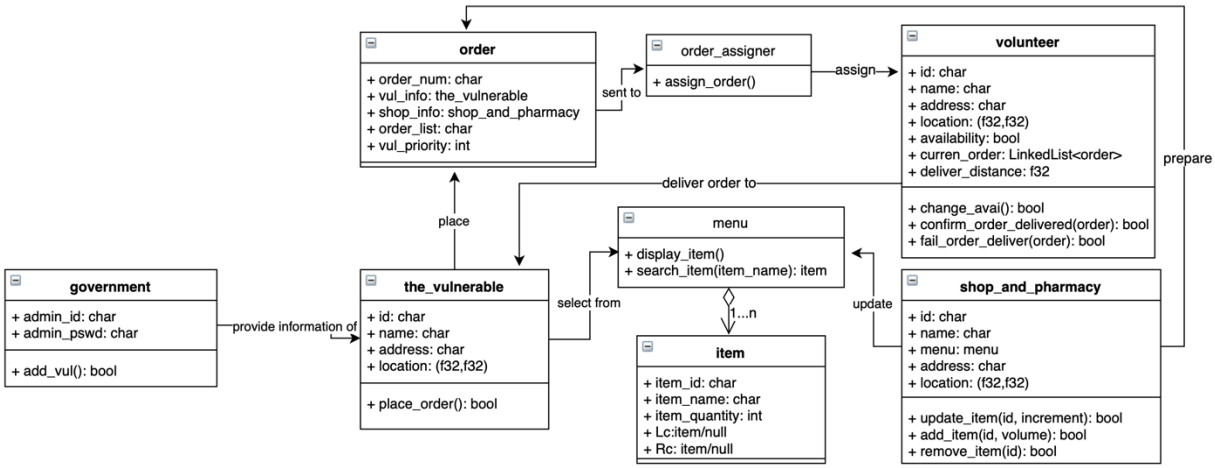
Year 1 UCL-CS Capstone Project Report

Student number: 19077928

1. System Overview

the system provides a platform, which is administrated by the government, for vulnerable people to request essentials from local shops and pharmacies, for registered volunteers to deliver the requested items and for shops to update stocks and await orders.



2. Data Structures

2.1 Users

the vulnerable (vul)		
information	type	description
id	char(8)	8-digit unique id (00000001-99999999)
name	char(20)	
address	char(40)	
location	(float(32),float(32))	(longitude, latitude) of address

The vulnerable is implemented as a hash table of hash (id) = key.

volunteer		
id	char(8)	8-digit unique id (00000001-99999999)
name	char(20)	
address	char(40)	permanent address provided by volunteers when they registered.
location	(float(32),float(32))	(longitude, latitude) of address
availability	bool	the volunteer's availability the day.
current_order	LinkedList<order>	Orders assigned to the volunteer Size cannot exceed 3 (maximum order number)
deliver_distance	float(32)	in km, orders within the distance can be given.

volunteer is implemented as a hash table of hash(id) = key.

shop_and_pharmacy		
id	char(4)	4-digit unique id (0001-9999)

name	char(20)	
menu	SplayTree<item>	the menu tree structure
address	char(40)	
location	(float(32),float(32))	(longitude, latitude) of address

Shop & pharmacy is implemented as a hash table of hash (id) = key.

Data sizes of Hash tables for the vulnerable and shop & pharmacy are predictable, so perfect hash function can be made. For volunteers, the data size is dynamically changing, hash collision may happen. The solution is to make linked lists in hash buckets, the advantage of linked list against linear probing is that it avoids frequent resize and allows for manipulations such as swap and sort, which are useful for data management.

the government		
admin_id	char(8)	
admin_pswd	char(16)	

The government is regarded as the ROOT, to ensure the system security, login is required each time the government gets access to and manipulates data in the system.

2.2 System

order		
order_num	char(8)	provided to the shop when volunteer collect the order.
vul_info	the vulnerable	
shop_info	shop_and_pharmacy	
order_list	varchar	(item_id“-”quantity\r)*, parsed when it is sent to shop.
vul_priority	int(2)	1-3, the day within which order must be delivered, updated automatically per day. Tier 1 order refers to the orders with highest priority

Order is implemented as a priority queue (min binary heap), the comparator compares vul_priority.

