**C950 Project WGUPS ROUTING PROGRAM**

### A.D. Stated Problem and Solution:

This project will use Dijkstra's algorithm, hash table, and array list for finding the shortest paths for delivering packages to different locations. In this program, I will be using Dijkstra's algorithm to find the shortest path between a source node and all other nodes in a weighted graph. Additionally, I will be utilizing hash tables to efficiently store and retrieve information about the nodes and their distances, as well as array lists to keep track of the nodes that have been processed and the ones that are yet to be processed. The program will be written in Python, a popular, high-level programming language known for its readability and ease of use. With this combination of algorithms and data structures, the program will find the optimal solution for delivering packages to their destinations in an efficient and reliable manner.

### D. identifies a self-adjusting data structure that performs well with the algorithm in part A and can store the package data.

* + Chaining hash table.

### D1. explains the data structure and how that data structure accounts for the relationship between the data points to be stored.

* + Hash table is a data structure that allows for efficient storage and retrieval of key-value pairs. By using a hash function, the key is transformed into an index of an array, which allows for constant time access to the associated value. This makes it a powerful tool for managing large data sets, as it allows for fast access to information without needing to search through the entire data set. In a chaining hash table, collisions can occur when two or more keys map to the same index.

### B1. Explain the algorithm’s logic using pseudocode.

Main Logic:

# Define package IDs for each truck

truck1\_pkgId\_list = [13,15,19,20,16,14,1,29,30,31,34,37,40,2,4,5]

truck2\_pkgId\_list = [6,25,28,32,18,36,38,7,8,10,11,12,17,19,21,22]

truck3\_pkgId\_list = [9,23,24,26,27,33,35,39]

# Combine package IDs for all trucks in a list

truck\_pkg\_list = [truck1\_pkgId\_list, truck2\_pkgId\_list, truck3\_pkgId\_list]

# Define departure times for each truck

truck1\_departTime = '08:00 AM'

truck2\_departTime = '08:00 AM'

truck3\_departTime = '11:00 AM'

# Get shortest path and distance for each truck

truck1\_path\_Mil = get\_shortes\_truckPath(truck1\_departTime, 1, truck1\_pkgId\_list, pkgs\_hashTable)

truck2\_path\_Mil = get\_shortes\_truckPath(truck2\_departTime, 2, truck2\_pkgId\_list, pkgs\_hashTable)

truck3\_path\_Mil = get\_shortes\_truckPath(truck3\_departTime, 3, truck3\_pkgId\_list, pkgs\_hashTable)

# Print the map list of package 20

print(pkgs\_hashTable.get(20).mapList)

# Loop through each truck's package list and print their delivery status and time

for each\_truck\_pkgList in truck\_pkg\_list:

for each\_pkgId in each\_truck\_pkgList:

thePkg = pkgs\_hashTable.get(each\_pkgId)

print(f'pkgId: {thePkg.get("pkgId")}, deadline: {thePkg.get("deadline")}, deliTime: {thePkg.get("deliTime")}, truckId: {thePkg.get("truckId")}, traMil: {thePkg.get("traMil")}')

print("This truck is done.")

# Calculate the total distance traveled by all trucks

ttlMil = truck1\_path\_Mil[1] + truck2\_path\_Mil[1] + truck3\_path\_Mil[1]

print(f'The total mileage traveled by all trucks: {ttlMil}.')

Hash Table:

class hashTable:

def \_\_init\_\_(self, input\_hashKeyNmeList=[], capacity=20):

self.mapList = []

self.hashKeyNmeList = input\_hashKeyNmeList

for i in range(capacity):

self.mapList.append([])

def hashValue(self, key):

return key % len(self.mapList)

def insert(self, key, value):

keyHash = None

translated\_key = None

if type(key) is str:

translated\_key = self.hashKeyNmeList.index(key)

keyHash = self.hashValue(translated\_key)

else:

translated\_key = key

keyHash = self.hashValue(key)

if len(self.mapList[keyHash]) == 0:

self.mapList[keyHash].append([translated\_key, value])

