**Name**: Eric Westfall

**Student ID**: 000842244

**Date**: 03/10/2021

**Project**: C950 Data Structures and Algorithms II PA

This is the readme for the C950 Data Structures and Algorithms II PA. Enclosed is all requested discussion around the Python program built.

**Part A**: **Identify a named self-adjusting algorithm (e.g., “Nearest Neighbor algorithm,” “Greedy algorithm”) that you used to create your program to deliver the packages.**

For the routing part of this program, I used a Greedy Lookahead algorithm. It is a dynamic programming algorithm that evaluates the next x steps and takes a single next step in the best permutation k steps ahead. It uses a breadth-first search to search through x tree layers (each permutation is a branch with further branches).

Further, alpha-beta pruning is used to improve the runtime of the algorithm.

**Part B**: **Write an overview of your program, in which you do the following:**

1. **Explain the algorithm’s logic using pseudocode.**

I’ve included pseudocode for the whole program here, because it’s unclear to me if the rubric is asking for pseudo code for the entire program or just the algorithm used in part A:

ABSOLUTE\_PACKAGE\_LIMIT = the maximum number of packages a truck can load

SHORTER\_PACKAGE\_LIMIT = the maximum number of packages truck #2 can load (for the purpose of shorter routes to grab priority packages)

MAX\_LOOKAHEAD = a number greater than 0 for the route calculation to look ahead

Import packages

Import locations

Import location distance\_matrix

**Perform Djikstra’s algorithm on each row in the distance\_matrix:**

# each row represents the “From” location and each column the “To” location

Nodes\_from\_start = []

Add the start node (the node # is the row in the distance\_matrix)

for each column in row:

set the distance to infinite and no path

if the vertex is not the start:

append the vertex to the unvisited list

set the start node’s distance from itself to 0 with no pathing

while there are unvisited nodes:

current\_vertex = pop the next unvisited node

for each to\_vertex connected to current\_vertex:

edge\_distance = distance from current\_vertex to to\_vertex

alt\_path\_distance = distance from start to current\_vertex\_num + edge\_distance

if alt\_path\_distance < nodes\_from\_start[to\_vertex][0]:

nodes\_from\_start[to\_vertex] = [alt\_path\_distance,

current\_vertex\_num]

**Run k-medoids clustering algorithm and assign each location to a cluster:**

k = number of clusters to form

pick k locations from the imported location set as initial medoids

set all medoids cost to 0

for an arbitrary number of maximum iterations:

cluster\_map = []

clusters = list of empty lists of length k

associate each location with its closest medoid:

for each location:

find closest medoid

append clusters[closest\_medoid] with the location

append cluster\_map with distance between closest medoid and location

(index of cluster\_map is the location number)

For each cluster:

medoid\_cost = the variance of the cluster

Update the medoids to find the medoids closest to the center of the clusters found in

association phase:

new\_medoids = []

for all clusters:

new\_medoid = current medoid of the cluster

old\_cost = medoid\_cost

for each location in this cluster:

find the distance from this location to all other locations in the

cluster and add it to the current\_location’s distance list

current\_cost = variance of the current\_location’s distance list

if current\_cost < old\_cost:

new\_medoid = location

old\_cost = current\_cost

add new\_medoid to new\_medoids list

if the previous medoids and new\_medoids are the same set of medoids, then k-medoids

is concluded

else, set the old medoids to the new\_medoids and go to the next iterations

while packages remain at the hub or en route to the hub or 10 iterations have passed:

truck\_to\_calculate = the truck with the lowest time value

# Load Truck

**load truck\_to\_calculate using either ABSOLUTE\_PACKAGE\_LIMIT or**

**SHORTER\_PACKAGE\_LIMIT**:

hub\_packages = get all packages

filter packages in hub\_packages that haven’t arrived at the hub by the truck’s time

value or require loading onto another truck

priority = packages with deadlines with in the next two hours

load priority packages and their bundles onto the truck if all said packages are in

hub\_packages and the truck isn’t above max capacity

if there is at least 1 package on the truck at this point:

cluster\_ = the cluster with the highest count among loaded packages

else:

cluster\_ = the cluster with the highest count among hub\_packages

next\_closest\_cluster = the cluster whose medoid is closest to the medoid from cluster\_

for packages in hub\_packages:

load packages (and their bundles) in cluster\_ as long as said packages are all in

hub\_packages and the truck isn’t above max capacity

for packages in hub\_packages:

load packages (and their bundles) in next\_closest\_cluster as long as said

packages are all in hub\_packages and the truck isn’t above max capacity

report to the user the packages that were loaded

# Calculate/Run Route

**calculate route for truck\_to\_calculate:**

destination\_set = the set of destinations for the packages loaded on the truck

destination\_queue = []

while there are locations left in the destination\_set:

reset best\_path to infinite distance and no path

for all permutations of the destination\_set of length MAX\_LOOKAHEAD:

reset current\_path to 0 distance and no paths

if the length of the destination\_set < k:

append the hub to the end of the permutation so the return is

considered

for each location in the permutation:

if there is a previous step in current\_path:

add the path and its distance between location and the

previous step in current\_path to current\_path

else:

add the path and its distance between location and the

previous step in destination\_queue (or hub if

no previous step in destination\_queue) to

current\_path

# Check for inefficiencies

if current\_path distance >= best\_path distance

or any location in the path from the previous step in current\_path (or hub, or destination\_queue, as specified in the previous conditional), besides the arrival location, is in the destination\_set

or the time taken to deliver to this location would cause a package to go past its deadline:

do not set a new best\_path (set\_best\_path =

False)

n = length of the destination\_set

skip the next max(1, n – number of locations in

current\_path – 1)! / max(1, n – MAX\_LOOKAHEAD)! - 1

permutations in the destination set

if set\_best\_path is true:

set a new best path

if no valid path is found: exit the program and report that no proper path was

found

update the truck’s time and total distance traveled with best\_path[0]

update the last\_step with best\_path[0]

deliver the package by removing it from the truck and adding it to the location

append the single next step to the destination\_queue (best\_path[0])

remove best\_path[0] from the destination\_set

report the delivery

append the hub to the destination\_queue

update the truck’s time and distance traveled going back to the hub

report the truck returning the destination queue

clear the destination\_set

append the full path to the truck (including all the steps between delivery locations) for

later reporting

Report results of the entire algorithm

1. **Describe the programming environment you used to create the Python application.**

Python Interpreter: 3.8

IED: PyCharm 2020.3.3 Professional Edition

Machine Model: MSI GE 66 Raider 10SGS

CPU: Intel i7-10750H @ 2.6GHz, 6 cores with hyperthreading (12 CPUs)

GPU: NVIDIA GeForce RTX 2080 Super with Max-Q, 8GB VRAM

RAM: 16GB RAM

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

**Time Complexity:**

Djikstra’s: Djikstra’s algorithm has a time complexity O(|E| + |V|log|V|).1 Since each vertex connects to each other vertex in the given adjacency matrix, the number of edges is |V|\*|V-1|. Therefore, Djikstra’s total computational complexity here is O(|V||V-1| + |V|log|V|), which evaluates to O(|V|2).

k-medoids: the time complexity of k-medoids is O(ik(n-k)^2), where i is the number of iterations, n is the number of objects, and k is the total number of clusters.2

Load Packages: O(4|P|) which evaluates to O(|P|), where p is the number of packages.

Greedy Look-ahead (calculate routes): this is a traveling salesman heuristic. TSP is an NP-hard problem and therefore no algorithm here will be polynomial time. Each iteration of the algorithm searches permutations of k length (max lookahead) with n locations to visit, which is just the standard permutations formula: n!/(n-k)! However, since the algorithm only takes one step at a time n times, it becomes O(n(n!/(n-k)!), which evaluates to O(n(n!)). When on iteration i, and n-i becomes less than or equal to k, each iteration evaluates to (n-i)! where n-i is the number of packages left. Therefore, the entire algorithm evaluates to the sum of two sequences:

n!/(n-k)! + (n-1)!/(n-k)! + (n-2)!/(n-k)! + … + (n-(n-k))!/(n-k)!

+

k! + (k-1)! + (k-2)! +…+ (k – (k – 1))!

In turn, since the number of iterations is equal to n, the sum of these sequences evaluates to O(n(n!)).

The entire program, then, uses time complexity O(n(n!)) due to Greedy Lookahead and space complexity O(n2) due to the adjacency matrix.

**Space Complexity:**

Djikstra’s: operates on the adjacency matrix, so it uses O(n2) space.

k-medoids: operates on the adjacency matrix, so it also uses O(n2) space.

Load Packages: uses hash tables to store locations and packages, which have O(n) space complexity. No other data structures have larger space complexity.

