

A. Algorithm Identification

- Name: this software uses “Dijkstra’s shortest path algorithm ” to find the shortest path between each package.
- Self-adjusting: the “Dijkstra’s shortest path algorithm ” takes a “fromAddress” and a list of “packages” as the parameters and returns the “toAddress” that has the shortest distance in the “packages”. The algorithm adjusts the return value based on the parameters: “fromAddress” and “packages”.
- The code that loads packages to the truck is not self-adjusting.

B1. Logic Comments

- Basic concepts:
 1. Load all packages to the trucks
 2. Find the list of all adjacent edges
 - a. Take an initial address and a list of packages as parameters
 - b. Iterate through the list of packages, find the distance of the current package from the list of packages and the initial address
 - c. Append them to the adjacent edges list
 - d. Sort the adjacent edges list by the distance
 3. Find the optimal path for all the packages’ destinations from each truck
 - a. Iterate through all packages in the truck
 - b. Find the adjacent edges of the current package
 - c. Get the edge that has the shortest distance
 - d. Get the next package of that edge
 - e. Repeat the process until all the packages in the truck are connected
 4. Ask the user to insert a time interval
 5. Print reports that are based on the optimal path in step 3
 6. Print the total distance

Step 2 Pseudocode:

```
define function findAllAdjacentEdges(packageSet, fromAddress)
    allAdjacentEdges = new List
    for package in (packageSet)
        append to allAdjacentEdges [package ID,
            graph.connections(fromAddress, package.Address)]
    return sorted allAdjacentEdges
```

Step 3 Pseudocode:

```
define function findOptimalPath(packages in truck, startTime)
    optimalPath = new Dictionary
    currentTime = new Time object
    nextPackage = findAllAdjacentEdges(truck, initial Address)
    currentTime = startTime + (truck velocity(18mph) *
        package.distance)
    optimalPath[nextPackage] = [nextPackage.address,
        currentTime]
    for package in truck
        nextPackage = findAllAdjacentEdges(truck, nextPackage)
        currentTime = startTime + (truck velocity(18mph) *
            package.distance)
        optimalPath[nextPackage] = [nextPackage.address,
            currentTime]
    return optimalPath
```

B2. Development Environment

- IDE: PyCharm
- Programming Language: Python
- Operating System: Windows 10
- Hardware: AMD Ryzen 9 3900x 12-Core Processor, 32 GB RAM, GeForce RTX 2060 Super

B3. Space-Time and Big-O

dijkstra.py

Functions	Time Complexity	Space Complexity
intersection()	$O(1)$	$O(1)$
findAllAdjacentEdges()	$O(N)$	$O(N)$
findOptimalPath()	$O(N^2)$	$O(N^2)$

chainingHashTable.py

Functions	Time Complexity	Space Complexity
insert()	$O(N)$	$O(N)$
search()	$O(N)$	$O(N)$
remove()	$O(N)$	$O(N)$

loadData.py

Functions	Time Complexity	Space Complexity
setStatus()	$O(1)$	$O(1)$
convertTime()	$O(1)$	$O(1)$
loadPackages()	$O(N)$	$O(N)$
setDistance()	$O(1)$	$O(1)$
loadDistance()	$O(N)$	$O(N)$

main.py

Block of Code	Time Complexity	Space Complexity
Line 31 - 50	$O(N)$	$O(N)$
Line 62 - 80	$O(N)$	$O(N)$
Line 84 - 90	$O(N)$	$O(N)$
Line 97 - 131	$O(N)$	$O(N)$
Line 139 - 146	$O(N^2)$	$O(N^2)$

B4. Adaptability

- Dijkstra's shortest path algorithm in this software can adapt to the increasing number of packages in the truck because it takes the list of packages as an input. Then it iterates the list of packages to generate an optimal path. If the list of packages is increased or decreased, the algorithm will iterate it and generate the optimal path in the same way.
- The data structure that holds all the packages in this software is the Chaining Hash Table. Therefore, it can hold different sizes of packages.