else:

for eachItem in self.mapList[keyHash]:

if eachItem[0] == translated\_key:

eachItem[1] = value

return True

self.mapList[keyHash].append([translated\_key, value])

def get(self, key):

if type(key) is str:

translated\_key = self.hashKeyNmeList.index(key)

hash = self.hashValue(translated\_key)

else:

translated\_key = key

hash = self.hashValue(key)

if len(self.mapList[hash]) > 0 or self.mapList[hash] is not None:

for eachItem in self.mapList[hash]:

if eachItem[0] is translated\_key:

return eachItem[1]

return None

def keys\_list(self):

if len(self.mapList) == 0:

return None

theList = list()

for eachItem in self.mapList:

for eachSubItem in eachItem:

theList.append(eachSubItem[0])

return theList

def has\_theVal(self, input\_key, input\_val):

target\_val = self.get(input\_key)

if target\_val is input\_val:

return True

return False

def searc\_byValue\_list(self, value):

resultList = []

for itemList in self.mapList:

for eachItem in itemList:

if eachItem[1] is value:

resultList.append(eachItem)

return resultList

def searcSndLvlMapTable\_byValue\_list(self, input\_key, input\_val):

resultList = []

for itemList in self.mapList:

for eachItem in itemList:

if eachItem[1].get(input\_key) == input\_val:

resultList.append(eachItem)

return resultList

Dijkstra’s Algorithm of shortest path:

function dijkstra\_shortestPath(which\_result, origin\_VertKey, destination\_VertKey):

graph\_list = get\_graph\_list()

vertices\_hashTable = get\_vertices\_hashTable()

unvisited\_vertices\_keyList = vertices\_hashTable.keys\_list()

vertices\_hashTable.get(origin\_VertKey).insert("distance", 0)

LeastDist\_vertexId\_list = [None]

function define\_aVertexDistance(aVertex\_key, unvisit\_vertices\_keyList):

if length(unvisit\_vertices\_keyList) = 0:

return true

for curr\_adjVert\_key in graph\_list[aVertex\_key][1]:

if graph\_list[aVertex\_key][1][curr\_adjVert\_key] is not None:

curr\_adjVert\_distance = vertices\_hashTable.get(aVertex\_key).get("distance") + graph\_list[aVertex\_key][1][curr\_adjVert\_key]

if curr\_adjVert\_key in unvisit\_vertices\_keyList:

if curr\_adjVert\_distance < vertices\_hashTable.get(curr\_adjVert\_key).get("distance"):

vertices\_hashTable.get(curr\_adjVert\_key).insert("distance", curr\_adjVert\_distance)

vertices\_hashTable.get(curr\_adjVert\_key).insert("pred", vertices\_hashTable.get(aVertex\_key).get("pred") + [aVertex\_key])

vertices\_hashTable.get(curr\_adjVert\_key).insert("id", curr\_adjVert\_key)

if LeastDist\_vertexId\_list[0] == None:

if vertices\_hashTable.get(aVertex\_key).get("distance") > 0:

LeastDist\_vertexId\_list[0] = aVertex\_key

elif vertices\_hashTable.get(LeastDist\_vertexId\_list[0]).get("distance") > vertices\_hashTable.get(aVertex\_key).get("distance"):

LeastDist\_vertexId\_list[0] = aVertex\_key

if aVertex\_key in unvisit\_vertices\_keyList:

unvisit\_vertices\_keyList.remove(aVertex\_key)

for adjVert\_key in range(0, length(graph\_list[aVertex\_key][1])):

if graph\_list[aVertex\_key][1][adjVert\_key] != 0 and adjVert\_key != None:

if adjVert\_key in unvisit\_vertices\_keyList:

return define\_aVertexDistance(adjVert\_key, unvisit\_vertices\_keyList)

define\_aVertexDistance(origin\_VertKey, unvisited\_vertices\_keyList)

if which\_result == "lstDist\_vertex":

return vertices\_hashTable.get(LeastDist\_vertexId\_list[0])

if which\_result == "updated\_vertices":

return vertices\_hashTable

if which\_result == "desti\_vertex":

return vertices\_hashTable.get(destination\_VertKey)

