Core Algorithm Overview

Author: Maxwell Brown (000933363)

Stated Problem

The purpose of this project is to create an algorithm for the Western Governors University Parcel Service (WGUPS) to determine the best route and delivery distribution for their Salt Lake City Daily Local Deliveries (DLD). Packages are currently not being delivered by their promised deadline and a more efficient and accurate solution is necessary. The Salt Lake City route is covered by three trucks, two drivers, and has a daily average of approximately 40 packages.

Assumptions: The following are the constraints imposed on the WGUPS delivery service:

- Each truck can carry a maximum of 16 packages.
- Trucks travel at an average speed of 18 miles per hour.
- Trucks have a "infinite amount of gas" with no need to stop.
- Each driver stays with the same truck as long as that truck is in service.
- Drivers leave the hub at 8:00 a.m., with the truck loaded, and can return to the hub for packages if needed. The day ends when all 40 packages have been delivered.
- Delivery time is instantaneous, i.e., no time passes while at a delivery (that time is factored into the average speed of the trucks).
- There is up to one special note for each package.
- The wrong delivery address for package #9, Third District Juvenile Court, will be corrected at 10:20 a.m. The correct address is 410 S State St., Salt Lake City, UT 84111.
- The package ID is unique; there are no collisions.
- No further assumptions exist or are allowed.

Restrictions Each package may have **one** special requirement that must be addressed by the WGUPS. The following are possible restrictions that may be imposed on a given package:

- The package must be delivered with other packages.
  - We refer to these packages as *linked* packages in our algorithm.
- The package must be delivered by a specific truck.
- The package has a specific deadline by which it must be delivered.
- The package is delayed in arriving to the depot and will not be available for pickup until later in the day.

Algorithm Overview

Assignment Requirements: A, B1

The core WGUPS Package Routing system utilizes a greedy algorithm to solve the routing problem and is primarily composed of three components. The first component utilizes a prioritization system to determine the order in which packages should be loaded onto trucks. The second component utilizes a minimization system to determine the route of a given truck that will minimize distance traveled based upon the packages that were loaded in the prioritization system. The third component utilizes a clock system to keep track of the time taken to deliver each package. The algorithm makes use of two trucks that execute a total of three trips based upon the assigned departure time assigned to the truck by the depot. The departure time allows for holding trucks at the depot to accomodate late package arrivals.

Prioritization System The prioritization system is used by the WGUPS Package Router to determine which packages should be given precedence when loading a given truck. The system separates packages into two groups - those considered to have **high priority** and those considered to have **regular priority**. If a package is not considered **high priority**, then it is considered **regular priority**.

Packages are considered **high priority** if they meet **ANY** of the following criteria:

- The package has an associated deadline.
- The package has a requirement to be delivered by a specific truck.
- The package is *linked* to other packages.

Trucks are first loaded to capacity with any high priority packages remaining at the depot. Once all high priority packages have been loaded onto trucks, loading of the regular priority packages commences and continues until the truck is at capacity.

Prioritization Pseudocode

```
high_priority = determine the list of high priority packages
regular_priority = determine the list of regular priority packages
```

```
while there are still packages to load:
  truck = select the next truck to load

  priority_deliveries = select packages from high_priority list that can be delivered by the truck
  regular_deliveries = select packages from regular_priority list that can be delivered by the truck

  for each package in the priority_deliveries list:
    if the truck is not full and the package is not on the truck already:
      load the package on the truck

  for each package in the regular_deliveries list:
    if the truck is not full and the package is not on the truck already:
      load the package on the truck
```

Minimization System The minimization system is used by the WGUPS Package Router after loading a truck to determine the shortest possible route that the truck can take to deliver its cargo. Beginning at the depot address as the **current address**, the system uses a greedy approach to exhaustively compare the distance from the current address to each remaining package destination to determine the next delivery. The algorithm will always select the shortest route from the remaining deliveries. Each package delivery updates the **current address** to the current delivery location.

Minimization Pseudocode

```
current_address = initialize the current_address to the depot address
total_distance = 0

while there are still packages on the truck:
  deliveries = sort remaining deliveries by distance from current_address to delivery_address
  closest_delivery = obtain the closest delivery_address from the deliveries
  distance = distance from current_location to closest_delivery
  total_distance = sum of the previous total_distance and the new distance

if the truck should return to the depot:
  total_distance = sum of the previous total_distance and the distance to return to the depot

return the total_distance traveled by the truck
```

Clock System The clock system is used by the WGUPS Package Router to keep track of both the internal time for each truck and the pickup and delivery time for each package. In addition, the clock system is used to determine the time at which packages arrive at the depot and when trucks should depart the depot. This allows the system to determine if a truck is able to deliver a given package.