1 menu -> n item		
item_id	char(8)	
item_name	char(20)	
item_quantity	int(16)	natural number
Lc	Item/null	
Rc	Item/null	

The menu is implemented as a splay tree, it is of the feature that the more frequently an item is visited, the closer it is to the root. This makes “popular” items displayed ahead in the menu for the vulnerable and shortens the search time of those items for shops & pharmacies.

3. Tasks

3.1 User Actions

o The vulnerable: the main function required is to place order. The system restrains request from one shop per order, working in the same way Deliveroo (a uk food delivery app) does.

1. scan user input of items wanted.
2. store shopping list as a string in form of (item_id“-”quantity\r)*, create an instance of the order.
3. send the order information to the shop for preparation and push it into the priority queue.

o Shop: the main function required is to update stock, i.e., update item quantity, add and remove items. Since the items in the stock is implemented as a splay tree, one can insert, remove and search for an item as they do in a normal splay tree.
o Deliveryman: the main functions are to update their daily availability, to confirm the accomplishment of a delivery and to fail a delivery.

Change availability:

1. scan user input.
2. search for the corresponding volunteer instance in the hash table.
3. change the volunteer's availability.
4. If availability turns to true, move the instance to the first of the linked list in the hash bucket, which makes it easier for "order assigner" to collect all available volunteers.

Confirm order delivered:

1. scan user input of the order delivered.
2. remove the order from current_order.

Fail to deliver an order:

1. scan user input of the order failed.
2. remove the order from current_order, send the order back to "order assigner".

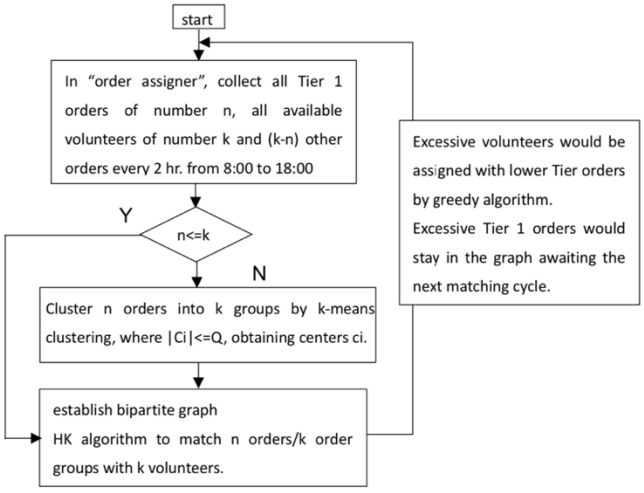
o Government: the main function is to register all the vulnerable citizens. A unique id is generated for each citizen chosen. Their information would be wrapped and inserted in a hash table.

3.2 System Actions

o Order assigner: the main function is to assign orders requested by the vulnerable to volunteers.

1. Collect all Tier 1 orders and available volunteers (with availability true and size of current_order 0) from the corresponding data structures at the moment.
2. Do the matching. The process is explained in-detail below.
3. update volunteers' current_order, pop orders from the priority queue. Orders that fail to get matched are maintained in "order assigner" for next matching process.

Order-volunteer matching is obtained by k-means clustering [1] and Hopcroft-Karp's Hungarian algorithm (HK algorithm) for Tier 1 order, and greedy algorithm for other orders. Its aim is to make as many orders assigned successfully as possible. Order and volunteer are modelled as two parts of a bipartite graph, for order clusters, the order closest to the centroid represents the whole cluster. The matching process is shown in the graph.



Assumptions made in the model:

1. order number and volunteer number are static in the model at the point of matching.
2. the locations of the vulnerable, volunteers and shops are fixed points.
3. Only must orders in Tier 1 be delivered the day, so they are always handled first.
4. orders are prepared and packed by shops immediately after they are placed.
5. orders are delivered to the vulnerable as soon and the same day as they have been assigned.
6. In order to make sure all orders get delivered on time, there is slight possibility that volunteers be given orders that are out of the distance they cover, though the excess would be little.

k-means clustering: the aim is to group objects so that objects within each cluster are as close to each other as possible, and as far from objects in other clusters as possible. The initial centroids chosen makes huge difference in efficiency, so an advanced k-means++ algorithm is used. [2]

Input: From order obtain **X** = {x | x = (shop's longitude, shop's latitude, vul's longitude, vul's latitude)}
the number of available volunteers **k**.
Output: order clusters C
Q is the maximum number of orders a volunteer can be assigned with at one time.
D(x) gives the shortest distance from a data point to the closest center we have already chosen

1a. Take one center c1, chosen arbitrarily from **X**.

