ACS-Algorithms

March 2019

1 Introduction

Two self-organization solutions have been proposed in this work. The first one created is a centralised version, in which a central node with all the knowledge acts as a manager of the system and schedules lifts. However, such a solution would not be recommended for a real life application. Having a single centre of information and computation, exposes the system to the possibility of bottlenecks when the computation is not supported adequately or when it is not able to adapt promptly to a rapid raise in requests. Moreover, if such unexpected situation were to happen, it may also be possible for the central node to crash, hence the whole system would crash and shut down and the service offered would not be available any more since there are not replicas that could rapidly help the system overcoming the crash of the main node.

For these reasons, a second, decentralised version has been created. Aiming at alleviating the problems derived from centralisation, the computational power and information are distributed among various nodes, the autonomous vehicles. This leads to the necessity of a more complex logic, yet, since the information that each car will store will be a fraction of all the information in the system, the less resources will be needed and the system will be more flexible in case of a possible high number of users in the system.

In the remainder of this document, first the centralised solution will be introduced, along with the algorithm used for the grouping cost computation implemented for this version. The following section will illustrate the decentralised solution and the related algorithms implemented to group the passenger, compute the initial costs and then recompute them, if there have been changes on the road.

2 Centralised Solution

The first attempted solution consisted in a simulation were people looking for a lift at the same time were grouped in a centralised fashion. As a matter of fact, the global host, which in reality would correspond to a central node, contains all the knowledge about people, roads and cars. It automatically collects information about the people location and whether they are searching for a lift. Then proceeds to create groups composed by a maximum of 5 people in close proximity and assigns them to the closest car. The car will then enter a state in which it will wait between one and two minutes in order to be assigned other passengers and, hopefully, reach full occupancy. After this interval has passed, the car will compute the shortest path starting from its current location, passing through the passengers locations to pick them up and finally drop them off at

their destinations. Therefore, it will be able to notify each passenger about the time they will have to wait for the car to reach them and also the time necessary to reach their destination, along with the cost for the travel.

2.1 Grouping algorithm of the manager node

As it was previously mentioned, in this first solution, the task of grouping people together with a car is a task assigned to a central, or manager, node. This node corresponds to the global host in the GAMA platform and has knowledge of all the species and their instances created. Therefore, it can group all people into groups based on their location and a given distance. It can therefore separate groups with more than five people into smaller groups that could fit into a single car. In addition to this, it knows the states of the autonomous vehicles that are wandering in the world of the simulation and, accordingly to this information, it can discern the cars which would be able to accept other passengers. Using these two abilities, the manager nodes creates the inputs of the first algorithm for grouping: a set of groups of people, that will be referred to as G, and a list of cars (*cars_avail* in the *pseudo-code* reported below) that could take the people the manager will propose on the base of some criteria that will explained in the following paragraph.

As a matter of fact, after creating the set of groups, if there are cars available, the manager node will start a loop over the groups. First, it will store information about the destinations of the components of the group, (lines 3-7) then it will explore the cars available, namely those who are either wandering or waiting for other passengers, looking for the closest one, by ordering them on the base of the length of the path from their respective location to the position of the group, as can be seen in line 2. Starting from the closest one, the manager will try to assign the group to the car. If the car is empty, this assignment will succeed, otherwise, as represented by the lines 13 and 14, the angle between the location of the new group and the location of the group already assigned to the car will be checked together with the time needed to reach the former from the latter, that should be less that about 3 minutes (line 16). This inquiry is necessary in order to avoid that the groups that will be together in the same car are too distant from one another or in completely different neighbourhoods.

If this control is passed, a check on the destinations will be made, as can be seen in the lanes from 17 to 25. As a matter of fact, it will be inspected whether the set of destination already assigned to the car and those of the new group are close to each other. In order to do so, a copy of the list of the passengers is created and the centroid between the destinations already assigned to the car will be computes as the mean of the locations of the destinations. If the destination considered is within a 500 meters radius from the centroid, it will be considered as valid and therefore added to the list of destinations of the car, following the proper controls, otherwise, the people that have such destination will be removed from the group.

Finally, if the group has passed the checks, the agents in the copy will be added to the passengers of the car and removed from the group g and the loop over the cars available will be broken. On the other hand, if it was not successful and there are other cars available the same procedure will be applied with the second closest car and so on.