Internal Truck Time The time of package pickup and delivery is determined by the internal time of each truck. The internal time of a truck is updated when the truck receives a new departure time from the depot, or when the truck executes delivery of a particular package.

Package Delivery Status Pseudocode

```
# Truck Departure Time
  if the new departure time is after the internal truck time:
    set the trucks internal time to the new departure time
  else:
    keep the internal truck time the same

# Truck Delivery Time
  when a truck delivers a package:
    calculate the time for the truck to travel the distance to the package delivery location
    update the internal truck time with the travel time
```

Package Delivery Status When a package is loaded onto a truck or delivered to its final destination, the package is updated with a *timestamp* indicating when these events took place. This allows for efficient reporting of the status of each package throughout the delivery period.

Package Delivery Status Pseudocode

```
# Package Pickup
  when a package is loaded onto a truck:
    set the package status to ON_TRUCK and the pickup_time to the current truck time

# Package Delivery
  when a package is delivered by a truck:
   set the package status to DELIVERED and the package delivery_time to the current truck time
```

Package Delivery Restrictions When a truck is selected to begin loading packages, the arrival time of each package to the depot is compared to the departure time of the truck from the depot to determine (in addition to any package restrictions) if the package can be loaded.

Package Delivery Restrictions Pseudocode

```
if the departure time of the truck from the depot \
  is later than the arrival time of the package at the depot AND \
  all package restrictions are met:
    the package can be delivered by the truck
else:
  the package cannot be delivered by the truck
```

Algorithm Programming Model

Assignment Requirements: B2

The core WGUPS Package routing system is implemented as a **command-line interface** that is designed to execute locally on the user's computer. All user interaction with the system will be through the command line interface. Users will be prompted to select from the available program options and input data on the command line as necessary. Program output will also be viewable by the user from the command line as text-based application reports. The user can exit the program at any time by typing the `exit` command.

There is no communication protocol for the application's access to data because all data and program code is stored locally on the user's machine. The application pulls the external data it needs locally from JSON files.

Algorithm Evaluation

Assignment Requirements: I

Strengths One strength of the chosen algorithm is that for the same set of data as input, the system will always plan the same route for the trucks of the WGUPS. This is not always the case for other solutions, such as combinatorial optimization solutions that employ metaheuristics to solve similar problems. These solutions may return slightly different solutions with each execution of the program, despite the same input.

Another strength of the chosen algorithm is that

Accuracy The algorithm meets all criteria as specified in the project requirements. All package restrictions are respected by the algorithm, and the route planned by the algorithm delivers all packages on time and in under 110 miles. This can be directly observed by running the program and observing the `all` packages report at varying time points.

Alternatives The two main alternatives that I considered when examining this problem were both metaheuristic algorithms. Metaheuristic algorithms are higher level procedures designed to select lower level heuristics to solve optimization problems.

The first is a metaheuristic algorithm named **Simulated Annealing**. The concept behind simulated annealing comes from the technique with the same name in metallurgy. The algorithm involves progressively "cooling" the "temperature" of the algorithm, and as the temperature cools the probability of accepting worse solutions decreases as the solution space is explored (van Laarhoven, 1987).

The second is another class of metaheuristic algorithms referred to as the **Genetic Algorithm**. The concept behind the genetic algorithm comes from the process of natural selection. The algorithm involves progressively generating increasingly "fit" solutions through crossover, mutation, and selection operations (Mitchell, 1995).

Both metaheuristic algorithms differ from the chosen greedy algorithm in several ways. The metaheuristic algorithms will not necessarily always generate the same output for the same input, so you may end up with varying results. In addition, the metaheuristic algorithms are designed to scale as the combinatorial optimization problems increase in complexity. This is in contrast to the greedy implementation, which does not scale well as the search space of the problem increases.

