Case Study:-
Smart cab allocation system for Efficient trip planning.
Name:- Ramakrishna Raju Alluri

Objectives of case study:

My understanding:- We need to create an framework where we have multiple types of objects; each representing a particular actor in the environment who interact with the app and also we have to optimize and smoothen these interactions so that there is an ease of use and better resourse efficiency.

Main Objects/Actors in our framework:
1)User
2)Cab Driver
3)Admin

Priorities for each:
User:- Ease of use and less waiting time.

Cab driver:- Less travel and waiting time between two rides.

Admin:-Best use of available resources(cabs).

Now as a system developer we also have to take into consideration the number of objects in the framework and take care that whatever algorithm we have taken is optimized for huge scale of operations.

Another important consideration is that if possible we have to make as many small algorithms (which are part of the main algorithm) as possible to be as flexible as possible.

Frame work:-

We have a user , a set of drivers and an administrator.

User:-
1):-Every user logs into the app; We authenticate him and then allow him to enter.H then enters his destination. We also recieve his cartesian co-ordinates with respect to a particular reference system.

Driver:-
1) Every Driver has a geolocation system and after every fixed time interval t he updates his location (in the same cartesian co-ordinates system).
2) He recieves a message from adminstrator about a booking.

Adminstrator:-
1) He recieves requests from user; he finds a set of most optimal cabs and send them to the user.
2) After recieving a the desired cab from the user he as to update the framework to reflect it.
3)He also recieves continuous updates from cab drivers about their location and which he has to take into consideration.

Some Basic Algorithms;

Naive approach :-

1) One naive approach is to apply Breadth First Search algorithm directly on the enitre co-ordinate system. We search the space using find the closest cab and if it fits user requirements select it or else we continue the process of searching untill the we find the next cab.

Drawbacks:-

1) The major problem of this apporach is scaling ie finding the closest cab using BFS algorithm on the enitre co-ordinate space is very time consuming. Most of the time we pass through vertices which do not even have a cab thus wasting computations.
2) If there is a cab that is not providing service for some reason but still is active on the system; every user in the area always recieves it as a possibility untill the adminstrator removes the cab from service.

My Apporach:-

Algorithm description:-

Here I give an overall description of my algorithm.I will explain each term later in detail.

In our algorithm we divide the entire co-ordinate system into Regions.Each User after logging into the app is assigned a Region accoring to the co-ordinate system. Now every Region has a set of drivers who are cuurently active in it. This set of drivers is sent to the User who then sorts them accoring to the destination he wants to got to.When he finally selects a cab the status of cab Driver is updated to include this passenger and his destination.There is an Administrator who takes care of number of cab drivers and other overall specifications of the system.

This is our overall algorithm now smaller algotihms in this include:-

1)How to find the region of user given co-ordinates.
2)How to maintain the set of cabs in a region.
3)How to sort the set of cabs recieved by the user.

Now defining Each term in the above algorithm.

I have set them as classes with attributes:-

Regions:-

```
auto cmp = [](Driver* a, Driver b) {
    return a->timeEmpty>b->timeEmpty;
};
```

```
class Region
{
    ll int regionId;
    vector<Region*> neighbours[n];
    func(int x,int y)
    {

        //Mathematical function which returns a bolean
        //which asserts us if the coordinate is in this region.

    }
    set<Driver*, decltype(cmp)> currentDrivers;
    friend Region* find_region_id(Region* prevRegion,int X,int Y);
    friend Driver* book_a_cab(User A);
    friend void update_regions(Driver* B,int X,int Y);
};
```

Class Region:
Attributes:-
1)RegionId(type:-long long int):- This gives a unique id for each Region.

2)neighbours(type:- vector of pointers to class Region,size:-n(depends on administrator)) :- Consists of neighbours of a particular region. This list is to be defined by the administrator beforehand for every region.It remains constant unless changed by the administrator.

3)currentDrivers(type:-set of pointers to drivers;this also comes with a custom comparator function):-This consists of set of drivers (sorted by some method) int he current region.

Member functions:-
1) Func(co-ordinate x,co-ordinate y) This function uses some algorithm (which is left to the administrator to decide) and return the boolean which tells us if the given co-ordinates are in the given region.

Users:-

```
class User
{
    ll int UserId;
    Region* region;
    ll int X;
    ll int Y;
    ll int X2;
    ll int Y2;
    Region* mostRecentRegionId;
    friend Region* find_region_id(Region* prevRegion,int X,int Y);
    friend Driver* book_a_cab(User A);
    friend void update_regions(Driver* B,int X,int Y);

};
```

Class user:-
Attributes:-
1)UserId(type:-long long int):- Unique id for each user. Asigned after authentication.
2)region(type:- pointer of region class):-Points to the region of the user.
3)X,Y(type:-long long int):-Refers to co-ordinates of current location of the user.
4)X2,Y2(type:-long long int):- Refers to co-ordinates of destination.
5)mostRecentRegionId(type:- pointer of region class):- This pointer points to the region where the user was dropped of last.
This is a valuable addition to our class because when finding the region of the user when he logs in it is highly likley that this region is same as the one he was last dropped of.