B5. Software Efficiency and Maintainability

- Efficiency: according to the analysis in section B3, the efficiency of the entire software will be $O(N^2)$.
- Maintainability:
 - + All main functionalities in this software are separated into three files: "chainingHashTable.py", "loadData.py", and "dijkstra.py". They are used in the file "main.py" that runs the software. Within each file, tasks are separated into small functions. For example, the file "dijkstra.py" has three functions: "intersection", "findAllAdjacentEdges", and "findOptimalPath". Calling those functions will achieve tasks that are on their names. By breaking all functionalities into small pieces, this software allows developers to quickly debug and enhance the code when needed. In addition, this practice allows the developer to modify a specific task while not affecting the rest of the code.
 - + On top of each function and each block of codes, there is a line of comment that describes the code below. Comments will help developers to navigate quicker. They also help developers understand the code faster.

B6. Self-adjusting Data Structures

- Hash-table, dictionary, and list are the self-adjusting data structures that are used in this software
 1. Hash-table:
 - Strength: lookup data quickly
 - Weakness: difficult to look up the key with the provided data
 2. Dictionary:
 - Strength: lookup data quickly
 - Weakness: can not be sorted
 3. List:
 - Strength: all elements are in order
 - Weakness: need to iterate the list to lookup an element

H. Screenshots of Code Execution

```
Enter start time (HH:MM:SS): 08:35:00
Enter end time (HH:MM:SS): 09:25:00
-----
Package 1 - Status: en route
Package 2 - Status: en route
Package 3 - Status: at the hub
Package 4 - Status: en route
Package 5 - Status: at the hub
Package 6 - Status: at the hub
Package 7 - Status: en route
Package 8 - Status: at the hub
Package 9 - Status: at the hub
Package 10 - Status: en route
Package 11 - Status: at the hub
Package 12 - Status: en route
Package 13 - Status: at the hub
Package 14 - Status: delivered at 8:06:20
Package 15 - Status: delivered at 8:13:00
Package 16 - Status: delivered at 8:13:00
Package 17 - Status: at the hub
Package 18 - Status: at the hub
Package 19 - Status: delivered at 8:31:20
Package 20 - Status: delivered at 8:29:40
Package 21 - Status: delivered at 8:06:40
Package 22 - Status: at the hub
Package 23 - Status: at the hub
Package 24 - Status: at the hub
Package 25 - Status: at the hub
Package 26 - Status: at the hub
Package 27 - Status: at the hub
Package 28 - Status: at the hub
Package 29 - Status: en route
Package 30 - Status: en route
Package 31 - Status: delivered at 8:17:40
Package 32 - Status: at the hub
Package 33 - Status: at the hub
Package 34 - Status: at the hub
Package 35 - Status: at the hub
Package 36 - Status: at the hub
Package 37 - Status: en route
Package 38 - Status: at the hub
Package 39 - Status: at the hub
Package 40 - Status: delivered at 8:12:00
-----
Total Distance: 117.3
```

```
Enter start time (HH:MM:SS): 09:35:00
Enter end time (HH:MM:SS): 10:25:00
-----
Package 1 - Status: delivered at 8:55:00
Package 2 - Status: delivered at 9:04:40
Package 3 - Status: at the hub
Package 4 - Status: delivered at 8:58:40
Package 5 - Status: delivered at 9:25:20
Package 6 - Status: en route
Package 7 - Status: delivered at 9:10:00
Package 8 - Status: delivered at 9:28:40
Package 9 - Status: at the hub
Package 10 - Status: delivered at 9:19:20
Package 11 - Status: at the hub
Package 12 - Status: delivered at 8:36:00
Package 13 - Status: en route
Package 14 - Status: delivered at 8:06:20
Package 15 - Status: delivered at 8:13:00
Package 16 - Status: delivered at 8:13:00
