Chris Loverchio
#000875972
C950 – Data Structures & Algorithms II

Algorithm Analysis and Overview

A. Identify a named self-adjusting algorithm that was used to deliver the packages.

This implementation of the project uses a greedy algorithm to facilitate the delivery of packages to their destinations. Using Dijkstra's algorithm, the application is able to make a decision about what the next best delivery location is going to be from the current location, therefore making a locally optimal choice at each location.

B1. Explain the algorithm's logic using pseudocode.

Package Delivery

Time complexity: O(N^2)

- **Delivery route: O(N^2) time**

Space complexity: O(N)

```
given a truck and its associated start time:
    set the truck's current time to the start time
    set the truck's current location to the hub location
    get all undelivered packages assigned to the truck
    create a delivery route using the hub location and the truck's undelivered packages
    for each delivery in the delivery route:
        deliver the package with the truck
    get the shortest distance from the truck's current location back to the hub
    return to the hub
    update the total mileage using the truck's mileage
```

Delivery Route

Time complexity: O(N^2)

- **Closest delivery: O(N) time**

Space complexity: O(N)

```
given a current location and a collection of undelivered packages:
    while there are undelivered packages:
        find the closest delivery to the current location
        add the closest delivery to the delivery route
        update the current location to reflect the address of the closest delivery
        unassign the package associated with the closest delivery
    return the delivery route
```

Closest Delivery

Time complexity: O(N)
Space complexity: O(N)

given a current location and a collection of undelivered packages:
    for each package in the remainder of undelivered packages:
        find the shortest distance between the current location and the package address
        add the package and the shortest distance associated with it to a list of possible deliveries
    return the possible delivery with smallest distance

Shortest Distance

Time complexity: ~O(E log N)

- **Cache hit: O(1) time**
- **Cache miss: O(E log N) time - heap implementation of Dijkstra's algorithm**

Space complexity: O(N)

- **Cache: O(N) space**
- **Graph: O(V) space**

given an origin address and a destination address:
    if the shortest distance for the given origin and destination has been previously cached:
        return the shortest distance
    create a graph of locations and distances using the given origin
    find the shortest path between the origin and destination using the graph (Dijkstra's algorithm)
    set the shortest distance to the sum of distances in the shortest path
    add the shortest distance to the cache for the given origin and destination
    return the shortest distance

B2. Describe the programming environment used to create the application.

The application was programmed on an Ubuntu VM using the Python3 programming language and the PyCharm IDE.

B3. Evaluate the space-time complexity of each major segment of the program.

Space-time complexity analysis of the delivery logic is available in section B1.

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

The packages were manually loaded into the truck in this implementation; however, with minor changes to the loading functionality the application can function with any arbitrary distance and location data, provided it follows the same CSV formatting. Caching was also implemented in different spots within the application to reduce the impact of repeated queries and support scalability.

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

The application as a whole leverages object-oriented constructs and abstractions to promote maintainability. Service layer classes are named in accordance with functionality they provide. Functions are designed to be composable while having limited responsibilities.

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

There were two major self-adjusting data structures that were used in this implementation of the project. The first one being a hash table, which was largely used to store package data by its corresponding id but was also leveraged to cache distance data as well as status queries from the interface portion. The benefit to using a hash table in this context is that it provides the capability for constant time lookup operations. The disadvantages being the additional overhead in terms of spatial complexity and the risk of collisions during the insertion process. The second self-adjusting data structure that was used is a heap. The heap is mainly there to facilitate Dijkstra's algorithm, providing a convenient way to access the smallest unvisited vertex from a distance graph in constant time.

C1. Create an identifying comment within the first line of a file named "main.py" that includes your first name, last name, and student ID.

See line 1 of *main.py*

C2. Include comments in your code to explain the process and the flow of the program.

Single-line comments and multi-line docstrings are included on all notable functions.

D1. Explain how the data structures account for the relationship between the data points being stored.