### B2. Describe the programming environment you used to create the Python application.

* + Python version : 3.9.7
  + IDE: Visual Studio Code 1.74.3
  + Operating System: Windows 11 Pro (Version: 22H2)
  + Processor AMD Ryzen 7 3700X 8-Core Processor 3.60 GHz
  + Installed RAM 16.0 GB
  + System type 64-bit operating system, x64-based processor

### B3. Evaluate the space-time complexity of each major segment of the program, and the entire program, using big-O notation.

* + Dijkstra’s algorithm[2]:
    - The time complexity: O(E \* log(V)) where V is the number of vertices and E is the number of edges in the graph.
    - space complexity: O(V).
  + Hash table:
    - The space complexity of the hashTable class is O(n+m), where n is the capacity of the hash table and m is the number of key-value pairs stored in the hash table.
  + The entire program or shortest truck delivery path:
    - Time complexity: O(N^2 \* E \* log(V) + N \* K) where N is the number of unique delivery locations, K is the number of packages associated with the current vertex, E is the number of edges and V is the number of vertices.
    - Space complexity: O(N + E + V + K).

### B4. Explain the capability of your solution to scale and adapt to a growing number of packages.

Together, Dijkstra's algorithm and hash table used in this solution can scale and adapt to a growing number of data points. As the data set grows, the hash table efficiently stores and retrieves data, while Dijkstra's algorithm navigates the network of connections between the data points. This makes it possible to manage larger and more complex data sets without sacrificing speed or accuracy.

### B5. Discuss why the software is efficient and easy to maintain.

By using Python modules, hash tables, and Dijkstra’s algorithm, the software will be efficient and easy to maintain in the ways of:

* Reusability: By using Python modules, developers can write code once and reuse it in multiple places, reducing the amount of duplicated code and effort needed to maintain the code. This approach also makes the code more modular and easier to understand and debug.
* Abstraction: The implementation codes of Dijkstra’s algorithm shortest path, hash tables, and data loading are organized in different module files. The modules provide a level of abstraction that hides the implementation details of a function or class, making it easier to use and maintain. By encapsulating the implementation details of a function or class, changes can be made to the implementation without affecting the rest of the code.
* Readability: By separating code into smaller, more focused modules, it becomes easier to read and understand the logic code, making it easier to maintain and update in the future.
* Testing: Using modules can simplify the process of testing code by allowing individual modules to be tested in isolation. This makes it easier to identify and isolate bugs, which can improve the overall quality of the software.

### B6. Discuss the strengths and weaknesses of the self-adjusting data structures

In this program, chaining hash tables were used to store the data of vertices and graphs for the delivery locations and distances between them. Chaining hash tables are self-adjusting data structures that are widely used for their efficient storage and retrieval of key-value pairs. Here are the strengths and weaknesses of chaining hash tables:

Strengths:

* Fast Access: Chaining hash tables allow for fast access to data, making them ideal for use in applications that require quick retrieval of data.
* Efficient use of Memory: Since chaining hash tables use linked lists to store key-value pairs with the same hash code, they are able to store data in a space-efficient manner.
* Self-Adjusting: Chaining hash tables can self-adjust by resizing the table and rehashing the data to avoid collisions.

Weaknesses:

* Slower Lookup Time: If a hash table has a high number of collisions, the linked list can become long, which can increase the lookup time.
* Not Suitable for Iterating: Chaining hash tables are not suitable for iterating over all the key-value pairs in a particular order.
* Load Factor: The performance of chaining hash tables can be affected by the load factor, which is the ratio of the number of items in the table to the number of slots. If the load factor is too high, it can increase the number of collisions and decrease the performance of the hash table.

### I1. 2 strengths specific to Dijkstra's Algorithm identified in part A are accurately described, and both strengths apply to the scenario.