Algorithm 1: Centralised grouping algorithm

```
Data: G ← g_1, g_2, ..., g_n;
      cars_avail ← car where (c.state=='wander') or (c.state=='still_place');
Result: As many passengers as possible are assigned to cars, on the base of their location
        and destinations
foreach group in G do
    cars_avail ← (c ∈ cars_avail where ((c.seats_avail > 0) and path(c.loc, g.loc))) sort_by
     |(path(c.loc, g.loc))|;
    foreach p ∈ g do
        if ∃ g.destinations[d] then
            g.destinations[d] ← g.destinations[d]+p;
        else
            g.destinations ←<p.dest, [p]>;
    foreach c in cars_available do
        if c.to_pickup == [] then
            angle ← 0;
            time ← 0;
        else
            angle ← angle_between (g.loc, c.loc, first(c.to_pickup).loc);
            time ← time(path_between(last(c.to_pickup), g.loc));
        copy ← g;
        if angle < 30 and time < 180" then
            foreach d in g.destinations.keys do
                centroid ⟵ mean(foreach p in c.to_pickup collect p.dest);
                if distance(d, centroid) < 500 meter then
                    if d ∉ c.to_drop.keys then
                        c.to_drop ←<d, copy.destinations[d]>;
                    else
                        c.to_drop[d] ← c.to_drop[d] + copy.destinations[d];
                else
                    copy ← copy - (p where p.dest == dest);
            if copy != [] then
                c.origins ← c.origins + copy.loc;
                c.to_pickup ← c.to_pickup + ¡copy.loc,copy¿;
                g ← g - c;
                break;
```

2.2 Paths generation and cost computation

After being assigned the passengers and having waited between 1 and 2 minutes, the car will compute the path to first pick up the passengers and then to drop them off. In the first case, it is

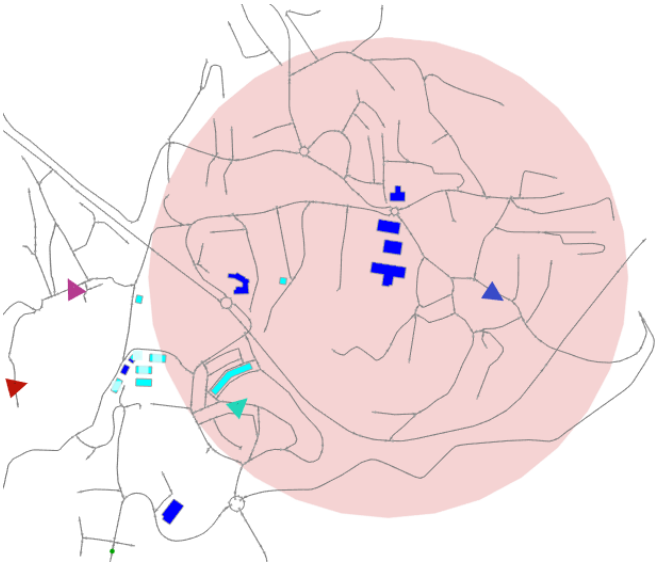

Figure 1: Example of a radius of 500 metres for a given working building

assumed that the locations of the passengers are quite near to each other, hence the passengers are picked up in the order in which they were assigned to the car. On the other hand, the destinations will be first sorted by the length of the path from the last origin, as can be seen in line 1. This operation is done under the assumption that since all destinations are near, it should be better to reach the area of the destination through the shortest path and then proceed to drop off all other passengers.

Algorithm 2: Centralised algorithm for path creation and computation of time and costs for path

Data: O $\leftarrow o_1, ..., o_j$ D $\leftarrow d_1, ..., d_k$ P $\leftarrow d_1, ..., d_n$
Result: D ordered costs_passenger costs_legs

```
1  D ← D sort_by |path(o_j,each)|;
2  time_to_p ← 0;
3  leg ← path(c.loc, o_0);
4  time_to_p ← time_to_p + time(leg);
5  waiting_time ←<to_pickup[o_0],time_to_p>;
6  for i ← 0 to |O| − 1 do
7      people_on ← people_on + |to_pickup[o_i]|;
8      leg ← path(o_i, o_{i+1});
9      time_to_p ← time_to_p + time(leg);
10     waiting_time ← <to_pickup[o_{i+1}),time_to_p>;
11     cost ← |leg|/1000*costkm;
12     cost_legs ← <[o_i, o_{i+1}], [cost, people_on, time(leg)]>;
13 i ← 0;
14 if o_j!=d_1 then
15     people_on ← people_on + |to_pickup[o_j]|;
16     leg ← path(o_j, d_0);
17     cost ← |leg|/1000*costkm;
18     cost_legs ← <[o_j, d_0], [cost, people_on, time(leg)]>;
19 else
20     people_on ← people_on + |to_pickup[o_j]| - |to_pickup[d_1]|;
21     leg ← path(o_j, d_1);
22     cost ← |leg|/1000*costkm;
23     cost_legs ← <[o_j, d_1], [cost, people_on, time(leg)]>;
24     i ← 1;
25 while (o_j! = d_1) ? i < |D| − 1 : i < |D| − 2 do
26     people_on ← people_on - |to_drop[d_i]|;
27     leg ← path(d_i, d_{i+1});
28     cost ← |leg|/1000*costkm;
29     cost_legs ← ¡[d_i, d_{i+1}], [cost, people_on, time(leg)];
```

