C950 WGUPS Algorithm Overview

Tim Stewart

ID #001476583

WGU Email: tstew91@my.wgu.edu

December 27, 2021

C950 Data Structures and Algorithms II

# Introduction

For the course in Data Structures and Algorithms II (C950), students are required to use the Python programming language to solve a vehicle-routing problem (VRP) for delivery of packages that has constraints on: number of trucks on the road, number of packages on trucks, package deadlines, packages that must be on a certain truck, groups of packages that must be delivered together, packages that cannot leave the hub before a certain time, and a limit on the cumulative mileage of all trucks.

# A. Algorithm Identification

To solve this VRP, I wrote an algorithm that uses backtracking and a nearest-neighbor strategy. The backtracking portion of the algorithm tries to send various combinations of 1 truck or 2 trucks at each possible time. The backtracking algorithm keeps track of the cumulative mileage for various combinations of 1-truck routes and 2-truck routes. The combination of routes with the lowest cumulative mileage is selected to be the winner.

Heuristics are used to load the trucks with packages that have deadlines first. Packages that are not able to be delivered before a certain time are skipped over until they are eligible to be delivered. Business logic is also applied to ensure that packages that must be on a certain truck or packages that must be delivered with other packages are handled correctly.

After a truck is loaded with packages, a nearest-neighbor strategy is used to find the route to deliver these packages with the least mileage: From the starting point of the hub, the nearest location by distance is chosen, and from that location the nearest unvisited location to that location is chosen. This strategy continues until all locations have been visited and the truck returns to the hub.

# B1. Logic Comments

My solution uses two important algorithms, and both of them are described below.

The backtracking algorithm recursively creates lists of truck delivery routes for packages that have not yet been delivered. When the number of packages reaches zero, that list of truck routes and its total mileage is recorded. When all calls to the backtracking function are completed, the list of truck routes with the overall least total mileage is selected as the winner. Here is pseudocode for this backtracking algorithm:

function: backtracking\_algorithm(list\_of\_routes, current\_time, remaining\_packages):

- If the list of remaining\_packages is empty, record this list\_of\_routes and its total mileage in a global list\_of\_list\_of\_routes and return.

- Otherwise if there are any remaining\_packages:

- Create a single route to deliver the packages remaining and add this single route to the running list of routes. Then recursively call solver\_helper() with an updated copy of the running list of routes, the current time, and the new list of remaining packages.

- Also create two routes to deliver the packages remaining and add these routes to the running list of routes. Then recursively call solver\_helper() with an updated copy of the running list of routes, the current time, and the new list of remaining packages.

When solver\_helper() returns from all its recursive calls, the global list\_of\_list\_of\_routes is sorted by mileage and the list\_of\_routes with the least mileage is returned as the winner.

A nearest-neighbor algorithm is used to create the actual driving route that a truck will take to deliver its packages. Here is pseudocode for the nearest-neighbor algorithm:

function: create\_truck\_route\_with\_nearest\_neighbor\_algorithm(list\_of\_packages):

- From the list\_of\_packages, we create a list of locations (or stops) the truck must visit to deliver all the packages.

- The truck starts at the hub.

- While there are still some locations the truck hasn’t visited yet:

- Sort the unvisited locations by distance from our current location

- Go to the unvisited location closest to our current location

- Mark this location as visited

- Continue the while loop

- When all package delivery locations have been visited, the truck returns to the hub.

# B2. Development Environment

I used the Visual Studio Code (“VS Code”) editor on my Windows 10 personal computer to write this application in Python 3.9.7.

# B3. Space-Time and Big-O

Program segment: hash table with separate chaining

Insertion into my hash table is **O(1)** because the hash function is O(1) and if there is a hash collision it is still O(1) in Python to append the (K, V) to the list at that bucket, for a total cost of O(1).

Retrieval from my hash table is **amortized O(1)** because the hash function is O(1) and in the case of hash collision it is O(m) to search the list of length m at that bucket for the requested (K, V). On average, lookup is O(n/m) for n lookups, which amortizes to O(1). In the very worst case, the cost of retrieval is O(n) if all insertions into the hash table collide to the exact same bucket and retrieval must search through a list of all key-value pairs stored in the hash table.

Program segment: backtracking algorithm

At each iteration of the backtracking algorithm, as many as four truck routes are created (i.e., one truck route named truck 1, one truck route named truck 2, two truck routes named truck 1 and truck 2, and two truck routes named truck 2 and truck 1). Even when only 1 package is passed to the recursive backtracking function, four more instances of the function will be called. Thus the time complexity for the backtracking algorithm has an upper bound of **O(4ⁿ)**.

