number the delivery is assigned to and the order in which the location will be visited by that truck. Each chromosome is then assessed a fitness value that consists of the distance traveled by each vehicle and penalty values added if any of the constraints placed on the deliveries are not met. The chromosomes are then ranked by fitness, a low fitness value indicating that the chromosome is a good fit for the problem. Subsequent generations are created by taking a percentage of the most fit chromosomes from the prior generation, creating a percentage of chromosomes from sequence value crossover of randomly selected parents from the previous generation, and a percentage of new chromosomes generated randomly. Through this natural selection process the route that is the most fit for the parameters provided will found.

**Algorithm Pseudocode:**

Create locations:

For each line in the distance table:

Read the line in the csv file

Split data points at commas

Create a location object with the address, zip, and provided distances

Insert the location into the location list

For each location:

If distances are missing from the location’s distance table:

Add missing distances by using the index of the location and the index of the missing distance

Load Package Data:

Number of packages = number of rows in csv file

Create packages hash table with number of packages

Create locations list

For each line in the package list:

Read in the line

Split data points at commas

Create a package object with the package number, delivery location data, delivery deadline, weight, and associated notes

Add the package location to the locations list

If the package is delayed:

Add a delayed package flag

If the package has a deadline:

Compare the deadline time and add an early or late delivery deadline flag

If the package must be on truck 2:

Add truck 2 flag

If the package is a combine delivery:

Add combine delivery flag

Insert the new package into the packages hash table

Locations set = unique values in locations list

Determine the length of the locations set and pass the value to create deliveries

Package Insert:

Create a new package object with the package id, address, delivery deadline, city, zip code, weight, special notes, and delivery status of “At hub”

For x = 0 to the length of the package table:

Bucket = (built in hash of package id + H2 function \* x) % length of the table

If the bucket is empty:

Bucket entry = new package

Return true and the delivery location

Return False and null value

Create Deliveries:

Create the deliveries hash table using the locations set length

For each package in the packages data structure:

If the package is in the delivery table:

Do nothing

Else:

If the package address matches another address in the delivery table:

Add the package to the delivery

If the package has a delay:

Add a delay flag to the delivery

If the package has a deadline:

Add a late/early deadline to the delivery

If the package must be on truck 2:

Add truck 2 flag to delivery

Else:

Insert a new delivery to the delivery table

Assign a location to the delivery

If the package has a delay:

If the delivery has a delay flag:

Do nothing

Else:

Add a delay flag to the delivery

If the package has a deadline:

If the delivery has an early deadline flag:

Do nothing

Else if the delivery has a late deadline flag and the package has an early deadline flag:

Add an early delivery deadline flag

Else:

Add a late/early deadline to the delivery

If the package must be on truck 2:

Add truck 2 flag to delivery

If the package is a combine delivery:

Add combine delivery flag

Add the package to the delivery

Delivery Insert

Create a new delivery object with the address and location

For x = 0 to the length of the delivery table:

Bucket = (built in hash of delivery id + H2 function \* x) % length of the table

If the bucket is empty:

Bucket entry = new delivery

Return

Return False

Generate random route sequences:

If generation 1:

Sequences = 10

If generation > 1:

Sequences = 2

For # of sequences:

Create a route object

For each delivery in the deliveries table:

Generate a sequence value (random floating point between 0 and 1):

If the package has a delivery deadline:

Sequence value = Sequence value / 2

Generate truck number:

If the delivery is delayed:

Sequence value = Sequence value + 3

Else if the delivery must be on truck 2:

Sequence value = Sequence value + 2

Else if the delivery is a combine delivery

Sequence value = Sequence value + 1

Else:

Sequence value = Sequence value + random int between 1 & 3

Assign route sequence to route object:

If sequence value < 2:

Package is assigned to truck 1

Else if sequence value < 3:

Package is assigned to truck 2

Else:

Package is assigned to truck 3

Sort delivery sequence values in ascending order per truck

Add route to current generation list

Calculate fitness of route plans:

OVER\_MILAGE\_PENALTY = 40

OVER\_LIMIT\_PENALTY = 20

MAX\_PACKAGES\_PER\_TRUCK = 16

For each route in the current generation list:

Route fitness = 0

Distance = 0

For each truck in route:

For delivery in route:

Calculate distance to delivery from location distance list

Distance = distance + delivery distance

Route fitness = route fitness + distance

Assign penalties:

Penalty = 0

Time over = 0

Packages over deadline = 0

Weight over = 0

If distance > 140:

Penalty = penalty + OVER\_MILAGE\_PENALTY

For each delivery in the route plan:

If delivery in route plan does not meet time deadline:

Packages over deadline = Packages over deadline + 1

Calculate time over package deadline:

Time over package deadline = delivery time – deadline time

Time over = time over + time over package deadline

Penalty = penalty + (time over \* packages over time) ^ 2

Trucks over package limit = 0

For trucks in route plan:

If truck packages > MAX\_PACKAGES\_PER\_TRUCK:

Trucks over limit = trucks over limit + 1

Penalty = Penalty + Trucks over limit \* OVER\_LIMIT\_PENALTY

Route fitness = route fitness + penalty

Assign route fitness value to route object

Rank current generation routes by fitness:

Sort routes in current generation list in ascending order by the route’s fitness value data member using the built-in sorting function

Mutate current generation with uniform crossover:

For 6 iterations:

Generate two random numbers between 1 and 10(Inclusive):

Random number 1 = parent 1

Random number 2 = parent 2

For each delivery in the sequence:

Generate a random number between 0 and 2:

If the number is 1:

Delivery sequence value = parent 1 delivery sequence value

Else:

Delivery sequence value = parent 2 delivery sequence value

Add route plan to next generation list

Create next generation:

If generation > 20:

Return

Else:

Create empty next generation list

Add the top 2 current generation routes (min fitness value) to next gen list

Call “mutate current generation with uniform crossover”

Clear current generation list

Current generation list = next generation list

Call “generate random route sequences” passing in the generation #

Call “rank current generation routes by fitness”

Generation = generation + 1

Call “create next generation” passing in the generation #

Determine route:

Best fit route = top ranked route in the current generation list

For each truck in the route:

For each delivery in the truck:

Assign the delivery to the truck’s delivery list

**Programming Environment:**

For the creation of the program, I used an Apple MacBook Pro running macOS Monterey Version 12.5.1. The program was created using Visual Studio Code version 1.53.0 and Python 3.8.9.

**Space-Time Complexity Evaluation:**

Operation time of the genetic algorithm will vary depending on the size of the location and package data. Additionally finding a solution to meet the provided criteria may require more generations to be generated if additional constraints are added to the deliveries or vehicles. An analysis of the time and space used for this implementation will be broken down by methods used by the algorithm.