* + Correctness: Dijkstra's Algorithm guarantees the correctness of the shortest path solution, making it a reliable algorithm.
  + Efficiency: Dijkstra's Algorithm is efficient for dense graphs, where the number of edges is close to the maximum possible number of edges in the graph. It has a time complexity of O(E log V), where E is the number of edges and V is the number of vertices in the graph.

### I3.A.: 2 algorithms different from the one used in the solution, and both algorithms meet the requirements in the scenario.

* + Bellman-Ford Algorithm: Bellman-Ford Algorithm is also used to find the shortest path between two nodes in a weighted graph. Like Dijkstra's Algorithm, it can handle graphs with negative edge weights and can be used for the single-source shortest path problem to find out the shortest pathes of package deliveries. However, unlike Dijkstra's Algorithm, Bellman-Ford Algorithm is slower, with a time complexity of O(VE), where E is the number of edges and V is the number of vertices in the graph.
  + A\* Algorithm: A\* Algorithm is a heuristic search algorithm that is often used for pathfinding and routing. Like Dijkstra's Algorithm, A\* Algorithm can find the shortest path between two nodes in a weighted graph to find out the shortest pathes of package deliveries. However, unlike Dijkstra's Algorithm, A\* Algorithm uses a heuristic function to guide the search, which can lead to faster results for some applications. A\* Algorithm is generally faster than Dijkstra's Algorithm but may not always find the optimal solution.

### J.: 1 aspect that would be done differently if the project were attempted again, and it includes details of the modifications that would be made.

* + For the data structure aspect, I would use Linear Probing. Linear probing is a type of open addressing, where when a collision occurs, the algorithm checks the next available slot in the hash table to store the element. If that slot is already occupied, it checks the next slot, and so on, until it finds an empty slot. Linear probing is simple to implement and has good cache locality since elements are stored contiguously. However, it can suffer from clustering, which can lead to poor performance.

### K1.A.B: addresses how changes in the number of packages directly affect the time needed to complete the look-up function and space usage.

* + If the package id is sure, the time to search for the package is constant in the hash table. If the package id is not sure, the time complexity of the look-up function for the packages is O(N). As the number of packages is reduced, it will cost less time to iterate in order to find out a package.
  + The space complexity is O(N) which is based on the number of packages.

### K1C. How changes to the number of trucks or the number of cities would affect look-up time and space usage.

* + The change in the number of trucks or the number of cities will change the time complexity of Dijkstra’s Algorithm. The time complexity is O(E \* log(V)) where V is the number of vertices and E is the number of edges in the graph. When there are fewer cities, there will be fewer vertices and edges and it will cost less time to find out the shortest path. Meanwhile, it cost less memory to store the data of vertices and edges.
  + Each truck has different packages and a certain different shortest path by running Dijkstra’s Algorithm. When there are fewer trucks, it will run Dijkstra’s Algorithm less which means less time.

### K2.A. 2 data structures other than the one used in part D that could meet the requirements in the scenario.

* + Linked lists: Linked lists are a data structure consisting of a collection of nodes, each containing a data element and a pointer to the next node in the list. They can be used to implement dynamic data structures that can grow or shrink as needed. One key advantage of a linked list is that it can be used to efficiently insert or delete elements at any position in the list, while a hash table requires rehashing if the size of the table needs to change.
  + Trees: Trees are a hierarchical data structure consisting of nodes with parent-child relationships. They can be used to represent hierarchical structures and can be used to implement search and sorting algorithms. Trees are more useful for representing relationships between objects and hierarchical organization of data, while hash tables are more useful for fast lookups, insertions, and deletions of key-value pairs.[1]

### L. Source

* + [1] Lysecky, R., &amp; Vahid, F. (n.d.). 4.1 Binary trees Version (June 2018). In C950: Data Structures and Algorithms II. Retrieved February 19, 2023, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/4/section/1.
  + [2] Lysecky, R., &amp; Vahid, F. (n.d.). 6.11 Algorithm: Dijkstra's shortest path Version (June 2018). In C950: Data Structures and Algorithms II. Retrieved February 19, 2023, from https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/6/section/11