The operations reported in the line from 2 to 5 are the instantiation of a variable, time_to_p used, to keep track of the time needed by the car to reach the passengers, the computation of the path between the car location and the location of the first group of passengers, the update of time_to_p on the base of the path created and the storing of this last information for later use together with the list of passengers that will wait for such time. In the lines from 6 to 13, similar operations are made. the car will loop over the origins and:

line 7 update the number of passengers on the car. In order to do so, the car checks the information concerning the passengers that get on at each origin. Considering that the computation of the information related to path the path from the car location to the first origin is done before the loop, the first value of `people_on`, the variable containing the number of passengers on

the car, will be the number of passengers that boarded the car at the first origin. In the later repetitions of the loop, the variable will be updated by adding the number of passengers to be picked up at the new origin to the current value.

line 8 the path between the supposedly reached origin, o_i, and the following origin, o_{i+1}, is computed.

line 9 the time supposedly needed to reach this new origin is computed and added to the variable `time_to_p`, that will now contain the time needed by the car to reach the passengers at origin $i + 1$.

line 10 this information is added to the map `waiting_time` that will be used later to give to each passenger information about the time they will have to wait for the car.

line 11 the final cost of covering the leg of the path is compute as follows:

$$cost = \frac{\frac{length(path)}{1000} \times cost_{km}}{|peopleonboard|} \tag{1}$$

However, only the actual cost of covering the leg:

$$cost = \frac{length(path)}{1000} \times cost_{km} \tag{2}$$

will be stored in `cost_legs`. The reason for this choice will become clear in the following subsection 2.3.

line 12 the cost obtained at the previous line, along with the current number of passengers on the car and the time needed to cover the leg, is stored in the variable `cost_path` that will be later user to compute costs and times for the passengers.

Similarly to what done before the loop over the origins, the stretch of road between the last origin and the first destination is computed before the loop over the destinations, as shown in the lines from 13 to 24. In both cases, it was necessary in order to be able to record all the information about the path also in the cases where the car had to pick up only one passenger from one location and then drop it to its destination. However, in this case, it is necessary to check that the last origin is not equal to the first destination. In this case, the number of people on the road is updated for the last time by adding to it the passengers picked up at the last origin. Then the path is computed together with the related cost and time needed, and everything is added to `cost_legs`. In the second case the number people on the leg will be updated by both adding the passengers picked up at the last origin and subtracting the passengers dropped at destination d_1. Moreover the leg that will be computed is between the last origin and the second destination.

The final lines of the pseudo-code, from 25 t 29, are a loop over the remaining destinations. Here, the number passengers on the road is updated by subtracting, instead of adding, the number of passengers who dropped of at d_i. Finally, once again all the information about cost, time and people on the leg are added to `cost_legs`.

The output of all these operations will be cost_legs, were all the information about the segments composing the path of the car with regards to costs, people on board and time needed to cover each segment is stored.

2.3 Computing costs for passengers

After having computed all the segments of the path and their corresponding costs and time for covering, this information will be reported to the passengers. In order to do so the algorithm 3 has been implemented. It consists in a loop over all passengers p assigned to a given car. The variables `cost` and `time_lift` are initialised to 0, whereas the variable `time_wait`, representing the time the passenger will have to wait for the lift, is set with the value of the map `waiting_time`, introduced in Algorithm 2, for the key containing the passenger p. Following this initialisation, there is a loop over the keys of `cost_legs`. The keys that are important in order to compute the cost for the considered passengers are those comprehending those having:

1. the starting point of the passenger as first key (line 7),
2. the destination of the passenger as second key (line 15),
3. all the keys in between the two reported before, if they exist.

Therefore, when the first is found a boolean, `next_too`, it set to *true*, in order to consider also the following keys. Therefore, the time needed to cover the legs and the cost for the leg is computed, according to the equation 1, and added respectively to the `time_lift` and to the `cost` variables for each leg in between the first and the last in which the passenger is on the car (lines from 8 to 14). However, after this has been done for the last leg, namely the one having as second key the destination of the passenger (line 15), the loop is broken and the information computed is stored into the `cost_passengers` map, having as keys, the passengers names and as values, lists containing the cost and the time needed for the whole trip of the passenger, and the time it will have to wait in order for the car to reach it.