To mitigate this exponential time complexity, I also implemented a branch-and-bound (B&B) paradigm to discard the partial solutions that cannot be winning solutions or that are highly unlikely to be winning solutions. For example, partial solutions whose route mileage exceeds 140 miles are discarded. Also, partial solutions whose mileage is greater than the current best solution are discarded. Also, partial solutions that have not completed by 6 p.m. are discarded. By discarding partial solutions that already fail to meet the requirements or are not likely to meet the requirements, the backtracking algorithm can work on only the partial solutions that have the potential to be a winning solution.

Program segment: nearest-neighbor algorithm

To create a truck route to visit *n* locations, my nearest-neighbor algorithm must sort the list of yet unvisited locations *n* – 1 times to determine which unvisited location is closest to the truck’s current location. Python 3.9’s inbuilt list sorting function is based on mergesort, which has a time complexity of O(n log₂n). Since we perform this sort n – 1 times, the nearest-neighbor algorithm has an overall time complexity of O(n² log₂n), or simply **O(n²)** since we only have to worry about the fastest-growing term.

# B4. Scalability and Adaptability

As the number of packages grows, the greatest effect on my application would be the increased size of the search space that the backtracking algorithm has to explore. As noted in section B3, the time complexity of my backtracking algorithm is O(4ⁿ). Even though I have mitigated some of the exponential time complexity by implementing a branch and bound paradigm, an increase in the number of packages would not scale very well. Realistically, if the number of incoming packages increased greatly, then it would also make sense to increase the number of trucks in operation and to increase the number of hubs in different parts of the city. For example, if the number of packages grew from 40 to 400 (a tenfold increase), then I would want to create ten zones around the city, each with its own hub and trucks and each responsible for only 40 packages. In this way, even if the number of packages were to increase tenfold, the amount of resources to handle the packages would also be increased proportionally, allowing the level of performance and efficiency to remain similar to before.

# B5. Software Efficiency and Maintainability

The application is **efficient** and easy to maintain because I have used classes to create objects for the trucks, packages, routes, and lists of routes. By encapsulating the data for individual trucks, packages, routes, and lists of routes I have made it easier to make changes to the software if the business requirements change. For example, if an application uses fixed variables like truck\_1\_mileage and truck\_2\_package\_list, then if the application needs to be heavily updated to allow three trucks or thirty trucks or three hundred trucks. But if the trucks are used as objects, then it becomes much easier to create lists of as many trucks as required and adjust the algorithms and functions of the application to process the lists of trucks. Using objects also makes it very easy to make copies of objects, which is crucial for the backtracking algorithm used in this application.

Another way that the application is **efficient** is that I calculate and cache the truck details and package statuses for each minute of the day so that when the user wants to navigate the trucks details and package statuses throughout the day, the application can simply load the relevant information for each minute and show it instantly. An inefficient alternative would be if the application had to recalculate the truck details and package statuses every time the user wanted to switch to a different time. By loading the details of each minute into a large array beforehand, the application lets the user browse the truck and package activity for the day easily and without any delays.

Regarding **maintainability**, a key aspect of this application’s code is that I have used very descriptive names for the variables and for the function names. In the professional software world, you will write software that someone else will have to take care of, and you will have to take care of software that someone else wrote. Because of this, we should write software in such a way that it will be easy for the next programmer to understand what is happening in the code. Using descriptive and clear variable names and function names will help a future programmer to read the code and understand right away what is happening.

Another way that the code for this application is maintainable is that I have used block comments and inline comments to help show what certain code is doing and to explain to a future programmer what some of the business logic is. Since I will not be around to coach the future programmer in person about the details of the program, these code comments are one of the best ways I can document the source code so that it is maintainable in the future.

# B6. Self-Adjusting Data Structures

My chosen self-adjusting data structure is a simple implementation of a hash table. The main **strength** of my hash table is its very fast lookup of stored entries. The hash table does this by converting a string key into a numeric value (i.e., the hash code), which is used to store the key-value pair in an array or in a list that allows indexed access like an array. Indexed lookup is very fast for a data structure. Another **strength** of the hash table is that it uses separate chaining to handle hash collisions that might occur. If two keys hash to the same bucket, both key-value pairs are stored in a list located at that bucket.

But one of the **weaknesses** of my hash table is that it is a fixed size. A possible future improvement of my hash table implementation would be a dynamic resizing property. Dynamic resizing would compare the number of entries in the hash table to the current capacity of the hash table, and when the number of entries approaches the capacity of the hash table then a larger hash table would be created and the entries from the old hash table would be re-hashed to the new hash table. Another **weakness** of my hash table is that the keys must be strings. This means it is not possible to use objects such as class instances as keys. A possible future improvement of my hash table would be to accept objects as keys and either retrieve the object’s own hash code representation or use the object’s string representation as the input to the hash function. There is a lot of risk and complexity involved with using objects as keys, so this feature should be added carefully and thoughtfully.