1b. Take a new center ci, choosing x∈**X** with probability $P(x) = \frac{D(x)^2}{\sum_{x \in order} D(x)^2}$ by roulette wheel method.

1c. Repeat the step 1b until **k** centers have been taken.

2a. For each i∈{1,...,k}, set cluster Ci to be the set of points in **X** that are closest to ci.

2b. For each i∈{1,...,k}, if |Ci| > Q, points that are further from ci in the cluster Ci are rearranged to the second closer cluster until |Ci| = Q.

3. For each i∈{1,...,k}, set ci to be the center of mass of all points in Ci: $c_i = \frac{1}{|C_i|} \sum_{x \in C_i} x$.

4. Repeat Steps 2a, 2b and 3 until C no longer change.

Hopcroft-Karp's Hungarian algorithm: it works out the maximum matching of a bipartite graph. HK algorithm is superior to the traditional Hungarian algorithm in terms of time complexity. [3]

Graph G = G1 ∪ G2 ∪ {NIL}, where G1 are order vertexes and G2 are volunteer vertexes, NIL is a special null vertex.
Output: match pair p = {(u, pair_G1[u]) | u ∈ G1}

```
def bfs():
    queue q
    for u in G1:
        if pair_G1[u] == NIL
            dist[u] = 0
            q.push(u)
        else:
            dist[u] = inf
    dist[NIL] = inf
    while !q.empty():
        u = q.front()
        q.pop()
        if dist[u]<dist[NIL]:
            for v in adj[u]:
                if dist[pair_G2[v]] == inf:
                    dist[pair_G2[v]] = dist[u] + 1
                    q.push(pair_G2[v])
    return dist[NIL] != inf
```

```
def dfs(u):
    if u != NIL:
        for v in adj[u]:
            if dist[pair_G2[v]] = dist[u] + 1:
                if dfs(piar_G2[v]):
                    pair_G1[u] = v
                    pair_G2[v] = u
                    return true
        dist[u] = inf
        return false
    return true
def hk():
    for u in G:
        pair_G1[u], pair_G2[u] = NIL
    while bfs():
        for u in G1:
            dfs(u)
    return pair_G1
```

o Menu: to traverse and display all items of shops and to search for a particular item.

Display items for a vulnerable person:

1. for each shop within the same borough as the user, its menu is called and traversed.
2. all items of which the quantities are greater than 0 are displayed.

Search for an item the shops may have:

1. scan user's input of an item's name.
2. traverse shops' stocks, displaying all the matched results in stock.

o Distance comparator: locations of the vulnerable, shops and volunteers are refined by longitude and latitude. The distance between two locations is calculated by:

```
Input: x,y ∈ {(longitude, latitude)}
Output: distance d
R is the radius of Earth in km
pi is 3.14159265358
all trigonometric functions take and return angle in radians.
    def get_distance(x,y):
        lon_x = x[0]*pi/180
        lon_y = y[0]*pi/180
        lat_x = x[1]*pi/180
        lat_y = y[1]*pi/180
        c = sin(lat_x)*sin(lat_y)+cos(lat_x)*cos(lat_y)*cos(lon_x-lon_y)
        return d = R*arccos(c)
```

For any volunteer of location u, and any order of vul's location v and shop's location s, if get_distance(u,v) and get_distance(u,s) are smaller than the volunteer's deliver_distance, they can be matched.

A quad-tree structure can be implemented for spatial indexing. It would spare some unnecessary distance measurements.

o ID verification: id verification is required if users would like to use the system.

1. scan user input: id, name.
2. return HashTable[hash(id)].name == name.

4. Analysis

o Place order: the dominant operation is to push an order into priority queue, so O(logn), where n is the size of the queue.

o Update stock: the stock of a shop is implemented as a splay tree, so O(logn) for insert, remove and search.

o Change availability & confirm/fail a delivery: manipulated in hash table, ideally, the complexity is O(1).

o Register the Vulnerable: for each user, O(1).

o Order assigner: for n orders, O(n^2) for graph establishment, O(n^2.5) for HK algorithm, O(n^2) for k means clustering, O(n^2) for greedy algorithm, where volunteer number is assumed to be similar to order number and the iteration time to find centroids in k means clustering is assumed much smaller than n when n goes very big. The dominant part is HK algorithm, so O(n^2.5) is taken for the whole matching process.