Package 17 - Status: en route
Package 18 - Status: en route
Package 19 - Status: delivered at 8:31:20
Package 20 - Status: delivered at 8:29:40
Package 21 - Status: delivered at 8:06:40
Package 22 - Status: en route
Package 23 - Status: en route
Package 24 - Status: en route
Package 25 - Status: en route
Package 26 - Status: at the hub
Package 27 - Status: at the hub
Package 28 - Status: at the hub
Package 29 - Status: delivered at 8:37:00
Package 30 - Status: delivered at 8:54:40
Package 31 - Status: delivered at 8:17:40
Package 32 - Status: at the hub
Package 33 - Status: at the hub
Package 34 - Status: delivered at 9:29:00
Package 35 - Status: at the hub
Package 36 - Status: at the hub
Package 37 - Status: delivered at 8:51:20
Package 38 - Status: at the hub
Package 39 - Status: at the hub
Package 40 - Status: delivered at 8:12:00
-----
Total Distance: 117.3
```

```
Enter start time (HH:MM:SS): 12:03:00
Enter end time (HH:MM:SS): 13:12:00
-----
Package 1 - Status: delivered at 8:55:00
Package 2 - Status: delivered at 9:04:40
Package 3 - Status: delivered at 10:56:00
Package 4 - Status: delivered at 8:58:40
Package 5 - Status: delivered at 9:25:20
Package 6 - Status: delivered at 10:25:00
Package 7 - Status: delivered at 9:10:00
Package 8 - Status: delivered at 9:28:40
Package 9 - Status: delivered at 11:35:00
Package 10 - Status: delivered at 9:19:20
Package 11 - Status: delivered at 10:26:40
Package 12 - Status: delivered at 8:36:00
Package 13 - Status: delivered at 9:42:40
Package 14 - Status: delivered at 8:06:20
Package 15 - Status: delivered at 8:13:00
Package 16 - Status: delivered at 8:13:00
Package 17 - Status: delivered at 10:09:20
Package 18 - Status: delivered at 10:12:00
Package 19 - Status: delivered at 8:31:20
Package 20 - Status: delivered at 8:29:40
Package 21 - Status: delivered at 8:06:40
Package 22 - Status: delivered at 9:44:40
Package 23 - Status: delivered at 10:10:00
Package 24 - Status: delivered at 9:55:00
Package 25 - Status: delivered at 9:40:20
Package 26 - Status: delivered at 10:34:40
Package 27 - Status: delivered at 11:19:00
Package 28 - Status: delivered at 10:54:20
Package 29 - Status: delivered at 8:37:00
Package 30 - Status: delivered at 8:54:40
Package 31 - Status: delivered at 8:17:40
Package 32 - Status: delivered at 11:04:00
Package 33 - Status: delivered at 10:50:40
Package 34 - Status: delivered at 9:29:00
Package 35 - Status: delivered at 11:19:00
Package 36 - Status: delivered at 10:30:20
Package 37 - Status: delivered at 8:51:20
Package 38 - Status: delivered at 10:52:40
Package 39 - Status: delivered at 11:24:20
Package 40 - Status: delivered at 8:12:00
-----
Total Distance: 117.3
```

I1. Strengths of the Chosen Algorithm

1. Dijkstra's shortest path algorithm has low complexity $\rightarrow O(V+E\log(V))$ where V is the vertex and E is the edge. This algorithm increases the efficiency of the software.

2. Since Dijkstra is using a graph with weight edges, it is simple to apply to the problem that deals with maps and distances. The map can be represented by the graph, distance can be represented by the weight edges and the dropping point is the vertex.

I2. Verification of Algorithm

1. The total miles traveled by all trucks is 117

2. All packages are delivered on time

3. All packages are delivered according to the specifications

4. After the user enters the start and end times, the software will generate a report showing the status of all the packages. Users can verify the validity of points 2 and 3 above by comparing the status of each package and its specifications. At the bottom of the interface, there are the total milages that are calculated by combining the traveled distances of the three trucks.

