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 becomes O(n) if all insertions into the hash table collide to the 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

Text goes here

# B5. Software Efficiency and Maintainability

Text goes here

# B6. Self-Adjusting Data Structures

Text goes here

# C. Original Code

Text goes here

# C1. Identification Information

Text goes here

# C2. Process and Flow Comments

Text goes here

# D. Data Structure

Text goes here

# D1. Explanation of Data Structure

Text goes here

# E. Hash Table

Text goes here

# F. Look-Up Function

Text goes here

# G. Interface

Text goes here

# G1. First Status Check

Text goes here

# G2. Second Status Check

Text goes here

# G3. Third Status Check

Text goes here

# H. Screenshots of Code Execution

Screenshots (and possibly labels) go here

# I1. Strengths of Chosen Algorithm

Text goes here

# I2. Verification of Algorithm

Text goes here

# I3. Other possible Algorithms

Text goes here

# I3A. Algorithm Differences

Text goes here

# J. Different Approach

Text goes here

# 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/>