Algorithm 3: Computation of costs for each passenger

```
   Data: cost_legs
         p ∈ P
   Result: cost_passengers
 1 foreach p ∈ P do
 2     cost ← 0;
 3     time_lift ← 0;
 4     time_wait ← waiting_time[contains p];
 5     next_too ← false;
 6     foreach key ∈ cost_legs do
 7         if key[0]==p.origin then
 8             next_too ← true;
 9             cost ← cost + (cost_legs[key][0]/cost_legs[key][1]);
10             time_lift ← time_lift + cost_legs[key][2];
11         if next_too then
12             cost ← cost + (cost_legs[key][0]/cost_legs[key][1]);
13             time_lift ← time_lift + cost_legs[key][2];
14             if key[1]==p.dest then
15                 break;
16         cost_passengers ← cost_passengers + < p.name,[cost, time_wait, time_lift] >;
```

3 Decentralised Solution

In the decentralised version of the simulation, the computational power regarding the grouping of agents and the computation of the costs for each trip is situated in the autonomous vehicles. As a matter of fact, each autonomous vehicle is capable of finding passengers on the road it is currently travelling on and automatically check whether to offer them a lift by verifying if certain conditions are met. When the first passengers are found, an initial path to follow is computed. In the presence of other passengers on the path, the car will decide whether or not to offer a lift to the passengers on the base of the objectives of the passengers already on board.

In the following subsections, the algorithms for the computation of an initial path on the base of the first passenger or passengers found and the computation of the related costs will be described, followed by the description of the process that is activated once passengers are found on the road.

3.1 First passenger or passengers group selection

When the first passengers are found, one of them is chosen of the base of the aerial distance between its target and the location of the car. It was chosen to select the passenger whose destination is the farthermost away, with the hope that it would be possible to add more passengers on the road. Therefore, by adding a passenger or a group of passengers, the car will have a first final objective, their destination, on the base of which it can select passengers when they are found on the road. In fact, they will be filtered on the base of the direction of their destination: the angle between the car location, the already chosen destination and their destination will be computed and, if greater than 20, they will be not be considered and therefore removed from the list of possible additional passengers. (Algotithm 4)

Algorithm 4: Removal of passengers due to discrepancy in direction of destinations

```
   Data: P ← p_1, ..., p_n ;
   Result: P' ⊂ P where p ∈ P' are going towards a similar destination
1  foreach p ∈ P do
2      if angle_between(last(targets).loc, origin, p.dest)>20 then
3          P ← P - p;
4      else
5          if p.dest ∈ c.destinations.keys then
6              c.destinations[dest] ← (c.destinations[dest] + p.dest);
7          else
8              c.destinations[dest] ← <p.dest, [p]>;
```

Once this first selection is made, the algorithm tries to place the destination of the remaining passengers in the list according to the aerial distance from the current location. Therefore, the *initial path* between the origin and the destination of the first passenger, or passengers, is computed along with its length (line 2 and 3), the available places, `open_seats`, will be initialised as the difference between the maximum number of passengers that the car can hold and the number of passengers chosen as first passengers. Therefore a cycle over the destinations of the remaining

possible passengers, `destinations.keys`, is started and for each destination $d \in destinations.keys$:

lines 6 and 7 the deviation legs to reach d are computed. If a new destination has yet to be added, they will be the one from the *origin* to d and the one from d to the destination of the first passenger, otherwise, the first one will be substituted by the one from the last added destination to d.

line 8 the sum of the length of these two paths if compared with the length of the initial path.

lines 23 to 25 If the length of the deviation exceeds the one of the initial path, the passengers with d has destination will be removed from the list of possible passengers and the destination will be removed from the list of destinations.

lines 9 and 10 On the other hand, if the deviation is shorter than the *initial path* it will be inserted in the second last position.

lines 11 to 16 A loop over the chosen destinations is started in order to update the length of the path with which the next deviation will be confronted.

lines 17 to 21 Finally, a check on the remaining places on the car is made. First, if the passengers dropping off at destination d are more than the available seats, only the appropriate number will be kept. Then, the remaining available seats will be updated by subtracting these passengers to the current value. Lastly, if there are not any remaining seats left, the loop will be broken.

Algorithm 5: Decentralised algorithm creation of an initial path after the selection of the first passenger or first group of passengers

```
   Data: P' ← p_1, ..., p_n (output of algorithm 4)
         targets ← origin ∪ d of first group destinations.
   Result: P"← p ∈ P that boarded the car;
         targets ← origin ∪ destinationsofp ∈ P''
 1 i ← 0;
 2 original ← path(origin, first(d));
 3 length_or ← |original|;
 4 open_seats ← max_pass - |firsts|;
 5 foreach d ∈ destinations.keys do
 6     dev1 ← path(targets[i], d);
 7     dev2 ← path(d, last(targets));
 8     if |dev1| + |dev2| < length_or then
 9         index ← |target|-2;
10         targets[index] ← d;
11         j ← 0;
12         length_or ← 0;
13         foreach t ∈ targets-origin do
14             leg ← path(targets[j], t);
15             length_or ← length_or + |leg|;
16             j ← j+1;
17         if |destinations[d]| > open_seats then
18             destinations[d] ← destinations[d][0, open_seats-1];
19         open_seats ← open_seats - |destinations[d]|;
20         if open_seats=0 then
21             break;
22         i ← i + 1;
23     else
24         P ← P - destinations[d];
25         destinations.keys ← keys - d ;
```