Greedy Look-ahead: permutation generation is done by Python’s innate itertools.permutations(), which uses Heap’s algorithm. Since the method is a dynamic programming algorithm that only needs the current permutation and the number of elements iterated over thus far, it does not require storing all the permutations to iterate over them. The algorithm requires O(n2) space to hold the adjacency matrix, which is the largest bit of space complexity needed.

1. **Explain the capability of your solution to scale and adapt to a growing number of packages.**

If the number of locations remains the same, Djikstra’s does not change nor does k-medoids with a growing number of packages.

The calculation time of loading packages grows linearly with a growing number of packages since each package at the hub is searched multiple times during the loading algorithm.

The Greedy Look-ahead portion of the algorithm grows non-polynomially with increases in the number of packages the truck has loaded, n, and (arbitrarily chosen) number of steps to look ahead, k. Given k remains constant, a growing number of packages will increase the amount of locations. However, since packages often go to the same location (and in real life, by something similar to the principle of locality, packages are much more likely to be going to the same location than by random chance, since people often make multiple orders at the same time), n will grow logarithmically. Additionally, each truck has a cap of 16 packages going to 16 different locations, so the maximum computational complexity is 16! + 15! + 14! + … + 2! + 1!.

The program will require O(n) space complexity to hold the packages in their hash table, so the space needed will grow linearly as packages are added.

1. **Discuss why the software is efficient and easy to maintain.:**

The software will adapt to any amount of packages, trucks, locations, and any MAX\_LOOKAHEAD set by the user up to the maximum of 16 (which makes the algorithm a brute force search through permutations), bounded by the user machine’s memory.

The routing algorithm is not overly efficient, but no TSP solution is. It absolutely solves the best path up to a length of MAX\_LOOKAHEAD from the current step on the path, but at the cost of time complexity O(n(n!)). Luckily, the runtime can be reduced by lowering the value of MAX\_LOOKAHEAD, although this makes it less likely to produce a valid solution. Each integer it is lowered by reduces the amount of permutations by n((n-k)! – (n-k-1))!, so the runtime is somewhat controllable.

Further, alpha-beta pruning is used to remove further searching down permutation paths that would cause a package to miss its deadline, result in a distance greater than the best current path, or contain intermediate steps already within the destination set (as then another permutation that doesn’t require the intermediate steps, somewhere else in the permutation’s space, will be present).

Luckily, the algorithm is fairly proficient even at MAX\_LOOKAHEAD 8, 9, and 10, which take only approximately 20 seconds, 150 seconds, and 180 seconds accordingly on my machine.

1. **Discuss the strengths and weaknesses of the self-adjusting data structures (e.g., the hash table).**

The hash table has an access time of O(1), which is as efficient as an algorithm can get. The hash table is indexed using an AVL tree. Since retrieving all items in a hash table would require iterating through the entire hash array, which is an O(n) algorithm with n = the size of the hash table including empty locations, I decided to index using an AVL tree, for which iterating through is O(n) where n = the size of the hash table without empty locations. Inserting into the AVL tree is O(log(n)).

**Part C: Write an original program to deliver all the packages, meeting all requirements, using the attached supporting documents “Salt Lake City Downtown Map,” “WGUPS Distance Table,” and the “WGUPS Package File.”**

All files have been included. MAX\_LOOKAHEAD has been set to 9, which was the value used to produce all screenshots here.

**Part D: Identify a self-adjusting data structure, such as a hash table, that can be used with the algorithm identified in part A to store the package data. (Note: Use only appropriate built-in data structures, except dictionaries. You must design, write, implement, and debug all code that you turn in for this assessment. Code downloaded from the Internet or acquired from another student or any other source may not be submitted and will result in automatic failure of this assessment).**

The package data is stored using a hash table. The hash table is indexed using an AVL tree. No dictionaries were used in constructing the hash table or AVL tree.

1. **Explain how your data structure accounts for the relationship between the data points you are storing.**

Hash tables are a mapping data structure. The data points I am storing are a mapping from a package or location number to the package or data object.

**Part E: Develop a hash table, without using any additional libraries or classes, that has an insertion function that takes the following components as input and inserts the components into the hash table:**

**• package ID number**

**• delivery address**

**• delivery deadline**

**• delivery city**

**• delivery zip code**

**• package weight**

**• delivery status (e.g., delivered, en route)**

The hash table requested is included.