# C. Original Code

Please see the accompanying main.py file for the entry point into my program.

# C1. Identification Information

The first line of main.py includes my name (Tim Stewart) and my student id (001476583) as a Python comment.

# C2. Process and Flow Comments

I have included block comments and inline comments throughout my code, in main.py and in the additional files that are imported in various places.

# D. Data Structure

The self-adjusting data structure I used was a hash table. I used three instances of my hash table class to store information used by the backtracking and nearest-neighbor algorithm described in part A above.

# D1. Explanation of Data Structure

I used three hash tables in my program to store important information used by my program. As shown in my config.py file, I used hash tables to store three important data relationships:

(1) the distances between pairs of locations/stops (key = a string containing names of a pair of locations, value = the distance between the locations as a float as given by the project Excel/CSV file);

(2) my stop tuples that represent package delivery locations (key = a string containing a street address, value = the associated stop tuple); and

(3) my package class instances (key = the package id cast to a string, value = the package object that has the associated package id).

In each hash table, the key is cast to a string and then used to calculate a hash code to store the value. The values in each hash table could be a float value, or a named tuple, or a class instance. For the first hash table above, the value is a float for the distance. For the second hash table above, the value is an instance of my named tuple Stop (which is defined in my geo.py file). For the third hash table above, the value is an instance of my Package class (which is defined in my my\_package.py file).

# E. Hash Table

I developed a hash table, and its code is in my hash\_table.py file. The insertion function for instances of my hash table is named “add”. As described in part E of the requirements, the package ID number, delivery address, delivery deadline, delivery city, delivery zip code, package weight, delivery status (e.g., delivered, en route) for each package are loaded into the hash table. To make the data elements easier to work with, I store each package’s data elements in a Package object (which is defined in my my\_package.py file). The name of the hash table used to store the package data elements is all\_packages\_by\_id\_ht, which is in config.py. The package data elements are loaded into the hash table in the ingest\_packages() function in data.py.

# F. Look-Up Function

The look-up function for my hash table is named “get”. I retrieve package data elements from the hash table in multiple places in my program. For example, in route.py I access package data in the hash table so I can correctly group packages that have to be delivered together. As an another example, in delivery\_schedule\_writer.py I access package data in the hash table so I can correctly update the package delivery statuses.

# G. Interface

I wrote a text user interface (TUI) to explore and examine the delivery details for all 40 packages during the entire time period from 8 a.m. until the last truck returns to the hub after delivering the last packages. The screenshots below show the package status and truck info at 9 a.m., 10 a.m., and 12:30 p.m. As described in part G of the requirements, the interface shows the current time, the total mileage of all trucks, the delivery status of all packages including time. Package statuses include: “not yet at hub”, “at hub”, “aboard truck X” (i.e., en route), and “dlvrd @ time”.

# G1. First Status Check: 9 a.m.

Graphical user interface, text

Description automatically generated

# G2. Second Status Check: 10 a.m.

Graphical user interface, text

Description automatically generated

# G3. Third Status Check 12:30 pm

Graphical user interface, text

Description automatically generated

# H. Screenshots of Code Execution

The following screenshots show the successful completion of the program code. The first screenshot shows a summary of package delivery details, and the second screenshot is the user interface where the user can visit any time and see the truck and package statuses for that time.

Text

Description automatically generated

Text

Description automatically generated

# I1. Strengths of Chosen Algorithm

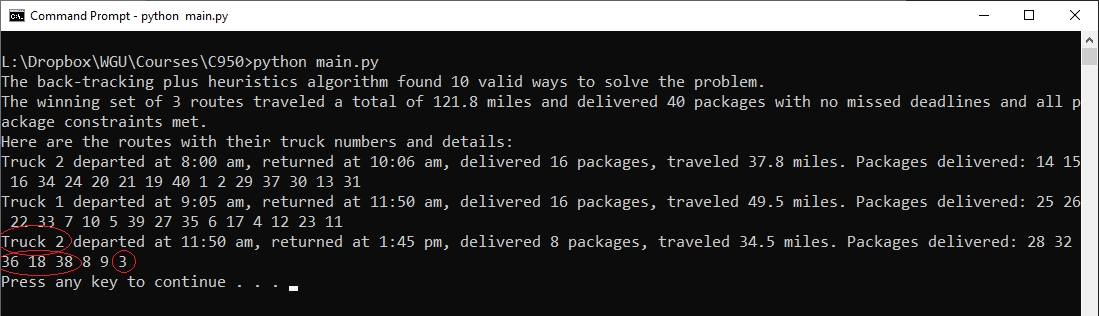
The backtracking strategy using branch and bound and the nearest-neighbor algorithm have several **strengths**. One strength is that backtracking offers a very thorough search of the possible solutions, so if a solution exists it is just a matter of time for the backtracking algorithm to find it. The nearest-neighbor algorithm also has its **strengths**: for example, it is a very easy algorithm to implement and it is very well researched and discussed in the academic literature.

