C950 WGUPS Algorithm Overview

Joshua Perry

ID #007217228

WGU Email: jper856@wgu.edu

12 / 30 / 22

C950 Data Structures and Algorithms II

# Introduction

The following document comments on the C950 WGUPS program that is submitted. Briefly, the program uses a Nearest Neighbor algorithm with a hash table data structure to find an approximation to the shortest total mileage to deliver. What the rubric outlines is a Minimal Viable Product (MVP) that one could elaborate on if enterprise application was needed.

# A. Algorithm Identification

The Nearest Neighbor algorithm is referenced numerous times throughout the provided supplementary resources. However, no mention of said algorithm can be found in the ZyBooks material. Upon further research it was decided to be implemented in the WGUPS project. The algorithm was used to decided which package, out of the packages manually loaded into a truck, would be delivered next. This application of Nearest Neighbor is self-adjusting as any combination of packages or modifications of packages would be handled dynamically.

# B1. Logic Comments

The following pseudocode outlines the Nearest Neighbor algorithm applied generally.

* Initialize vertices as unvisited
* Select any vertex. Set as current vertex x. Mark as visited
* Find minimum length edge from current vertex x to unvisited vertex y
* Set current vertex x to y. Mark y as visited
* Repeat from step 3 until all vertices visited

The following pseudocode outlines the Nearest Neighbor algorithm as it applies here.

* Load undelivered packages onto truck
* Set current location to “Hub”
* Find minimum length distance from current location to undelivered package address
* Set current location to minimum distance package address. Deliver package
* Repeat from step 3 until all packages delivered

In program comments also provide pseudocode.

# B2. Development Environment

Language: Python 3.9

IDE: PyCharm 2022.2 (Professional Edition) Build #PY-222.3345.131, built on July 27, 2022

OS: Windows 11 10.0

# B3. Space-Time and Big-O

Comments throughout the code analysis for Space-Time complexity and Big-O.

# B4. Scalability and Adaptability

Two parts of the provided program are self-adjusting and by nature scalable.

The package hash table is a self-adjusting data structure where any number of inputs can be handled. However, if a fixed size is continuously used and enterprise level of packages are to be tracked, then package retrieval from the hash table will become excessively slow. To adapt, the hash table would have to an implement resize function based upon a calculated load factor of the current table. This would keep size of the array and buckets to a respectable level and keep retrieval to an acceptable level.

The Nearest Neighbor algorithm applied here is a self-adjusting algorithm where any number of inputs can be handled. However, this application relies heavily upon the retrieval speed of the above hash table. Therefore, keeping the table optimized will keep the algorithm optimized. Furthermore, to properly scale the algorithm, it would be used to determine how each truck is to be loaded rather than running through whichever packages just happen to be on the truck. In an enterprise situation that could scale to Amazon level of packages, manually loading the trucks would be unacceptable and the algorithm would have to be applied in the way described above.

# B5. Software Efficiency and Maintainability

Overall, the program would be worst-case O(n^2). The comments throughout the program analysis time complexity.

As mentioned before, this rubric outlines an MVP. During the designing and constructing of the program documentation and maintenance was kept a priority. Each object has its own package. This allows for easy implementation of future ideas and refactoring. The documentation block each provide a quick synopsis, pseudocode, space-time analysis, and annotations. Inline documentation is in place to provide brief clarification or analysis. Finally, the code itself is self-documented.

# B6. Self-Adjusting Data Structures

The only self-adjusting data structure used in the program is a chaining hash table. The chaining hash table is a simple application of a hash table. A chaining hash table handles collisions by creating buckets or lists in place of where elements would be stored in a simple hash table. This effectively creates a list, holding a list in each index, which hold elements of key value pairs.

The chaining hash table has the following two strengths. First, it has a method to handle collisions. In an enterprise application where keys could become massive, eventually you will have collisions. Without any method to handle said collisions your system will fail. Second, insertion is Big-O(1). Apart from some constant time calculations, an insertion is just appending a key-value pair to the end of a bucket.

The chaining hash table has the following two weaknesses. First, while not the worst, the searching of an element in a chaining hash table is Big-O(n). In a statically sized table in an enterprise application each bucket can quickly become very large. The search function could potentially search through each item in a bucket to find said item. Second, the data structure is unordered. In an application where order matters this would not be a good choice as searching for a minimum/maximum value one would have to implement a slow search method as described above.

# C. Original Code

See program

# C1. Identification Information

See program

# C2. Process and Flow Comments

See program

# D. Data Structure

See program

# D1. Explanation of Data Structure

See section B.6 and the code comments.

Insertion is completed by using a key-value pair. A bucket for the key-value pair is determined by the following calculation: hash(key) % length(table). This bucket is essentially a list for all key-value pairs that come to the same bucket via calculation. The key-value pair is appending to the bucket.

Retrieval is completed by using a key. A bucket for the queried key-value pair is determined as seen above. Once the bucket has been determined, each key in the bucket is compared to the queried key. Once a match is found the key-value pair is returned. This is faster than a linear search. For example, compare a search for the key 100. The hash table is of length 10 and holds all values 1-100. The linear search uses a list 1-100. The hash table would only have to compare those in a similar bucket, bucket 0, comparing only 10 keys. The linear search would have to compare all keys up to 100.

