1 Algorithm Design

The most relevant algorithm of the myTaxiService application is the one implementing the queue management. It is important to optimize its implementation as long as it is the most complex and the one that distinguishes the system. Another significant algorithm is the one that manages the forwarding of the requests of the users to the taxi drivers.

- **Initialization.** The class which manages the areas distribution and the queues of the taxi driver also deals with the initialization of the queues of the taxis. It starts assigning to every area as many taxi as the average number of the requests expected in that area, based on the statistics. Once all the areas have been filled with their optimal number of taxis, the remaining taxi drivers are sent to the areas that are most needy.
  The "needyArea" function, in fact, returns the index of the most needy area at the moment, calculating the area that has the maximum difference between the maximum number of requests recorded and the average of requests.

  COMPLEXITY: O(T) with T = number of taxis.

Algorithm 1 Initialization

```
procedure initQueues(TaxiDriver[] taxiDrivers, Area[] areas)
    $i \leftarrow 0$
    $j \leftarrow 0$
    for $i <$ areas.size() do
        $k \leftarrow 0$
        for $k <$ areas[i].getAverage() do
            areas[i].addToQueue(taxiDrivers[j])
            $j \leftarrow j + 1$.
        end for
    end for
    if $j <$ taxiDrivers.size() then
        for $j <$ taxiDrivers.size() do
            areas[needyArea(areas)].addInQueue(taxiDrivers[j])
        end for
    end if
end procedure
```

- **Manage requests.** After the initialization, the taxi drivers are able to take requests. If they accept the request, the function sets their availability to false and stops monitoring their position. It is also possible that a taxi driver declines the request or is not able to give an answer to the server. In this case, the first taxi driver is moved to the last position of the queue and forwards the request to the second of the queue, who is now moved to the first position. This action is iterated in the queue of that area until a taxi accept the request. After accepting the call, the taxi driver is set unavailable and he is deleted from the queue, as it is likely that he will end up in another area.

  COMPLEXITY: O(A+Q) with A = number of areas and Q = number of taxi enqueued in that area.

Algorithm 2 Manage requests

```
procedure MANAGEREQUEST(Position pos, Area area)
    $answer \leftarrow$ "no"
    $timer \leftarrow$ new Timer(60)
    while $answer$ = "no" || $answer$ = "timeOut" do
        sendRequest($area$.getFirstOfQueue(), $pos$)
        $answer \leftarrow$ waitAnswer($timer$)
        if $answer$ = "no" || $answer$ = "timeOut" then
            $area$.moveToEndOfQueue()
        end if
    end while
    $area$.getFirstOfQueue().$setUnavailable$()
    $area$.deleteFirstFromQueue()
end procedure
```

- **Manage queues.** The areas of the city are represented by a graph with an array of adjacencies, which is a useful representation in order to decide how to distribute and move the taxi drivers within the areas.
  In the first place the function search for the area where the taxi is in and sets the "startingArea" variable to the index of that area, then adds the taxi driver to the queue of the area if it has not reached the maximum number of taxi. If the area is completely full it is necessary to iterate on the areas, exploiting a breadth-first search algorithm.
  The function searches for adjacent areas with the number of taxi lower than the average of the requests and, if it does not find any, it looks for adjacent areas with the number of taxi lower than the maximum allowed for that area.
  If these two conditions are not supplied, the function assign this taxi driver to the first adjacent area visited and takes from that area the last taxi of the queue. This "rejectedTaxi" is moved to an adjacent needy area, repeating the iterations done before, searching needy areas and moving taxi drivers, until it is possible to reach an equilibrium state.
  It is always feasible to exit the while cycle because the number of taxi driver can not be the maximum for all the areas, as it is not cost efficient. The system only ensures that the minimum number of taxi drivers per area is guaranteed.

  COMPLEXITY: O(A) with A = number of areas.

Algorithm 3 Manage queues

```
procedure MANAGEQUEUE(Area[] area, TaxiDriver taxiDriver)
    for $i < area$.size() do
        if $taxiDriver.position$ is in $area[i]$ then
            $startingArea \leftarrow i$
        end if
    end for
    if $area[startingArea]$.getNumOfTaxi() $< max[i]$ then
        $area[startingArea]$.addToQueue($taxiDriver$)
    else
        for $i < area$.size() do
            $area[i].distance \leftarrow$ INFINITY
        end for
        $area[startingArea].distance \leftarrow 0$
        $unvisitedQueue$.enqueue($area[startingArea]$)
        $rejectedTaxi \leftarrow taxiDriver$
        while !$unvisitedQueue$.isEmpty() do
            $tmpArea \leftarrow unvisitedQueue$.dequeue()
            for $i < tmpArea.adjacencies$.size() do
                if $tmpArea.adjacencies[i]$.getNumOfTaxi() $< avg[i]$ then
                    $tmpArea.adjacencies[i]$.addToQueue($rejectedTaxi$)
                    return
                end if
            end for
            for $i < tmpArea.adjacencies$.size() do
                if $tmpArea.adjacencies[i]$.getNumOfTaxi() $< max[i]$ then
                    $tmpArea.adjacencies[i]$.addToQueue($rejectedTaxi$)
                    return
                end if
                if $tmpArea.adjacencies[i].distance =$ INFINITY then
                    $tmpArea.adjacencies[i].distance \leftarrow tmpArea.distance + 1$
                    $unvisitedQueue$.enqueue($tmpArea.adjacencies[i]$)
                end if
            end for
            $firstUnvisited \leftarrow unvisitedQueue$.getFirstOfQueue()
            $rejectedTaxi \leftarrow firstUnvisited$.addToFullQueue($rejectedTaxi$)
        end while
    end if
end procedure
```