Order-volunteer matching can be abstracted as a Vehicle Routing Problem (VRP), i.e., NP hard problem. Simply using greedy algorithm leads to unwise decisions and waste of resources. Bipartite graph matching holds a rather satisfying complexity while it gives much better matching results.

o Menu display: O(n), where n is the number of items.

o Item search: O(mlogn), where m is the number of shops and n is the number of items of one shop.

The reason to use splay tree rather than hash table is that splay tree reveals items' popularity.

o Distance comparator: for each comparation, O(1).

o Id verification: for each user, O(1).

Reference:

[1] LI Tao-ying, LYU Xiao-ning, LI Feng, CHEN Yan. Routing optimization model and algorithm for tak eout distribution with multiple fuzzy variables under dynamics demand, 2019.

[2] David Arthur, Sergei Vassilvitskii. k-means++: The Advantages of Careful Seeding, 2007.

[3] John E. Hopcroft and Richard M. Karp. AN n^5/2 ALGORITHM FOR MAXIMUM MATCHINGS IN BIPARTITE GRAPHS, 1973.

o What happens if requests exceed the capacity of volunteers to deliver, or there are supply shortages? How does your system behave in such cases?

It has been taken into consideration that if requests exceed the capacity of volunteers the orders are grouped by k means clustering and assigned to volunteers, with a maximum order number per volunteer.

Supply shortage has not been discussed in the main paragraphs. The solutions can be:

1. allow user to book the item in shortage, the user would be notified as soon as the item is updated in stock.
2. Staff service could be offered if request is super urgent so that the vulnerable people can place their orders manually by telephone. Then the system would assign specific staff to get the item from all shops in the town or even nearby towns.

o How would you test your system?

To test core functions of the system, i.e., order matching, item displaying and item searching, one can randomly generate simulated data of orders and volunteers with locations within the scope of, for instance, London, and shops and items, then get the functions tested repeatedly to check their feasibility and performance. After tested on simulated data, open a beta version to a small group of testers before it is made public to all. The testers would play roles as the vulnerable, shops and volunteers to see if the system works as expected as a whole and detect bugs. After launched, the system should keep a log record for supervision.

o How quickly could your system be built and deployed? Could it be built by volunteers using available open-source software packages, or would it need professional developers or software development company support?

The build-up of the system involves knowledge that is no more sophisticated than that is taught to first-year students in universities. So, there is little pain to get it done. The package required is the one that obtains longitude and latitude of a given address, and there are plenty of them open-source on the Internet. However, on the other hand, the system collects private information from users, so one should be very careful when looking for the developers. Formal companies and professionals are more reliable regarding security reasons.

o Handling money and payments is not addressed in the specification above. Should the system manage this at all? If yes, what services should it provide?

Having admitted that the system comes out of humanitarian purposes, money should be part of the plan to secure the program in real life. The government could cooperate with banks and PayPal for money management and transaction. For the vulnerable group, they can be given an allowance for free item request every month and pay any extra. Also, the system can offer two delivery options, standard delivery for free and superior delivery, charged. These could avoid undue requests from just one individual and too many orders with the highest priority —— since people all want to get their orders as soon as possible, even though it is not an urgent need. As for volunteers, they can be supported financially, for example, some percentage of traffic fee may be refunded by the government, the most active volunteers can be rewarded with prize. These policies may encourage more citizens to register as volunteers.

o Some user requests will be natural language text. Parsing these inputs to automatically extract data from them could speed their processing. How would you design your system to enable such parsing?

The system is of the feature that items of shops are displayed to vulnerable users, with item names and item information, i.e., item pictures, item descriptions and etc. The vulnerable select items themselves. This ensures that the items ordered are exactly what the user wants. Also, the vulnerable may not remember or fail to provide names of some medicine since they are too long and peculiar to memorize and spell ——— but they can be recognized via pictures and descriptions. In that case, natural language processing seems unnecessary. However, the system also supports item searching by name, so NLP can be involved to optimize the searching process. The main tasks are query understanding and query rewriting, which would allow the system to correct spelling mistakes, cut off redundancy and make synonym expansions for users' query and therefore give back more accurate results intelligently.