* Create locations – For each location provided in the attached file a constant number of operations are performed to create each location object and assign data values. One half of the distance values are missing from the provided table and must be looked up and assigned to the missing list values. This additional lookup and assignment increased the run time of the method to **O(*N2*)** but can be decreased to **O(N)** by providing the missing distances in the table, with N being the number of locations. The space needed for this method is **O(N)** with N being the number of locations used.
* Load package data into hash table – For each package provided in the attached file a constant number of operations are performed to create the package object and assign values with a run time of **O(N).** Packages are inserted into a custom double hash table using an insert function that utilizes the package ID as the key. Because package IDs are sequential there are no collisions when inserting the provided packages into the table, achieving an insert time of **O(1)** for each package. The runtime of this method is therefore **O(N)** with N being the number of packages. The space for the double hash table is allocated once the number of lines in the package file is determined making the space complexity **O(N)** with N being the number of packages.
* Create Deliveries – To create deliveries each package in the package table is accessed and the delivery table is searched for a matching location. In the worst case each package is being delivered to a different address, so each package is accessed, and all subsequent deliveries are compared yielding a runtime of **O(*N2*)**. The best-case runtime for this method is **O(N)** in the case that all packages are being delivered to the same location and a constant number of operations are performed for each package. Deliveries are entered into a custom double hash table with sequential ID numbers generated and used as the delivery key, this results in a perfect hash table with no collisions and inserts are performed in **O(1)**. The space for the double hash table is allocated once the number of unique locations that will be visited is determined from the package insert function, making the space complexity **O(N)** with N being the number of deliveries.
* Generate random route sequences- For a constant number of sequences each delivery in the delivery table is accessed and assigned a sequence value. The list of sequence values is then sorted in place and assigned to a route object. All operations performed in this method are constant per delivery yielding a runtime of **O(N)** with N being the number of deliveries. Each generation a constant number of routes are generated making the space complexity **O(1).**
* Calculate fitness of route plans- For a constant number of sequences, each delivery is accessed, and a constant number of operations are performed to assess the distance traveled from one delivery to the next. This section of the method operates in **O(N)** with N being the number of deliveries in the delivery table. For a constant number of sequences and a constant number of trucks the number of packages is calculated and compared to the maximum allowed. This operation is performed in **O(1)** as there is always a constant number of trucks and sequences. For each delivery, if there is a deadline the delivery time and deadline are compared. In the worst case each delivery has a deadline, and this portion of the method operates in **O(N)** with N being the number of deliveries. This gives an overall runtime for this method of **O(N)** with N being the number of deliveries. For each constant number of routes one value is generated and stored making the space complexity **O(1).**
* Rank current generation routes by fitness- For each route the fitness value is accessed and used to sort the routes in place. There is always a constant number of routes per generation so this method operates in **O(1)**. All routes are sorted in place and do not require any additional storage to be allocated.
* Mutate current generation with uniform crossover – For a constant number of sequences two random parents are assigned and a value for each delivery is assigned. After the values are assigned, the sequence is added to the route list. This method operates in **O(N)** with N being the number of deliveries in the delivery table. Each generation a constant number of values are generated from crossover making the space complexity **O(1).**
* Create next generation – For a constant number of generations, four functions are called with a runtimes of: O(N), O(N), O(1), and O(N). This gives this method a runtime of **O(N)** with N being the number of deliveries.Each generation uses the space allocated to the previous generation and does not require any additional storage.
* Determine route – The last generation route with the lowest fitness value is directly accessed by index in **O(1).** For a constant number of trucks each delivery in the designated delivery list is assigned which gives a time of **O(N)** with N being the number of deliveries. When the route is determined one variable is allocated with the selected route and each truck is assigned a list of deliveries making the space complexity **O(N)** with N being the number of deliveries.

In the worst case the genetic algorithm applied to the VRP will run in **O(*N2*)** with N being the number of packages being delivered. The most time-consuming method in the algorithm is the creation of deliveries which will on average be **O(N)** in the worst-case **O(*N2*).** The largest amount of space is used by the package table giving the program a space complexity of **O(N)** with N being the number of packages.

**Scalability of my Solution:**

The genetic algorithm is a great solution to the VRP because it can be easily adapted to a larger volume of input and can handle additional restrictions without a significant increase in runtime. When working with an increased number of packages and vehicles, the number of generations created can be tuned to find a balance between the fitness of the solution and the time it takes to generate it. When creating the hash table to store package data, the number of lines in the file is first read and a hash table is created with the number of buckets needed. Additionally, no additional programming is needed to adapt the solution to a changing number of packages, locations, or vehicles. Any variables that determine the functionality of route or generation creation have been coded as constants that can be easily updated in the code or a GUI can be created so a user can update these values before generating a solution. The solution also leaves room to adapt to changing priorities through the generation of the fitness value. For this implementation the fitness value is determined by the distance the vehicles travel, if deadlines are met, and if vehicles are over their package limit. If heavy packages are being shipped the fitness value could be updated to account for weight with only a few lines of code being added.