Algorithm Time Complexity

Assignment Requirements: B3

The time complexity of the core WGUPS Package Routing algorithm is $O(n^3 * \text{log}(n))$. This is because the algorithm utilizes two nested while loops which iterate through all packages. The time complexity of the two nested while loops is $O(n^2)$. The algorithm also utilizes the built-in Python `sorted` function within the nested while loops to sort the packages destinations by distance with each iteration. The built-in Python `sorted` function has a time complexity of $O(n * \text{log}(n))$. Therefore, the final time complexity of the algorithm is:

$$O(n^2) * O(n * \text{log}(n)) = O(n^3 * \text{log}(n))$$

Every class within the application has its methods annotated with their associated space and time complexity. However, for convenience, the space and time complexity of each class method is also listed below.

Method	Space Complexity	Time Complexity
get_distances	$O(n)$	$O(n)$
get_packages	$O(n)$	$O(n)$
get_prompts	$O(n)$	$O(n)$
load_json	$O(n)$	$O(n)$
load_distances	$O(n)$	$O(n)$
load_packages	$O(n)$	$O(n)$
load_prompts	$O(n)$	$O(n)$

DataLoader

Method	Space Complexity	Time Complexity
distance	$O(n)$	$O(n)$
to_depot	$O(n)$	$O(n)$

DistanceTable

Method	Space Complexity	Time Complexity
all	$O(n)$	$O(n)$
get	$O(n)$	$O(n)$

PackageTable

Method	Space Complexity	Time Complexity
can_deliver	$O(1)$	$O(1)$
deliver_packages	$O(n^3)$	$O(n^3 * \text{log}(n))$

Depot

Method	Space Complexity	Time Complexity
deliver	$O(1)$	$O(1)$
delivery_report	$O(1)$	$O(1)$
inline_report	$O(1)$	$O(1)$
is_high_priority	$O(1)$	$O(1)$
pickup	$O(1)$	$O(1)$
status_at	$O(1)$	$O(1)$

Package

Method	Space Complexity	Time Complexity
can_deliver	$O(1)$	$O(1)$
can_load	$O(1)$	$O(1)$
deliver_packages	$O(n^2)$	$O(n^2 * \log(n))$
depart_at	$O(1)$	$O(1)$
destinations	$O(n)$	$O(n)$
has_package	$O(1)$	$O(1)$
load_package	$O(1)$	$O(1)$
load_packages	$O(n)$	$O(n)$
travel_time	$O(1)$	$O(1)$
unload_package	$O(1)$	$O(1)$
unload_packages	$O(n)$	$O(n)$

Truck

Method	Space Complexity	Time Complexity
add_hours	$O(1)$	$O(1)$
add_minutes	$O(1)$	$O(1)$
hours	$O(1)$	$O(1)$
minutes	$O(1)$	$O(1)$
clone	$O(1)$	$O(1)$

Clock

Method	Space Complexity	Time Complexity
delete	$O(1)$	$O(n)$
get	$O(1)$	$O(n)$
keys	$O(n)$	$O(n)$
rehash	$O(n)$	$O(n)$
resize	$O(1)$	$O(1)$
set	$O(n)$	$O(n)$
should_rehash	$O(n)$	$O(n)$
values	$O(n)$	$O(n)$

HashSet

Method	Space Complexity	Time Complexity
execute_command	$O(1)$	$O(1)$
package_report	$O(n)$	$O(n)$
packages_report	$O(n)$	$O(n * \log(n))$
prompt	$O(1)$	$O(1)$
register_commands	$O(1)$	$O(n)$
register_prompts	$O(1)$	$O(n)$
route_distance	$O(n^3)$	$O(n^3 * \log(n))$
start	$O(1)$	$O(n)$
stop	$O(1)$	$O(1)$

Application

Method	Space Complexity	Time Complexity
execute	$O(1)$	$O(n)$
options	$O(n)$	$O(n)$
register	$O(1)$	$O(n)$

Commander

Method	Space Complexity	Time Complexity
clear	$O(1)$	$O(1)$
options	$O(n)$	$O(n)$
prompt	$O(1)$	$O(n)$
register	$O(1)$	$O(n)$

Prompter

Algorithm Scalability

Assignment Requirements: B4

The problem facing the WGUPS stems from a class of NP-hard problems known as the vehicle routing problems. These problems are notoriously difficult to solve, especially when restrictions are added to the problem specification. The time complexity of the core WGUPS Package Routing algorithm develoepd here is $O(n^3 * \log(n))$. As a result, the algorithm is not well-suited to scale to increasing numbers of packages given that the time it takes to solve this problem will scale exponentially with each additional package that must be routed.