**Part F: Develop a look-up function that takes the following components as input and returns the corresponding data elements:**

**• package ID number**

**• delivery address**

**• delivery deadline**

**• delivery city**

**• delivery zip code**

**• package weight**

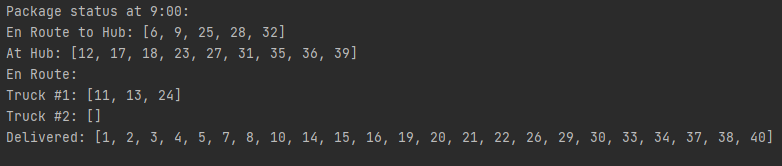
**• delivery status (i.e., “at the hub,” “en route,” or “delivered”), including the delivery time**

The hash table has a lookup function: search(x) where x is the number of the package or location.

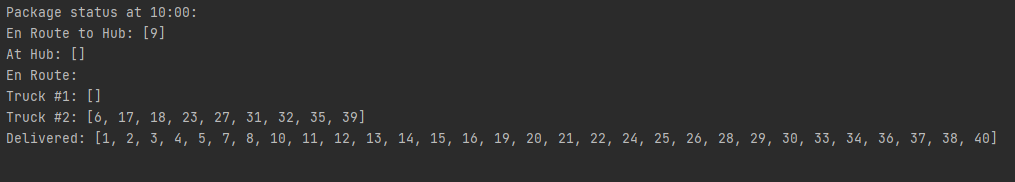
**Part G: Provide an interface for the user to view the status and info (as listed in part F) of any package at any time, and the total mileage traveled by all trucks. (The delivery status should report the package as at the hub, en route, or delivered. Delivery status must include the time.)**

After the algorithm has run in a Python console, the user has the option to search packages and truck times. Press ‘t’ and input a time to view the distances traveled by all trucks by the given time. Press ‘p,’ input a package, and input a time to view a package’s status at the given time. Press ‘n’ to exit the program.

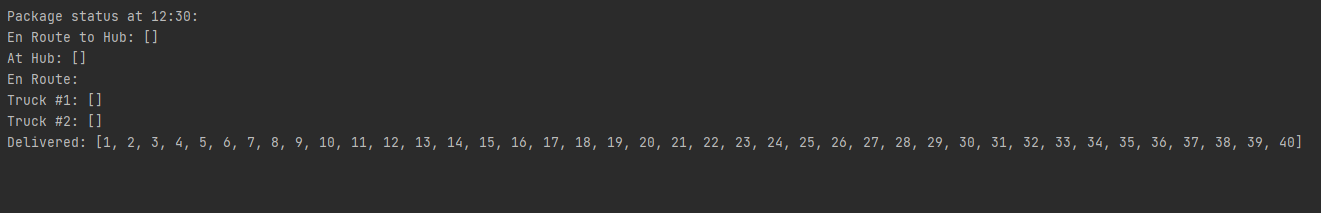
**1. Provide screenshots to show the status of all packages at a time between 8:35 a.m. and 9:25 a.m.**



**2. Provide screenshots to show the status of all packages at a time between 9:35 a.m. and 10:25 a.m.**

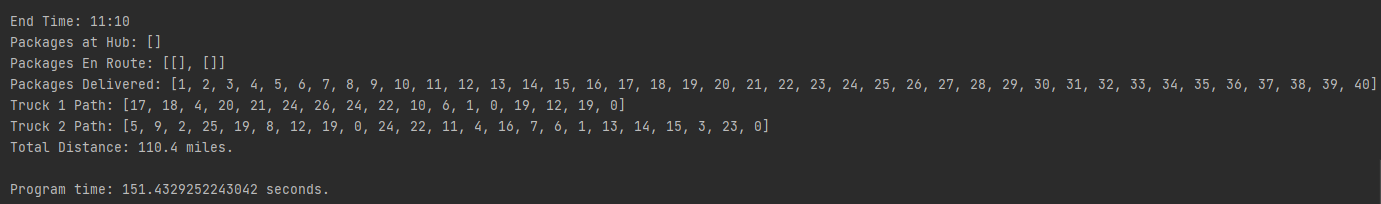


**3. Provide screenshots to show the status of all packages at a time between 12:03 p.m. and 1:12 p.m.**



**Part H: Provide a screenshot or screenshots showing successful completion of the code, free from runtime errors or warnings, that includes the total mileage traveled by all trucks.**

This screenshot is the result under conditions ABSOLUTE\_PACKAGE\_LIMIT = 16, SHORTER\_PACKAGE\_LIMIT = 16, MAX\_LOOKAHEAD = 9.