The backtracking algorithm has **weaknesses** as well. As noted in section B3 above, the backtracking approach can use a lot of time and space to exhaustively search all possible options. And if the number of packages is increased, then the search space increases exponentially, which means the backtracking algorithm will have many more possible solutions to have to evaluate. The nearest-neighbor algorithm has its **weaknesses** as well. For example, the nearest-neighbor algorithm only considers distance in its calculation of which location to visit next. This could be a problem if a package going to a faraway location has a very soon deadline. (In this case, the nearest-neighbor algorithm would need to be modified to first visit high-priority locations and then visit nearby locations, in which case it isn’t really a “nearest neighbor” algorithm anymore.) Fortunately for this project, even just choosing the nearest neighbor still resulted in all packages being delivered on time. But with a different set of packages that have different delivery deadlines, it’s possible that the nearest-neighbor algorithm would not be successful.

# I2. Verification of Algorithm

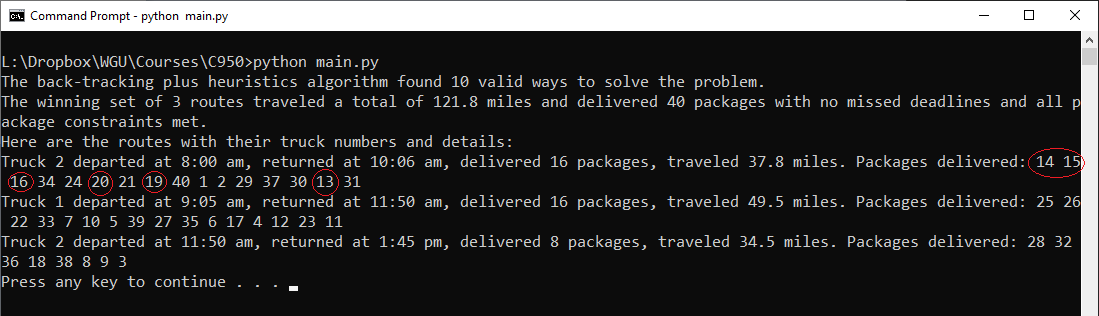
**Constraints on Which Truck Packages Can Go On**

Some packages had a constraint about which truck they could be delivered on. (In my source code I call this “truck affinity.”) Packages 3, 18, 36, and 38 needed to be on truck 2. This constraint was coded into the algorithm, and I manually verified that the algorithm put packages 3, 18, 36, and 38 on truck 2. See the red circles on the screenshot below.



**Constraints on Which Packages Must Go Together**

Some packages had a constraint about which other packages they needed to be delivered with. (In my code I call this “package affinity.”) Packages 13, 14, 15, 16, 19, and 20 needed to be delivered together. This constraint was coded into the algorithm, and I manually verified that the algorithm put packages 13, 14, 15, 16, 19, and 20 on the same truck route. See the red circles on the screenshot below.



**Constraints on Package Delivery Deadlines**

Some packages had a constraint about what time they needed to be delivered by, and this constraint was coded into the algorithm. I manually compared the package delivery deadlines to the actual delivery times produced by the algorithm, and for all packages with delivery deadlines, the algorithm delivered the package before the deadline arrived. (For package id #6, the package was delivered with only 3 minutes left before the deadline! Just in time!)