3.2 First path costs computation

Once the stops to be made by the car to drop off the passenger are ordered, it is possible to compute the various legs of the car's travel and compute the related costs. Moreover, it is possible to easily compute the cost for each passenger by looping over the *targets*, namely the origin of all the passengers and the destinations of the passengers. First of all, the path between each couple of stops is computed (line 7), then the cost of the leg is computed and summed to the ones from the previous legs in order to have a cumulative value of cost (lines 8 and 9). In fact, since all passengers have the same origin and at each destination d some passengers will have reached their destination, they will abandon the car. Therefore, the cost for each of them will be the sum of the costs of each leg from the origin to their destination divided by the number of passengers still on board in the given leg. Thus, at each cycle the cumulative cost will be assigned to the passengers dropping of at

d (line 10). All the information about the leg will be stored into `cost_legs` (line 11) and finally the number of passengers will be updated for the next leg by removing the passengers that will drop off at the stop d (line 12).

Once this process is complete, the car will change its state to *moving* and will start moving towards the first destination in the list. In the following subsection the process of adding new passengers and updating the costs will be described.

Algorithm 6: Algorithm for the computation of the costs for the first group of passengers

Data: P $\leftarrow p_1, ..., p_n$ *(output of algorithm 4)*
targets: destinations of passengers already chosen. Initialised with [origin, first group destinations].
Result: `cost_legs` for the initial path

```
1  i ← 0;
2  time ← 0;
3  distance ← 0;
4  cost ← 0;
5  p_on ← |P|;
6  foreach d ∈ targets-origin do
7      leg ← path(targets[i], d);
8      cost ← |legs|/1000 × ∗ cost_km ;
9      costs ← costs + cost ;
10     p_costs[destinations[d]] ← costs;
11     cost_legs ←<[targets[i],d],[cost, p_on, time(leg)]>;
12     p_on ← p_on - |destinations[d]|;
```

3.3 Addition of passengers on the road

While the car moves towards the first destination on its list, it is possible that there may be passengers on the road that are looking for a lift. In the event of such scenario, the car will have an initial list of new potential passengers. This list will be first filtered by the angle between the passengers destinations and the origin of the travel (line 1), as in the case of the creation of the first path (see Algorithm 4). Then, it will be filtered on the base of the stops already visited. In fact, there may be passengers that have to go to a destination that the car has already visited. This will mean that the car will be going back instead of going forward. Therefore, the stops already visited, namely those between the origin, index 0, and the next stop planned, with index `at_i` (line 2), along with the new origin, namely the current location of the new passengers and of the car, are removed from the initial list of potential new destinations whereas the related passengers are removed from the list of new potential passengers (lines 4 to 6).

Algorithm 7: Removal of passengers due to direction and previously visited stops

Data: P $\leftarrow \{p_1, p_2, ..., p_n\}$;
destinations $\leftarrow$ d of $\forall p \in P$;
new_or
Result:

```
1 rem_angle destinations ← P group_by (p.dest);
2 at_i ← stops index_of next_stop;
3 backwards ← stops[0, at_i-1] + new_or;
4 if destinations ∩ backwards ≠ ∅ then
5 |   foreach k in destinations ∩ backwards do
6 |   |   destinations ← destinations - destinations[k];
```

If after this first filtering, there are still new passengers, the origin is added (Algorithm 8) to the to the list of stops of the car right before the next target (lines 5,6), if it was not already inserted in the previous position, in which case it will be noted that the origin already is in the list of stops (line 8).

Algorithm 8: Insertion of the origin in the list of the stops

Data: stops;
Result: new_or $\in$ stops

```
1 at_i ← stops index_of next_stop;
2 origin_exists ← false;
3 or_ind ← at_i-1;
4 if stops[origin] ≠ new_or then
5 |   stops[at_i] ← new_or;
6 |   or_ind ← at_i;
7 else
8 |   origin_exists ← true;
```

After the insertion of the new origin in the list of global stops, the path from the current location to the destination of the first passenger that boarded the car among those that are on board is computed (Algorithm 9). Moreover, the time remaining before the starting hour of the shift of the agent and the current hour is computed. These computations are done only when the passenger's is going to his or her working place and therefore has a strict constraint about time.