**Part I: Justify the core algorithm you identified in part A and used in the solution by doing the following:**

1. **Describe at least two strengths of the algorithm used in the solution.**

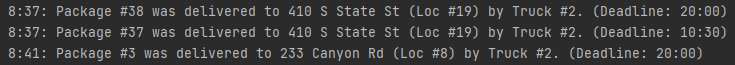
The algorithm used can be extended to an absolute solution in the routing phase by setting MAX\_LOOKAHEAD = ABSOLUTE\_PACKAGE\_LIMIT.

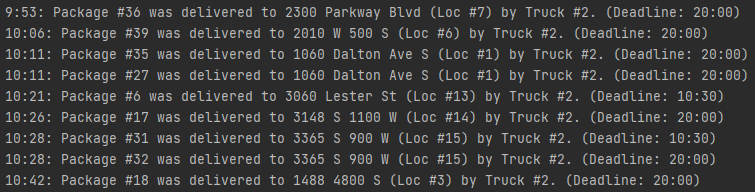
The algorithm finds better solutions than just a purely greedy algorithm, as it “looks ahead” by MAX\_LOOKAHEAD steps. It is a dynamic programming algorithm, as it updates the results of each permutation by only taking the one next best step in the best path found, and then finding the best path for the next MAX\_LOOKAHEAD steps from this location.

1. **Verify that the algorithm used in the solution meets all requirements in the scenario.**

The algorithm meets all requirements:

Packages that require a specific truck are accounted for (packages 3, 18, 36, and 38 require truck 2):

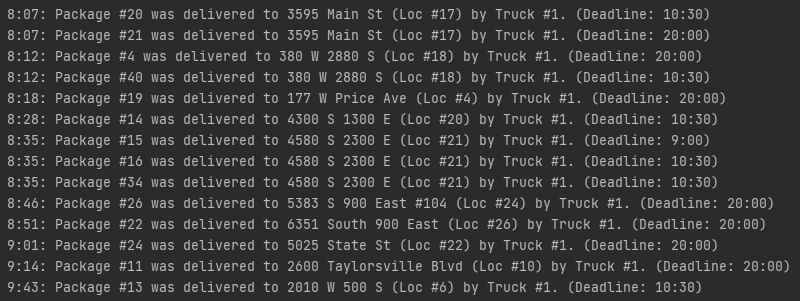


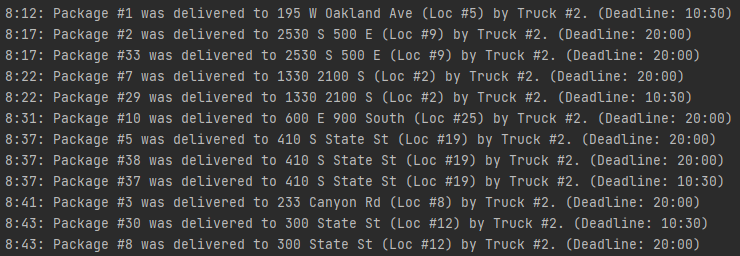


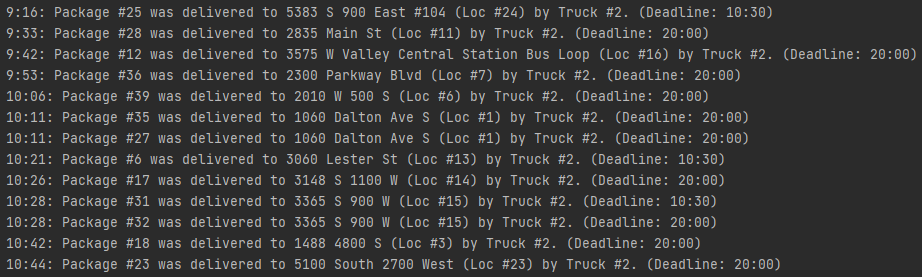
Packages that must be on the same truck together are accounted for. Packages 13, 14, 15, 16, 19, and 20 have said requirements:



All packages were delivered before their deadlines. Packages with deadlines of 10:30: 1, 6, 13, 14, 16, 20, 25, 29, 30, 31, 34, 37, and 40. Deadline of 9:00: 15.







Package 9 cannot be delivered until 10:20 since the address is not known:



Some packages don’t arrive at the hub before 8:00. Packages 6, 25, 28, and 32 do not arrive until after 9:05:



Total mileage is under 140 miles. This is the mileage for MAX\_LOOKAHEAD = 9:



However, if the final return to hub, after all packages are delivered, is not counted, then the total would be 96.4 miles.

1. **Identify two other named algorithms, different from the algorithm implemented in the solution, that would meet the requirements in the scenario.**

Two such examples would be 2-opt and simulated annealing.