Drivers:

```
class Driver
{
    ll int DriverID;
    int status;
    Region* region;
    ll int X;
    ll int Y;
    int timeEmpty;
    int capacity;
    vector<Region*> toRegions;
    friend Region* find_region_id(Region* prevRegion,int X,int Y);
    friend Driver* book_a_cab(User A);
    friend void update_regions(Driver* B,int X,int Y);
};
```

Attributes:-
1)DriverId(type long long int):-unique id of assigned to each driver.
2)X,Y(long long int):-co-ordinates of driver.
3)status(int):- represents the status of driver.
4)timeEmpty(int):-represents the time of idiling of each driver.
5)capacity(int):-capacity of cab.
6)region(pointer to region class):-shows the region the cab is in.
7)toregions(vector of pointers of region class):- shows the destinations of each of the cabs occupants.

Administrator:-

```
class Administrator
{
    ll int AdminId;
    vector<driver*>drivers_current(n1);
    loop:
    {
        //ask every driver for location after time t.
        //update respective objects(regions,drivers) using
        //update_regions function.
    }
    loop :
    {
        //wait for user requests.
        // then use find_region_id to get user regionid
        //then use book_a_cab.
    }

};
```

Attributes:-
1)AdminId (long long int):- unique id for each admin:

Loop1:-
This loop loops over time and after every fixed interval of time t requests every driver to update their location so that it can update their regions.

Loop2:-
This loop lick starts if a user logs in and then enters his destination.

Now various functions that operate on the above classes are:-

find_region_id:-

```
Region* find_region_id(Region* prevRegion,int X,int Y)
{
    if(prevRegion->func(X,Y))
    return prevRegion;
    else
    {
        queue<Region*> q;
        for(int i=0;i<n;i++)
        {
            q.push(prevRegion->neighbours[i]);
        }
        while(!q.empty())
        {
            Region* k=q.top();
            q.pop();
            if(k->func(X,Y))
            {return k;
            break;}
            else
            {
                for(int i=0;i<n;i++)
                q.push(k->neighbours[i]);
            }
        }
    }
}
```

This function (when given a start region and co-ordinates)returns the region the region associated with the given co-ordinates.
When we use it:
1)We use this function to find the region of the user when he logs in.
2)We use this function to constatly update the region of a particular cab.

book_a_cab:-

```
Driver* book_a_cab(User A)
{
    queue<Region*>q;
    q.push(A->region);
    while(!q.empty())
    {
        reqion* search=q.top();
        q.pop();
        auto driverSet = search.region->current_drivers;
        // Among these drivers assign if anyone of them is going to same destination
        // or else going to the nearest area to their destination

        Region* destn = find_region_id(A->region, X2, Y2);
        queue<Region*> q1;
        q1.push(destn)
        while(!q1.empty()){
            auto currDest = q1.top();
            q1.pop();
            for(auto driver: driverSet){
                bool currDestnInDriverstoRegions = false;
                for(auto r: driver.toRegions){
                    if(r==currDestn) {
                        currDestnInDriverstoRegions = true;
                        break;
                    }
                }
                if(currDestnInDriverstoRegions ==true)
                {
                    driver.toRegions.push(destn);
                    return driver;
                }
            }
            for(int i=0;i<n;i++)
            q1.push(currDest->neighbours[i]);
        }

        for(int i=0;i<n;i++)
        q.push(search->neighbours[i]);
    }
}
```

This function (when given a User) takes in the region of user and retuns the set of cabs in that region to the user, then sorts the drivers to find the best driver going to a user destination region(or close to it).

update_regions:-

```
void update_regions(Driver* B,int X,int Y)
{
    auto currRegion = B->region;
    B->region = find_region_id(B->region,X,Y);
    if(currRegion!=B->region)
    currRegion.currentDrivers.remove(B);
    B->region.currentDrivers.insert(B);
}
```

This function (when given the respective driver and current co-ordinates) updates the region of the driver in driver class; if the region has changed it removes the driver from the previous region and updates in the new region.

Web app :-

USER INTERFACES:

LOGIN PAGE:

1. Any unfamiliar user attempting to access the website's home page will be automatically redirected to the login page.
2. The login page incorporates a sign-in option, enabling users to create their credentials.
3. During the sign-in process, users are required to input a unique phone number and a robust password. The password criteria include at least one capital letter, one special digit, and one special character.
4. Passwords are securely stored using bcrypt, a one-way hashing method. This involves generating a 24-byte hash by applying a 16-byte random salt value to the password string, which can be a maximum of 72 bytes. Storing the hash instead of the raw password string enhances security, especially in the event of system information leaks. Bcrypt's deliberate slowness and computational expense provide added protection against brute force attacks.
5. Upon successful sign-in, users are redirected to their respective home pages.

EMPLOYEE:

1. After signing in, if the user is identified as an employee, they are directed to the employee home page.
2. The employee home page features options to search for a cab by providing the destination they plan to travel to.
3. Once the employee provides a destination, the system provides real-time information on optimal nearby drivers and their distances.

DRIVER:

1. After identification as a driver, the user is redirected to the driver home page. The system loads details about the assigned employee, their destination, and the driver's current location.
2. Drivers receive real-time notifications about new employee assignments and relevant details.

ADMIN:

1. If the user is recognized as an admin, they are directed to the admin page.
2. The admin interface includes provisions for adding drivers or employees to the system.
3. Admins have access to information about the location of all drivers, their destinations, and their assigned users.
4. The admin also has real-time access to information about all users utilizing the cab service, along with their details.