# E. Hash Table

See program

# F. Look-Up Function

See program

# G. Interface

See program

# G1. First Status Check

See documentation directory

# G2. Second Status Check

See documentation directory

# G3. Third Status Check

See documentation directory

# H. Screenshots of Code Execution

See documentation directory

# I1. Strengths of Chosen Algorithm

The Nearest Neighbor algorithm created has the two following strengths. First, as the rubric is stated it seems like an MVP. The algorithm is a simple and heuristic algorithm to provide proof of concept for further elaboration. Second, as written it is efficient. Normally applied NN could have n number of vertices and would have to loop through them twice. This would mean that time-complexity is O(n^2). However, the comments from my code provide details as to why applied here, it is O(n).

Looping through the packages in the truck is constant time. Big O notation gives an upper bound to another function in concern with its end behavior. As the max number of packages in a truck is 16, the end behavior for looping through n packages within a truck is a constant 16. This can be upper bound by a constant, c = 17, multiplied by 1. Therefore, O(1) for initial loop and O(n) for the function.

# I2. Verification of Algorithm

All verifications of the algorithm are complete. Total mileage is less than 140 miles. All packages are delivered on time. All delivery specifications were handled. All verifications can be completed with the provided CLI.

# I3. Other possible Algorithms

Cheapest insertion.

Start with two vertices. Find the unvisited vertex that when added will create the shortest cycle. Repeat. Includes travel back to hub rather than creating another method.

Nearest insertion.

Start with two vertices. Find unvisited vertex that is closest to any of the visited vertices. Create a path from the closest visited vertex to the unvisited vertex to a visited vertex that creates the shortest path. Repeat. For this application, the data structure implemented would not have to be changed. Each package is a vertex. Delivered is visited and en-route is unvisited.

# I3A. Algorithm Differences

Cheapest Insertion vs Nearest Neighbor.

Time complexity of CI is O(n^2logn). The first edge in NI is arbitrary and may not be minimal from start location. The first edge in NN from start location is always minimal. With NI, no explicit routing back is necessary.

Nearest Insertion vs Nearest Neighbor

NI would have to loop through both visited and unvisited vertices while NN would loop only through unvisited vertices. The first edge in NI is arbitrary and may not be minimal from start location. The first edge in NN from start location is always minimal. With NI, no explicit routing back is necessary.

# J. Different Approach

To further increase enterprise scalability, I would implement dynamic resizing of the package hash table. This in effect would balance the buckets leading to consistent and reliable retrieval. This would involve tracking number of elements, tracking load factor, and implementing a re-hashing call.

# K1. Verification of Data Structure

All verifications of data structure are complete. Total mileage is less than 140 miles. All packages are delivered on time. All delivery specifications were handled. An “efficient” chaining hash table with a lookup function is present. All verifications can be completed with the provided CLI.

# K1A. Efficiency

The lookup method of the chaining hash table has a time-complexity of O(n). For reference the following is pseudocode for the lookup method.

* Find bucket with hash calculation
* For each key in bucket:
  + If key is queried key
  + Return value

Each bucket could have n number of elements up to infinity to loop through. Therefore, O(n)

# K1B. Overhead

The chaining hash table holds buckets (lists) of key-value pairs. Each bucket would hold a space in memory. Each hash table would hold however many determined buckets in consecutive memory. The hash table within our program holds packages and as the number of packages of grew so would the size/space of the buckets. Even though the size of each bucket would increase the amount of memory blocks would not unless resizing is implemented.

# K1C. Implications

More cities mean more packages to deliver.

Lookup time would increase as currently implemented. There are a limited number of buckets. Any increase in number of packages directly increases the lengths of any of the buckets. This increase in length would increase lookup time on average as the key queried could be at the end of the bucket.

Adding more trucks would mean more calls to the Nearest Neighbor algorithm. Depending upon the number of packages in the hash table program run time and space usage could be affected.

# K2. Other Data Structures

Dynamically resizing chaining hash table.

Same as currently implemented. However, dynamically resizes based upon determined load factor. Once load factor threshold met, table twice the size is created and values from the previous are rehashed into the new table.

Binary search tree.

Binary trees are lists consisting of nodes. Each node can have up to two children. Can be sorted in various ways. Binary search tree is where the left child node <= parent <= right child node.

# K2a. Data Structure Differences

Dynamically resizing chaining hash table.

DRCHT can balance the buckets and provide a more consistent and quicker retrieval than a normal chaining hash table. The resizing/rehashing process can take up a little time. Resizing must be managed properly or could get out of control quick.

Binary search tree.

Uses nodes rather than key-value pairs. Retrieval is much faster as time-complexity is O(logn).

# L. Sources - Works Cited

*C950 - Webinar-1 - Let’s Go Hashing - Recording*. (n.d.). *Panopto*. Retrieved December 30, 2022, from https://wgu.hosted.panopto.com/Panopto/Pages/Viewer.aspx?id=f08d7871-d57a-496e-a6a1-ac7601308c71.

“C950 WGUPS Project Implementation Steps - Example.” WGU.

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

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