1. **Describe how each algorithm identified in part I.3. is different from the algorithm used in the solution.**

While Greedy Look-ahead is ultimately a heuristic, as it does not produce an exact result for truckloads with locations, n > MAX\_LOOKAHEAD, it does produce an exact solution for the next MAX\_LOOKAHEAD or fewer steps. It has time complexity O(n(n!)). It has space complexity O(n2).

2-opt has time complexity O(2n) and would have space complexity O(n2) here as it would operate on the adjacency matrix. It works by undoing crossed paths in the solution. Unfortunately, uncrossing a path may introduce another cross path, and thus, while 2-opt is often far more efficient in practice, its time complexity reflects the NP-hard nature of TSP.

SA is a probabilistic heuristic. It searches the solution space starting with solution sets that are most probable to be best before it searches gradually less probable to be best solutions. SA has time complexity O(n!) to completely search the space, but because it is intended to be a heuristic, it does not completely search the solution space. It would have O(n2) space complexity as it would also operate on the adjacency matrix.

**Part J: Describe what you would do differently, other than the two algorithms identified in I3, if you did this project again.**

The primary thing I would do is include loading in the route calculations. I would get more efficient routes if the trucks considered what to load. However, this would drastically increase the permutation space needed to calculate (which could, of course, be optimized or a heuristic could be used). As it is, I only use k-medoids clustering heuristic, which helps considerably (before applying it, I was getting mileage of about 140-150 miles for most complete runs of the program…and having a lot of trouble meeting all the deadlines), but a more exact solution could have been found otherwise. I could have brute force found the perfect solution by letting my computer run overnight as long as I continued to use alpha-beta pruning.

Another change I would make is in the routing itself. After the remaining number of destinations, n, became less than or equal to MAX\_LOOKAHEAD, k, I would write it as the last iteration where it finds the best remaining path and takes it instead of continuing to go one step at a time. This would shave off significant amounts of permutation searching from k! + (k – 1)! + (k – 2)! + … + (k – (k – 1))! to just k!.

I considered using the AVL tree to represent the permutation space to search, but Heap’s algorithm, which is what is used by Python’s built-in itertools.permutations() method, generates each permutation using a dynamic programming function, which drastically reduces the necessary space complexity.

**Part K. Justify the data structure you identified in part D by doing the following:**

1. **Verify that the data structure used in the solution meets all requirements in the scenario.**

The data structure (hash table) meets the requirements. It is self-adjusting, it stores the package data, and it is used by the main routing algorithm.

1. **Explain how the time needed to complete the look-up function is affected by changes in the number of packages to be delivered.**

The time complexity of the look-up function in a hash table is always O(1).

1. **Explain how the data structure space usage is affected by changes in the number of packages to be delivered.**

The hash table I am using uses a quadratic hash function. To guarantee that the hash function finds a place upon inserting an object, the hash table size must be greater than double the amount of objects placed with it. Therefore, the space complexity is always greater than 2n and evaluates to O(n).

1. **Describe how changes to the number of trucks or the number of cities would affect the look-up time and the space usage of the data structure.**

Both trucks and locations are stored using hash tables. They would be effected in the same way as increasing packages, for their respective object classes. Look-up is always O(1) and space complexity is always O(n).

**2. Identify two other data structures that could meet the same requirements in the scenario.**

Other associative data structures that could be used here would be binary trees and a directly addressed array.

1. **Describe how each data structure identified in part K2 is different from the data structure used in the solution.**

Associative data structures all have three functions: insertion, deletion, and searching. Hash tables have O(1) insertion, O(1) deletion, and O(1) searching. Their space complexity is O(n).

Binary trees have insertion O(logn), O(logn) deletion, and O(logn) searching. Their space complexity is O(n).

A directly addressed array has insertion O(1) if on the edges (just an append) and O(logn) if in the middle. O(logn) deletion in the middle, O(1) on the ends. O(logn searching). Their space complexity is O(n).

References

Mehlhorn, Kurt; Sanders, Peter (2008). “Chapter 10. Shortest Paths.” *Algorithms and Data Structures: The Basic Toolbox*. Springer. doi: 10.1007/978-3-540-77978-0.

Tiwari, Mahendra; Singh, Randhir (2012). “Comparative Investigation of K-Means and K-Medoid Algorithm on Iris Data.” *International Journal of Engineering Research and Development*, Volume 4, Issue 8 (November 2012), pp. 69-72.