As mentioned in section B6, the use of a hash table allowed for the mapping of package ids to the associated package data. Allowing for quick retrieval of all relevant data using just an identifier.

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

See *hashtable.py* for the initial implementation and *packaging_service.py* for its use with package data.

F. Develop a look-up function that takes the package id as input and returns the corresponding data elements.

See *hashtable.py* for the initial implementation and *packaging_service.py* for its use with package data.

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.

See *main.py* for the console interface and *tracking_service.py* for the status functionality.

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.

Screenshots are included in their own directory within the project structure.

I1. Describe at least 2 strengths of the algorithm used in the solution.

The greedy algorithm outlined in previous sections has a few distinct advantages. It leverages the current location of the truck to make an informed decision about where to go next. There's no need for backtracking or the analysis of previous choices and is fairly easy to reason about as a user.

I2. Verify that the algorithm used in the solution meets all requirements of in the scenario.

The implementation of the described algorithm was able to successfully deliver all 40 packages within ~120 miles, taking into consideration the specific deadlines and delays.

I3. Identify two other named algorithms that could have been used to meet the requirements.

Rather than choosing the next best delivery location at each step, a dynamic programming approach could have been used to build a more optimal route using previous steps in delivery process. An approach such as this would likely have a larger spatial complexity but can more accurately guarantee that an optimal route is generated. Recursive backtracking is another approach that could have yielded results for this kind of project. Using a recursive backtracking technique, each address starting from the hub could be explored, if one becomes unfeasible we could revert back to a previous address and explore other options. Similar to dynamic programming, this type of technique would likely require additional overhead compared to a greedy implementation but would be able to explore all permutations of delivery routes.

J. Describe what you would do differently if you were to do this project again.

Currently the project is using an implementation of Dijkstra's algorithm that attempts to mutate the graph object. This is not ideal because in order to avoid having to re-graph the data at each step, a deep copy of it has to be made specifically for Dijkstra runs. If I were to do this project again I would explore a different graph structure and therefore a different implementation of Dijkstra's algorithm that does not leverage mutation. I would have also liked to implement asynchronous creation of delivery routes which would reduce the startup time.

K1. Verify that the data structure used in the solution meets all requirements in the scenario.

A hash table as mentioned in section D1, fits the lookup and data modeling requirements of the project quite nicely. The ability to retrieve package data in constant time by an identifier such as a package id proved to be very useful for referencing and updating packages during the delivery process. The use of a hash table also worked out well for facilitating the required lookup queries from the console interface. A hash table's lookup function is expected to remain constant regardless of the amount of data being stored (depending on the hash function), which means that theoretically an increase of packages would increase the spatial complexity linearly but have little effect on runtime performance. The min heap implementation as mentioned in section B6 is what powered the delivery route process and is what ultimately allowed the algorithm to deliver all of the packages within the 140 mile limit imposed by the project requirements. Similar to a hash table, a min heap would expand linearly in proportion to the amount of data but its pop function would remain constant.

K2. Identify two other data structures that could meet the same requirements in the scenario.

A hash table's read performance is expected to be constant; however, it can be linear if the hash function doesn't work well with the data being stored. An AVL tree would make a great alternative to the hash table because it guarantees logarithmic read performance, which may even perform better in the context of this project because of the incomplexity surrounding the hash function that was chosen. Another benefit to AVL trees is that they can preserve order which would work well when all of the packages need to be listed. An alternative to the heap used by this project's implementation of Dijkstra's algorithm would be a priority queue; however, the performance would vary depending on how the priority queue was implemented.

L. Acknowledge sources, using in-text citations and references, for content that is quoted, paraphrased, or summarized.

ZyBooks. (n.d.). Retrieved December 21, 2020, from https://learn.zybooks.com/

Edmonds, J. (2008). *How to think about algorithms*. Cambridge: Cambridge University Press.