**Software Efficiency and Maintenance:**

As mentioned above the software created for this project was created with efficiency and ease of maintenance in mind. To achieve efficient route generation with a growing number of packages the number of generations can be adjusted and tuned to find a balance between the fitness of the solution generated for the provided input and the time it takes to generate the solution. Before any package data is read, the number of lines in the CSV file is determined and a data structure with a matching capacity is created. Storage for the delivery table is dynamically allocated after the number of locations that will be visited is determined by the package insertion function and can be adapted to any number of locations. The program can also be adjusted by changing constant values in the generation module to account for premature convergence of values if many generations are needed to find an appropriate solution. If the number of vehicles that are servicing the area changes a constant value in the delivery truck module can be easily adjusted to adapt the program to use a smaller/large number of vehicles in the solution. Values used to assess the fitness of the route (delivery deadline times, penalty values, and package limits) have also been coded as constants so they can be easily adjusted if the restrictions placed on the deliveries change over time.

**Strengths of the Genetic Algorithm for the VRP:**

The genetic algorithm is a good match for the VRP because it can be easily adapted to a varying number of packages, locations, and delivery vehicles. When applied to a real-life situation the number of packages delivered each day, locations that will be visited, and the number of vehicles used can change rapidly. A routing solution that can generate an optimal route timely and consider all relevant factors is essential. It also provides a robust solution to the varying needs of vehicle routing services. Over time constraints for vehicle routing services will change or be reprioritized to meet changing demands. With the genetic algorithm small changes to the assessment of fitness values can make a major impact on the route that will be selected, reflecting current delivery goals.

**Other Algorithms for the VRP:**

The nearest neighbor algorithm could also be used to solve to the VRP. Using the nearest neighbor algorithm, you start from the hub and visit the nearest location that has not been visited until all locations have been visited (Pop, 2011). Using this method, you can minimize the distance traveled by vehicles, but the solution does not account for restrictions that may be placed on the deliveries, such a deadlines and maximum packages per vehicles. The algorithm could be adjusted to visit the nearest locations with a deadline first and then locations without a deadline, this solution however is still inferior to the genetic algorithm because it misses solutions that minimize distance and meet deadlines.

Another algorithm that is commonly used to solve the VRP is the Clark-Wright Savings Algorithm. Using this algorithm, you would first consider dispatching an individual vehicle to each delivery point and calculate the total distance that would be traveled. You then would consider the savings in distance that would be achieved if a single vehicle instead was dispatched to two locations. Savings are ranked in descending order and added to routes if the locations have not been allocated and do not violate any restrictions. This process is repeated until the savings list is empty (Larson, 1981). This algorithm is very effective at finding the most optimal route for multiple vehicles minimizing distance. However, it’s solution for considering restrictions placed on the deliveries is very black and white and can cause appropriate solutions to be disregarded unlike the assessment of fitness values in the genetic algorithm.

**Strengths and Weaknesses of Self-Adjusting Data Structures:**

Self-Adjusting data structures provide functionality to rearrange themselves after operations are performed on them. These operations can increase the efficiency of subsequent requests or can allocate additional storage if the structure is full. This makes these structures very powerful if the input size is unknown when the structure is created or if they hold a large amount of data that needs to be accessed quickly. Although they are good fit for many applications, they are not always the most efficient option. A self-adjusting binary search tree attempts to decrease access time by moving items that have been recently requested to the root. If the same items are accessed several times this operation will decrease access times for subsequent requests. However, if each element only needs to be accessed once this structure performs unnecessary operations that do no decrease access times. A dynamic array supports resizing if it becomes full but will typically require the array to be moved to a new location in memory. If the programming language used does not support garbage collection this can become problematic as the old storage location will not be unallocated with the resizing operation.

**My Self-Adjusting Data Structure:**

For my solution I chose a custom double hash table to store both my package and delivery data. Packages and deliveries are both created as objects with data members storing the associated values and the object’s location is stored in the double hash table. The advantage of using this type of structure is that it enables fast storage and retrieval of values based on the items key. Each time the program is run many packages will be inserted into the package table, and deliveries will be accessed several times when generating the fitness value of routes. This makes storage and access time of the package and delivery objects a key priority. I chose to use a double hash table to minimize collisions and access time if package IDs are not sequential numbers as they are with the provided data. Delivery numbers are always generated sequentially by the program to minimize collisions when these objects are inserted and accessed. Upon startup the program first determines the number of packages that will be inserted into the table and appropriately sizes the table. When package data is being read each unique location that will be visited is recorded and used to size the delivery table. The insert functions utilize the unique ID numbers to determine the bucket and insert the object’s reference. Subsequent lookups are then run-in constant time by calculating and accessing the associated bucket.