Algorithm 9: Computation of time for the path of the first added passengers that is at the time in the car

```
   Data: first;
   Result: original
1  if first.next_state == 'go_work' or first.next_state == 'working' then
2      av_time ← first.start_work_time - current_hour;
3      n ← targets index_of f.dest;
4      original ← null;
5      for j=0 to n do
6          leg ← null;
7          if j ==0 then
8              leg ← path(c.loc, targets[j+1]);
9          else
10             leg ← path(targets[j], targets[j+1];
11         original ← original + leg;
12 else
13     av_time ← ;
```

After all the information necessary have been collected and the necessary checks have been concluded, it is possible to start a loop over the new potential destinations in order to understand whether it is possible to include them as future stops or if the passengers should either wait for another car or go alone to their destination (Algorithm 11). In order to do so, the remaining seats are computed and the loop is started. First of all, the last index of the cars *targets* is noted, (line 7 of Algorithm 11), then a check (Algorithm 10) is made over the length of the passengers dropping of at d: if there are too many passengers only as much as needed to cover the open seats are considered (Algorithm 10, line 2). If there are not passengers left (Algorithm 10, lines 3,4) or if indeed there are no open seats left (lines 8,9 of Algorithm 11), the cycle is broken. If this is not the case it is checked whether or not the destination of the passengers is already among the next stops and the passengers are added (lines 11,12).

Algorithm 10: Check on remaining available seats in the car and possible break of the loop

```
   Data: destinations;
   open_seats;
   Result: |destinations[d]| == open_seats
1  if |destinations[d]| > open_seats then
2      destinations[d] ← destinations[d][0, open_seats-1];
3  if |destinations[d]| == 0 then
4      break;
```

Otherwise, a loop is started (line 14 to 38) over the next stops in order to try to insert the destination at the appropriate index. First of all, the path between the the current target and the next target and the path between the current target and the destination considered are computed (lines 15 to 20). In the case in which they are being computed for the first time, their starting

point will be the new origin, new_or (lines 16, 17), otherwise it will be the destination at index $i-1$ (lines 19,20). This distinction has been necessary since the origin is not in the list of targets, but only in the list of global stops. The length of these two paths are compared and on the base of this comparison different choices are made:

lines 33 to 37 If the length of the path from the new origin, or the target at index $i-1$, to the the destination d is longer than the current path starting from target at index $i-1$ and ending at the target at index i, the counter i will be incremented and the loop will start again considering the following targets. In the case in which the value of i corresponds to the last index of the targets, the destination can be added as the last stop.

lines 22 to 32 Otherwise, if the length to the destination d is shorter than the one to the next stop, the path from the destination to the next stop is computed (line 22). Before the addition of the passengers and of the destination, if the first passenger that boarded the car among those still on board is going to work (line 24), some additional checks are made. The approximated time needed to cover the path to the destination of the first passenger if the path variation is introduced is computed. It is done by subtracting the approximate time needed to cover the leg that may be substituted from the approximate time needed to cover the path as it is, to which the approximated times needed to cover the leg between the target and the destination and the leg between the destination and the next target are summed (line 25). Thereafter, it is checked whether the new time needed is less than the time the first passenger has left before the starting of his or her shift at work and whether takes less that 1,5 than the time needed with the original path, in order not to lengthen the path too much. This information is stored in the boolean *check* (line 26) which is instantiated as true (line 23). On the base of the value of *check*, the passengers will be added. In the case in which the passengers' destination was added right before the current next stop, therefore becoming in turn the next stop to reach, the current path that the car is following will be recomputed to reach this new stop. Otherwise, once the car has completed the insertion of the stops and all the new passengers that meet the condition have boarded the car, it will continue to follow the path on which it stopped for the pick up.

Finally after each cycle of the loop over the destinations, the passengers that have just boarded the car will be removed from the list of potential new passengers (line 40). Lastly, if there have not been any additional passengers at the current location, the new origin previously added to the global list of stops will be removed from it, unless its the origin of the travel since it will be needed for the check on the direction of the passenger destination (lines 41 to 43).

Algorithm 11: Decentralised algorithm for adding passengers after the car has taken a first passenger or group

```
   Data: P;
   targets;
   original;
   Result: p ∈ Passengers;
   updated stops;
   updated targets ;
   updated cost_legs;
1  tot_added ← 0;
2  up_costs_pass ← false;
3  added ← false;
4  open_seats ← max_pass - |passengers|;
5  foreach d ∈ destinations do
6      i ← 0;
7      max_ind ← |targets|-1;
8      if open_seats==0 or |destinations[d]| == 0 then
9          break;
10     else
11         if d ∈ targets then
12             Add_Passengers;
13         else
14             while added == false do
15                 if i == 1 then
16                     t2d ← path(new_or, d);
17                     t2n ← path(new_or, targets[i]);
18                 else if i < max_ind then
19                     t2d ← path(targets[i-1], d);
20                     t2n ← path(current_stop, targets[i]);
21                 if |t2d| < |t2n| then
22                     d2t ← path(d, targets[i]);
23                     check ← true;
24                     if 'work' in first.next_state then
25                         w_change ← t_a(original) - t_a(t2n) + time_a(t2d) + t_a(d2n);
26                         check ← w_change < t_a(original)*3/2 and av_time ≥ w_change;
27                     if check then
28                         Add_Passengers;
29                         if i==1 then
30                             Change_Path;
31                     else
32                         break;
33                 if i == max_ind and !added then
34                     targets ← targets + d;
35                     stops ← stops + d;
36                     Add_Passengers;
37                     as_last ← true;
38                 break;
39             i ← i+1;
40         P ← P - destinations[d];
41 if tot_added == 0 then
42     if stops index_of new_or != 0 then
43         stops ← stops - new_or;
```