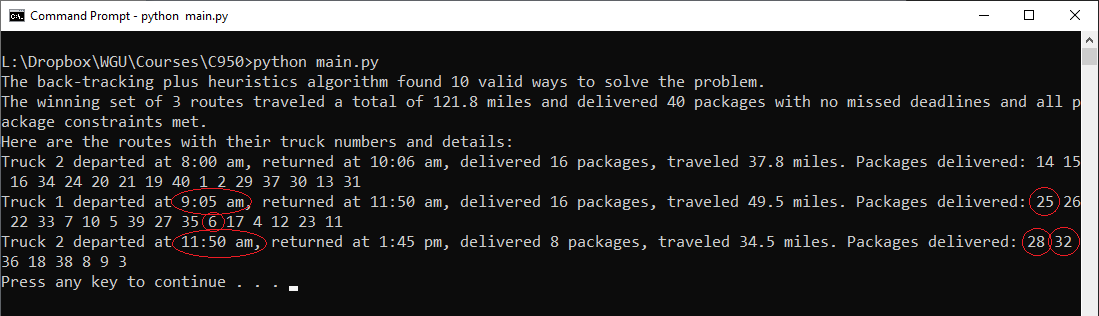
Table

Description automatically generated

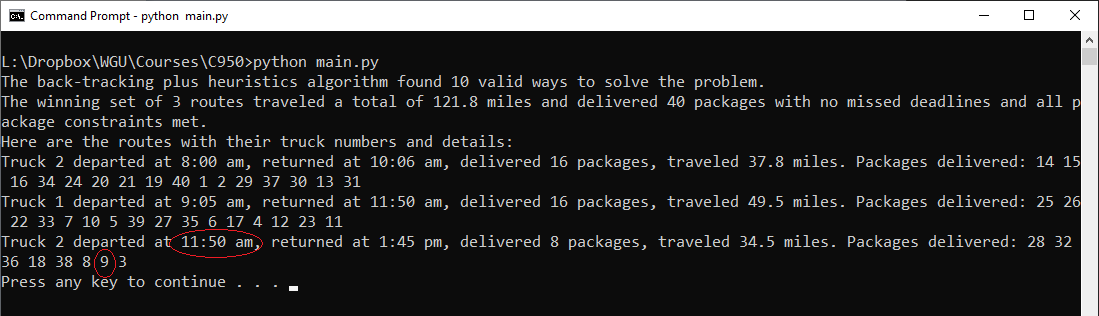
**Constraints on When Packages Can Leave Hub**

Some packages had constraints on when they could leave the hub. Packages 6, 25, 28, and 32 did not arrive at the hub until 9:05 a.m., and package 9 couldn’t leave the hub until 10:20 a.m. because it had an incorrect address. These constraints were coded into the algorithm, and I manually verified that these packages did not go out for delivery until they were allowed to.

This screenshot shows that packages that could not leave the hub until 9:05 a.m. all left the hub at 9:05 a.m. or later:



And this screenshot shows that the package that couldn’t leave the hub until 10:20 a.m. did not leave the hub until after 10:20 a.m.:



# I3. Other Possible Algorithms

For this project, I used the backtracking algorithm and the nearest-neighbor algorithm, but there are other algorithms that could have also been used instead of these. One alternative is a **brute-force** algorithm in which every possible combination of routes and departure times and package loads is tried and the outcome with the lowest mileage that still meets the constraints is used. Brute-force methods are impractical for all but the smallest problems. The backtracking approach that I used can be used to implement a brute-force approach, but since I also implemented branch and bound as part of my backtracking approach, I didn’t explore every possible candidate in order to find a winning solution. Another alternative is a **genetic** algorithm in which candidate solutions are combined and mixed to form new solutions. With each iteration of a genetic algorithm, the best solutions are kept and re-combined and the worse solutions are discarded. After many iterations, the resulting solutions contain the best parts of all the previous solutions that were tried. Genetic algorithms also incorporate elements of randomness, which can improve routes more quickly than techniques such as brute-force and backtracking, which operate more methodically and slowly.

# I3A. Algorithm Differences

My backtracking with branch and bound and nearest-neighbor algorithm is quite different from the pure brute-force approach or the genetic algorithm approach. Brute force has the disadvantage of continuing to evaluate candidate solutions even after it can be shown that it is not possible for that candidate solution to work or at least to work better than solutions already discovered. Genetic algorithms are even more different from the algorithm I used. For one thing, genetic algorithms introduce an element of randomness, and my algorithm is completely deterministic. Also, genetic algorithms combined components of past solutions to form new ones, and my backtracking algorithm treats each candidate solution as a separate entity and doesn’t combine them.

# J. Different Approach

If I have to solve a Vehicle Routing Problem (VRP) like this again, I would try to implement a solution using dynamic programming.

# K1. Verification of Data Structure

Text goes here

# K1A. Efficiency

Text goes here

# K1B. Overhead

Text goes here

# K1C. Implications

Text goes here

# K2. Other Data Structures

Text goes here

# K2a. Data Structure Differences

Text goes here

# M. Professional Communication

Text goes here

# L. Sources - Works Cited

Text goes here

An example:

Lysecky, R., & Vahid, F. (2018, June). *C950: Data Structures and Algorithms II*. zyBooks.

Retrieved March 22, 2021, from <https://learn.zybooks.com/zybook/WGUC950AY20182019/>