In addition, the solutions used to address to the restrictions imposed upon the WGUPS are implemented in an inflexible manner. The algorithm can only handle working with the restrictions described above. In reality, a package routing system will need to be able to handle various restrictions and constraints placed upon packages to ensure that they are delivered in a timely fashion. Therefore, this algorithm is also not well-suited to evolve with changing market demands.

Algorithm Efficiency & Maintainability

Assignment Requirements: B5

The core algorithm developed for the WGUPS was implemented using a clean, object-oriented architecture designed for efficiency and maintainability. Each class and class method is well-documented to ensure that the purpose and function of each block of code can be easily understood. Developers working with the codebase should be able to ascertain the responsibilities of each class within the application, simplifying system maintenance.

Algorithm Data Structures

Assignment Requirements: B6, D, K1, K2

The primary data structure utilized throughout the application for data storage was in the form of a **linear probing hash table**. The hash table implementation used for this application is self-adjusting in that it will expand its capacity and rehash its contents upon reaching the internal capacity of the hash table. However, this can have negative implications on the runtime of the program. If the programmer does not select an appropriate initial capacity for the hash table, the table will constantly expand and rehash itself which is computationally expensive.

Requirements This hash table implementation is able to efficiently store and retrieve the specified package data according to the package identifier. This is evidenced specifically in the `PackageTable` class, where package data is stored and retrieved constantly throughout the application.

Efficiency The chosen hash table implementation is quite efficient for key-value pair data storage and retrieval. The hash table can be constructed with an initial capacity allowing for improved space complexity. However, the worst-case time complexity for resolving collisions with the linear probing implementation is $O(n)$. This is similar to the worst-case time complexity for other collision resolution techniques utilized in hash tables.

Strengths The strength of a hash table as the primary mechanism of data storage in this application lies in its ability to store and retrieve values by a particular key. This characteristic lends itself quite nicely to storage of our package data, given that information related to a given package can easily accessed in a hash table via the package's identifier.

Weaknesses The primary weakness of using a hash table as the storage mechanism for this application lies in the overhead of needing to rehash the entire table if the capacity of the table is exceeded. This is a computationally expensive task, particularly as the number of items in our table expands, and has a time complexity that is proportional to the number of items we are storing in our table ($O(n)$). Therefore, if more package data is added to the system after construction of the underlying hash table, the table will need to be constantly expanded and rehashed.

Other Data Structures We could have chosen to represent our package destinations as a graph of nodes, with each node representing a package destination and the edges between nodes representing the distance between nodes. In particular, a directed, weighted graph would be particularly useful given that the direction of our edges could be used to emulate the path taken by our trucks and the weight of each edge could be used to represent the distance in miles between destinations. This is in contrast to our implementation in which we chose to utilize a nested hash table to exhaustively store the distances between package destinations. A graph would likely have been much more memory efficient.

We could have also represented the destinations visited during a truck's route as a stack data structure. Using a stack, we could traverse our route by simply "popping" successive destinations off of the top our list of destinations and calculating the distance between the previously "popped" value and the currently "popped" value. In this way, we could calculate the total distance of a route by repeating this procedure until our stack of destinations was empty. This is in contrast to our implementation in which we chose to utilize a list that was sorted in order of distance between locations. A stack would have allowed for a much smaller memory footprint.

Future Directions

Assignment Requirements: J

One thing that I would implement differently were I to repeat this project would be the handling of package restrictions. The current solution only accomodates the specific restrictions outlined by the problem. However, my goal would be to implement a more flexible system that used a more generic restriction system that would allow for more complex and varied restrictions.

Another aspect of the current solution that I would revisit would be the selected algorithm. Although a greedy algorithm solves the problem as specified, it does not scale well as the search space of the problem increases. In addition, the currently implementation likely does not identify the most optimal solution because it does not consider solutions that partially fill trucks. I believe that a metaheuristic implementation would avoid these global optima by considering a much larger search space in a more efficient manner.

References

1. van Laarhoven, PJM. (1987). Simulated annealing. In: Simulated Annealing: Theory and Applications. Mathematics and Its Applications, vol 37. Springer, Dordrecht. Retrieved from https://link.springer.com/chapter/10.1007/978-94-015-7744-1_2.
2. Mitchell, M. (1995). Genetic algorithms: An overview. Mathematics, Computer Science. Complexity. Retrieved from https://www.semanticscholar.org/paper/Genetic-algorithms%3A-An-overview-Mitchell/ 630a7d2f0506cb5a01b09f07eef8a3b5a3af0387.