**My Double-Hash Table and Changes in Input:**

With a perfect hash function lookups can be processed in O(1) regardless of package input size. My hash table’s performance is degraded if package IDs are not sequential. More bucket indices need to be generated to resolve collisions bring the processing time for lookup to at most O(N). This can result in many operations being performed each insert or look up and for a very large input it may cause the table to be inefficient. There are many techniques that can be used to resolve collisions, I selected a double hash function because it typically reduces collisions by 50% when compared to a linear probing.

I have designed both my package and delivery hash tables to be sized to exactly the amount of input that will be provided. The amount of storage used for these tables will always be equivalent to the number of packages provided in the csv file and the number of locations that appear in the package file for deliveries. The space used will therefore always be O(N) with N being the number of packages. If packages are added and the table is full a resize operation is performed that adds five storage locations, to keep the storage at a minimum while limiting additional resize operations.

Changes to the number of trucks used would not affect my package or delivery table, this information is stored as a constant in the program and is only used by the route generation and assessment functions. Additionally changes to the number of cities would also not affect the space usage of my tables. If more cities are added then more delivery locations would also be added to the program, this would result in more space being used by my location list. If additional locations are used, I would implement the locations as another hash table instead of a list to decrease access time to the location data when packages are created and distances between locations are determined.

**Alternate Data Structure Options:**

A binary search tree is a great option for storing items that need to be quickly accessed. With a binary search tree items are inserted with a key value and compared to other keys in the tree to determine placement. All keys placed to the left of a node are less than or equal to the node’s key and all keys placed to the right of a node are greater than or equal to the node’s key. When searching for a key in the tree a maximum of O(H) comparisons is needed to find the desired key, with H being the height of the tree. If the nodes are evenly distributed this search function is extremely effective. With the input provided all packages IDs are sequential numbers. In a binary search tree if nodes are inserted in sorted order the resulting tree’s height is N – 1 with N being the number of items in the tree and search time would increase to O(N). This feature makes the hash table’s insert and search functions a better match for the data than a binary tree.

A list is another great option for storing data that needs to be quickly accessed. In a list each item is inserted in order and indexed. The index can then be used to access the desired item directly. If a list was used for this problem a lookup would either require the index of the item or the list to be traversed from index 0 increasing access time to O(N) in the worst case. The time to access an item could be decreased by storing a dictionary alongside the list with package IDs and indexes, this would require additional storage to be used that is not needed for the hash table. Because package IDs in a real world application would not always be sequential a hash table is a better match for the problem.

**Project Evaluation and Reflection:**

If I was going to do this project again, I would implement a graphical user interface that allows the end user to change the programming constants discussed in “Scalability of my Solution” and “Software Efficiency and Maintenance”. This interface would enable an end user with little to no programming experience to apply the solution to their unique situation and enable a means to fine tune the solution without changing code. In this interface I would include the ability to edit; the number of vehicles, vehicle speed, the number of generations, the package capacity each truck can carry, and delivery deadlines. When the values are entered into the interface, modifications are made to the constants for the class and the algorithm is re-run to find a new solution with the provided parameters.

# References

Larson, R., Odoni, A. (1981) *Urban Operations Research.* Prentice-Hall

Mallawaarachchi, V. (2017, Jul 7). *Introduction To Genetic Algorithms.* Medium*.* Retrieved from https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3.

Pop, P., Zelina, I. (2011, April). *Heuristic Algorithms for Solving the Generalized Vehicle Routing Problem.*

Retrieved from https://www.researchgate.net/publication/228521119\_Heuristic\_Algorithms\_for\_Solving\_the\_Generalized\_Vehicle\_Routing\_Problem

Wang, T. W. (2017). *Liner Ship Fleet Planning: Models and Algorithms.* Elsevier.