I3. Other Possible Algorithms

1. Greedy algorithm - This algorithm could be used to determine which package will go to which truck. Since the Greedy algorithm relies on conditions to determine the best possible solution, it is useful when dealing with those packages that have special instructions (GeeksforGeeks, 2021).

2. Heuristics - This algorithm is a generalization of Dijkstra's shortest path. Heuristics can quickly determine the near-optimal path for the truck to deliver. A good Heuristic code can improve the execution speed of the software (Lysecky, 2018).

I3A. Algorithm Differences

1. Greedy algorithm v.s. Dijkstra shortest path: Greedy algorithm is difficult if it is applied to find the optimal path for the truck. The option-based logic of the Greedy algorithm is not directly related to the map, distance, and address. However, it can be combined with the Dijkstra algorithm to improve the result.

2. Heuristics v.s. Dijkstra shortest path: Heuristics is faster than Dijkstra. However, it uses a non-weight graph to find the optimal path. This is not ideal for the problem since the problem consists of maps and distances. Dijkstra's shortest path with graph and weight is a more suitable solution.

J. Different Approach

Instead of manually load the truck, I can use the Greedy Algorithm to determine the optimal packages that are loaded to the trucks so that I could obtain smaller total miles. However, the Greedy Algorithm needs to combine with Dijkstra's shortest path to achieve the goal. This will increase the complexity and execution time.

K1. Verification of Data Structure

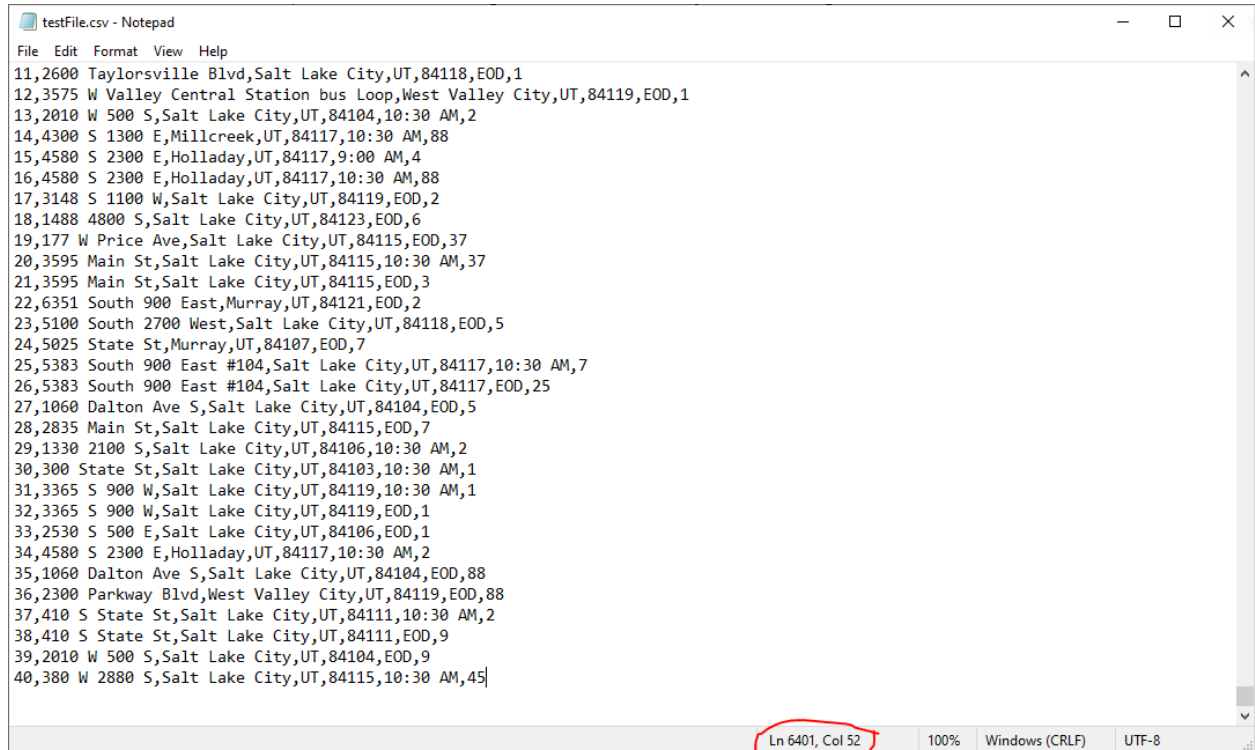
1. The total miles traveled by all trucks is 117
2. All packages are delivered on time
3. All packages are delivered according to the specifications
4. Hash-table has a look-up function that efficiently retrieves the data
5. After the user enters the start and end times, the software will generate a report showing the status of all the packages. Users can verify the validity of points 2 and 3 above by comparing the status of each package and its specifications. At the bottom of the interface, there are the total milages that are calculated by combining the traveled distances of the three trucks.