Algorithm 12: Addition of new passengers

Data: destinations;
Result: addition of new potential passengers to the set of actual passengers
1 dest_ind ← stops index_of d;
2 open_seats ← open_seats - | destinations[d]|;
3 tot_added ← tot_added + | destinations[d]|;
4 `Update_Cost_Legs`;
5 passengers ← passengers + destinations[d];
6 up_costs_pass ← **true**;
7 added ← **true**;
8 origin_exists ← **true**;

The addition of the qualified potential passengers to the passengers on board is explained in the Algorithm 12. First of all, the destination index `dest_ind` is updated (line 1), since it will be need in the function `Update_Cost_Legs` to update the information about the costs for each leg. The number of people added is then subtracted to the available seats and added to the counter of the added people at the current location (lines 2,3). The costs will be then updated (line 4) with the algorithm that will be explained in the following subsection 3.4. Now the passengers can board the car (line 5). Finally, the boolean `up_cost_pass` will be set to **true** in order to trigger the update of the costs also for the other passengers already on board. Thus, the passengers of the current destination will be considered as added and the origin will be considered as already existing in the next cycles of the loop over the destinations since the costs of the legs have already be updated with it.

3.4 Cost update algorithms

Each time a new passenger or a group of passengers can be added to the car's passengers, the costs for the legs need to be recomputed. In order to do so, it is necessary to know whether the origin already appears in the keys of the map containing the costs for the legs, the index of the origin and of the destination in the list composed by all origins and destinations of the car, the difference these two indexes, and the number of passengers that are being added with the same origin and destination. This information is represented in the algorithm 14 respectively by the variables `origin_exists`, `or_ind`, `dest_ind`, `diff` and `added`. First of all the current `cost_legs` map will be copied to a temporary map (line 2) in order to create the new version of the costs. Then a loop will begin over the keys of the temporary map and at each cycle the variable i representing the index in the stops list will be incremented by one.

The operations inside the loop can be divided into:

lines 5-6 *copy of entries before the new origin.*
In this case, the entries do not need to be modified, and will therefore be just copied to the new `cost_legs`.

lines 7-17 *insertion of a new origin if not existing and update of the entries.*
In this situation, a total of four new legs may be necessary: one from the stop before the new

origin to the new origin, one from the new origin to the next stop, one from the stop before the new destination to it and one from the new destination to the next stop. Therefore, the insertion of the origin can be divided into two cases:

1. **the origin exists and i corresponds to the origin index**.
   In this case, the leg containing the origin as first key already exists, hence, the only operation needed is to update the value of the people on this leg by adding the number of new passengers to the current value. In addition, if the new destination is the next stop and it does not exist, the origin leg will have as second key the destination, and an additional leg from the destination to the next stop will also be created.
2. **the origin does not exists and i corresponds to the index of the stop before the origin**.
   In this case, the leg from the previous stop to the new origin will be created in any case, keeping the number of passenger of the existing leg. If the destination is right after the origin, a leg from the origin to the destination and one from the destination of the next stop will be created, otherwise only a leg from the new origin to the next stop will be created

lines 19-21 *copy and update of entries between the new origin and the new destination, when the latter is not the last destination.*
In this situation, the existing legs will have the number of passengers aboard updated according to the additional passengers.

lines 22-28 *insertion of the new destination, if not already present.*
If it has not been added with the origin, the leg from the stop before it to the destination and the leg from it to the next stop will be created with the related costs and passengers on board. Otherwise, if the destination already exists, the number of people on the leg from the stop before it to the destination will be updated with the addition of the number of new passengers on board.

lines 29-31 *copy of entries after the new destination.*
Once the index of the destination has been reached, the remaining entries will be copied to the updated `cost_legs`.

lines 32-34 *copy entries after the origin when then new destination is added in the last position.*
All the existing legs will be copied with the number of passengers aboard updated according to the number of additional passengers.

Therefore, at the end of the loop an updated `cost_legs` will be created. It will be now be possible to add the new entry for the new destination in the case in which it was added as last. The insertion is quite simple as it consists in computing the path between the second last stop, the previous last stop, and the last stop, the new last stop, contained in the stops (Algorithm 13, lines 2 to 4). Then the related cost is computed and all the information regarding this leg, namely the starting and ending point, the cost, the number of people on board (corresponding to the new added passengers), and the time needed to cover the leg, is added to the costs of the legs (lines 5, 6).