K1A. Efficiency

To find the execution time of the Chaining Hash-table data structure, I wrote this code:

```
-----  
from chainingHashTable import ChainingHashTable  
import loadData  
import time  
  
startTime = time.time()  
  
hashTable = ChainingHashTable()  
loadData.loadPackages("data/testFile.csv", hashTable)  
  
print("--- %s seconds ---" % (time.time() - startTime))  
-----
```

The function "loadData.loadPackages" will load all the data from "testFile.csv" to the Hash-table. For this experience, I have made a copy of the file "WGUPS Package File.csv", renamed it to "testFile.csv" and increase the number of rows to 6401.



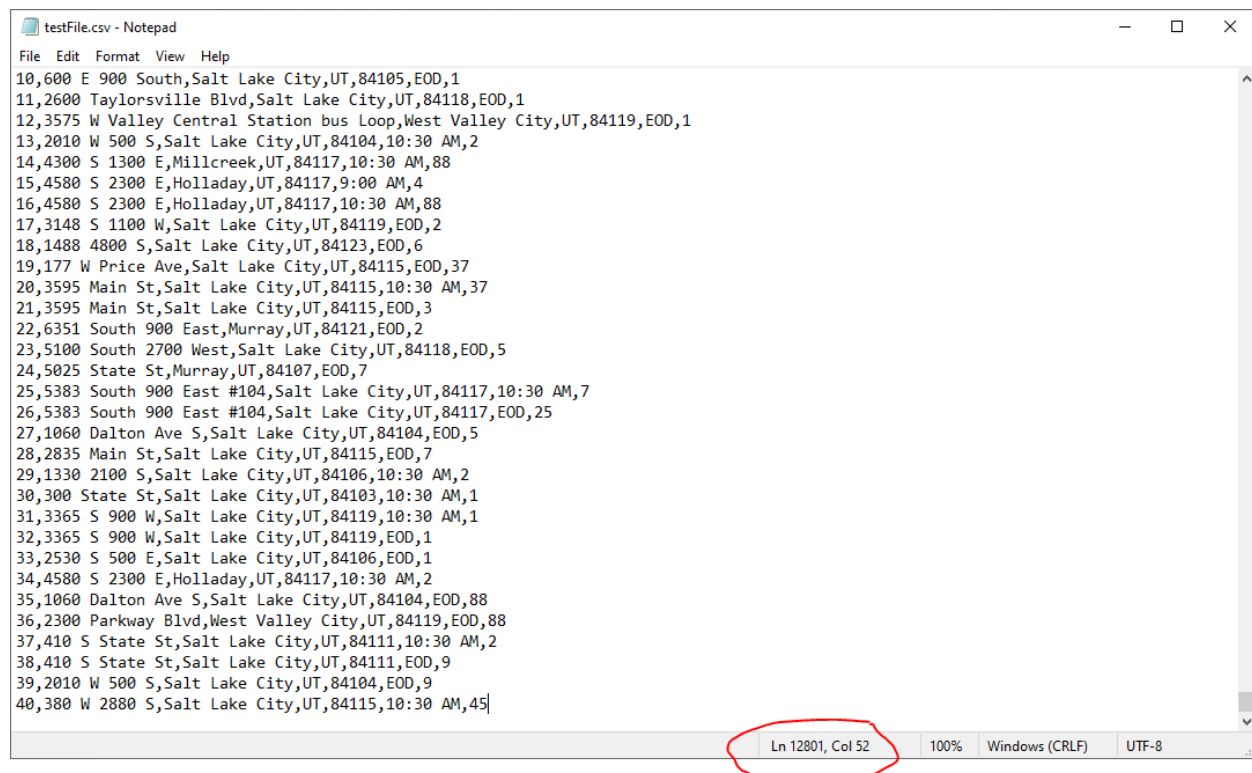
```
testFile.csv - Notepad
File Edit Format View Help
11,2600 Taylorsville Blvd,Salt Lake City,UT,84118,EOD,1
12,3575 W Valley Central Station bus Loop,West Valley City,UT,84119,EOD,1
13,2010 W 500 S,Salt Lake City,UT,84104,10:30 AM,2
14,4300 S 1300 E,Millcreek,UT,84117,10:30 AM,88
15,4580 S 2300 E,Holladay,UT,84117,9:00 AM,4
16,4580 S 2300 E,Holladay,UT,84117,10:30 AM,88
17,3148 S 1100 W,Salt Lake City,UT,84119,EOD,2
18,1488 4800 S,Salt Lake City,UT,84123,EOD,6
19,177 W Price Ave,Salt Lake City,UT,84115,EOD,37
20,3595 Main St,Salt Lake City,UT,84115,10:30 AM,37
21,3595 Main St,Salt Lake City,UT,84115,EOD,3
22,6351 South 900 East,Murray,UT,84121,EOD,2
23,5100 South 2700 West,Salt Lake City,UT,84118,EOD,5
24,5025 State St,Murray,UT,84107,EOD,7
25,5383 South 900 East #104,Salt Lake City,UT,84117,10:30 AM,7
26,5383 South 900 East #104,Salt Lake City,UT,84117,EOD,25
27,1060 Dalton Ave S,Salt Lake City,UT,84104,EOD,5
28,2835 Main St,Salt Lake City,UT,84115,EOD,7
29,1330 2100 S,Salt Lake City,UT,84106,10:30 AM,2
30,300 State St,Salt Lake City,UT,84103,10:30 AM,1
31,3365 S 900 W,Salt Lake City,UT,84119,10:30 AM,1
32,3365 S 900 W,Salt Lake City,UT,84119,EOD,1
33,2530 S 500 E,Salt Lake City,UT,84106,EOD,1
34,4580 S 2300 E,Holladay,UT,84117,10:30 AM,2
35,1060 Dalton Ave S,Salt Lake City,UT,84104,EOD,88
36,2300 Parkway Blvd,West Valley City,UT,84119,EOD,88
37,410 S State St,Salt Lake City,UT,84111,10:30 AM,2
38,410 S State St,Salt Lake City,UT,84111,EOD,9
39,2010 W 500 S,Salt Lake City,UT,84104,EOD,9
40,380 W 2880 S,Salt Lake City,UT,84115,10:30 AM,45|

Ln 6401, Col 52 100% Windows (CRLF) UTF-8
```