Algorithm 13: Addition of costs for last leg of path, when the destination of the passenger was added as the last stop

Data: cost_legs or_ind dest_ind stops origin_exists
Result: $last\_leg \in cost\_legs$

```
1 if last then
2     k[0] ← stops[|stops|-2];
3     k[1] ← stops[|stops|-1];
4     leg ← path(k[0], k[1]);
5     cost ← |leg|/1000*cost_km;
6     cost_legs ← cost_leg+ <[k[0], k[1]],[cost, added, time(leg)]>;
```

Algorithm 14: Decentralised algorithm for updating the costs after the addition of passengers to a car

```
Data: cost_legs or_ind dest_ind stops origin_exists added
Result: Updated cost_legs
i ← 0;
tmp ← cost_legs;
diff ← dest_in-or_index-1;
foreach k ∈ tmp.keys do
    if i < or_index-1 or (origin_exists and i < origin_index) then
        cost_legs ← cost_legs + <k,tmp[k]>;
    if origin_exists and i = or_ind then
        if diff == 0 and k[1] != stops[dest_ind] then
            add_dev(stops[or_ind, stops[dest_ind] stops[k[1]], tmp[k][1] + added) ;
        else if diff > 0 then
            cost_legs ← cost_legs + <k,[tmp[k][0], tmp[k][1]+added, tmp[k][2]]>;
    if !origin_exists and or_ind > 0 and i == (or_ind-1) then
        add_leg(stops[k[0]], stops[or_ind], tmp[k][1]);
        if diff == 0 and k[1] != stops[dest_ind] then
            add_dev(stops[or_ind, stops[dest_ind] stops[k[1]], tmp[k][1] + added) ;
        else if diff > 0 or (diff = 0 and key[1]= stops[dest_ind] ) then
            add_leg(stops[or_ind], stops[k[1]], tmp[k][1] + added);
    if !as_last then
        if (origin_exists and i > or_ind and i <dest_ind - 1 and diff > 0)
        or (!origin_exists and i≥ or_index and i < dest_ind-2 and or_ind > 0) then
            cost_legs ← cost_legs + <k,[tmp[k][0], tmp[k][1]+added, tmp[k][2]]>;
        if (origin_exists and i = dest_ind-1 and i ≥ or_ind and diff > 0)
        or (!origin_exists and i = dest_ind-2 and i≥ or_ind and or_ind > 0) then
            if k[1] == stops[dest_ind] then
                tmp[k][1] ← tmp[k][1] + added;
                cost_legs ← cost_legs + <k,tmp[k]>;
            else
                add_dev(stops[k[0]], stops[dest_ind] stops[k[1]], tmp[k][1] + added, tmp[k][1]) ;
        if (origin_exists and i ≥ dest_ind and diff = 0)
        or (!origin_exists and i ≥ dest_ind-1) then
            cost_legs ← cost_legs + <k,tmp[k]>;
    else
        if (i≥ or_ind and !origin_exists) or i>origin_index then
            cost_legs ← cost_legs + <k,[tmp[k][0], tmp[k][1]+added, tmp[k][2]]>;
    i ← i + 1;
```

In the algorithm 14 were introduced two functions, `add_leg` and `add_dev`. The first one is used to add a leg to the cost legs. It is necessary to know its starting and ending point and the people

on board. Thus, the path is computed along with the related costs and then a map entry, having as key the starting and ending points and as value a list containing the cost, the people on board and the time needed for the leg, is appended to the cost legs.

Algorithm 15: add_leg function

input : Starting point, p_1, and ending point, p_2, of additional leg and people on aboard on the leg, *on_board*

output: Append of additional leg to cost_legs

```
1 add_leg (p1, p2, On_board){
2     leg ← path(p1, p2);
3     cost ← |leg|/1000*cost_km;
4     cost_legs ← cost_leg + <[p1, p2 ],[cost, On_board, time(leg)]>;
5 }
```

The second one is used to add a deviation, namely a path composed by two legs having in common one point, the ending point of one and the starting point of the other, which will be called the *middle point*. This operation consists in applying the `add_leg` function two times, once for the first leg and once for the second leg composing the deviation. Therefore, it is necessary to know the origin, the middle point and the destination, and the number of passengers in each leg. If only one number for the passengers is given, it will be assumed that the two legs have the same number of passengers.

Algorithm 16: add_dev function

input : Origin, middle point and destination of the deviation, people on first leg, people on second leg

output: two legs that are added to the costs

```
1 add_dev (Origin, Middle, people1, people2){
2     add_leg (Origin, Middle,people1);
3     add_leg (Middle, Destination, people2);
4 }
```

After updating the costs for the legs of the path, it is possible to update the costs for each passenger. The procedure is the same as the one proposed in the algorithm 3 created for the centralised simulation, but it is applied over the list of passengers that are currently in the car, hence, those whose price has changed with the addition of new passengers.