I ran the code and get the result:

```
--- 0.01251983642578125 seconds ---
```

To prove that adding data increase the execution time, I increase the row in “testFile” to 12801



```
testFile.csv - Notepad
File Edit Format View Help
10,600 E 900 South,Salt Lake City,UT,84105,EOD,1
11,2600 Taylorsville Blvd,Salt Lake City,UT,84118,EOD,1
12,3575 W Valley Central Station bus Loop,West Valley City,UT,84119,EOD,1
13,2010 W 500 S,Salt Lake City,UT,84104,10:30 AM,2
14,4300 S 1300 E,Millcreek,UT,84117,10:30 AM,88
15,4580 S 2300 E,Holladay,UT,84117,9:00 AM,4
16,4580 S 2300 E,Holladay,UT,84117,10:30 AM,88
17,3148 S 1100 W,Salt Lake City,UT,84119,EOD,2
18,1488 4800 S,Salt Lake City,UT,84123,EOD,6
19,177 W Price Ave,Salt Lake City,UT,84115,EOD,37
20,3595 Main St,Salt Lake City,UT,84115,10:30 AM,37
21,3595 Main St,Salt Lake City,UT,84115,EOD,3
22,6351 South 900 East,Murray,UT,84121,EOD,2
23,5100 South 2700 West,Salt Lake City,UT,84118,EOD,5
24,5025 State St,Murray,UT,84107,EOD,7
25,5383 South 900 East #104,Salt Lake City,UT,84117,10:30 AM,7
26,5383 South 900 East #104,Salt Lake City,UT,84117,EOD,25
27,1060 Dalton Ave S,Salt Lake City,UT,84104,EOD,5
28,2835 Main St,Salt Lake City,UT,84115,EOD,7
29,1330 2100 S,Salt Lake City,UT,84106,10:30 AM,2
30,300 State St,Salt Lake City,UT,84103,10:30 AM,1
31,3365 S 900 W,Salt Lake City,UT,84119,10:30 AM,1
32,3365 S 900 W,Salt Lake City,UT,84119,EOD,1
33,2530 S 500 E,Salt Lake City,UT,84106,EOD,1
34,4580 S 2300 E,Holladay,UT,84117,10:30 AM,2
35,1060 Dalton Ave S,Salt Lake City,UT,84104,EOD,88
36,2300 Parkway Blvd,West Valley City,UT,84119,EOD,88
37,410 S State St,Salt Lake City,UT,84111,10:30 AM,2
38,410 S State St,Salt Lake City,UT,84111,EOD,9
39,2010 W 500 S,Salt Lake City,UT,84104,EOD,9
40,380 W 2880 S,Salt Lake City,UT,84115,10:30 AM,45
```

I ran the code again and get the result:

```
--- 0.029026508331298828 seconds ---
```

The differences in execution time of the two files prove that increasing the size of the packages will affect the execution time of the Chaining Hash-table.

K1B. Overhead

Chaining Hash-table uses a bucket list to store data. In python, when the list's size increases, the space usage increases. Therefore, when the amount of data increase, the bucket list will expand and will require more space to store data.

K1C. Implications

According to chapter 7.1 in zyBook, a hash-table space complexity for looking up data is $O(1)$. Therefore, the space complexity is constant regardless of the size of the packages.

K2. Other Data Structures

1. Set

Set is an alternative for Chaining Hash-table. A set can store different data types (Lysecky, 2018). To store packages, Set can take package ID as the key value and store the other package's data into a subset.

2. Direct hashing

Direct hashing has a similar concept as Chaining Hash-table. It also has a lookup function that can search a package by the package ID.

K2A. Data Structures Differences

1. Set v.s. Chaining Hash-table - the disadvantage of Set is that it has a longer execution time than Hash-table. To look for an element, we iterate the set until we find the element. The worst time complexity for this operation is $O(n)$. The advantage of set over Chaining Hash-table is that set does not have to deal with the collision.

2. Direct hashing v.s. Chaining Hash-table - Direct hashing reserves a unique key for each data. That means if there are 1000 data, there are 1000 keys. This feature helps prevent the collision (Lysecky, 2018). The disadvantage of direct hashing is that it will require a very large space. On the other hand, the Chaining Hash-table use modulo operation to determine the unique key value for the data. This reduces the space needed to store data.

L. Sources

Lysecky , R. (n.d.). zyBooks. (2018)

<https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/8/section/1>

Lysecky, R. (n.d.). zyBooks. (2018)

<https://learn.zybooks.com/zybook/WGUC950AY20182019/chapter/7/section/7>

Greedy Algorithms. GeeksforGeeks. (2021, June 24).

<https://www.geeksforgeeks.org/greedy-algorithms>