Distributed Autonomous Shuttles

Antonio Bucchiarone, Annalisa Congiu

May 29, 2020

1 Introduction

In this approach the computational power and information are distributed among various nodes, the autonomous shuttles. This leads to the necessity of a more complex logic, yet, since the information that each shuttle will store and 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 report a distributed solution will be introduced with 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 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 shuttles. Specifically, in this version each autonomous shuttle 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 shuttle 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.

2.1 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 shuttle. 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 shuttle 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 shuttle 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 (Algorithm 1).

Algorithm 1: 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 shuttle 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 shuttle 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 2: Decentralised algorithm creation of an initial path after the selection of the first passenger or first group of passengers

Data: $P \leftarrow \{p_1, ..., p_n\}$ *(output of algorithm 1)*;
$targets \leftarrow origin \cup d$ of first group destinations.
Result: $P' \leftarrow p \in P$ that boarded the shuttle;
$targets \leftarrow origin \cup d$ of $p \in P'$;

```
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 ← destinations.keys - d ;
```

2.2 First path costs computation

Once the stops to be made by the shuttle to drop off the passenger are ordered, it is possible to compute the various legs of the shuttle'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 shuttle. 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 shuttle 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 3: Algorithm for the computation of the costs for the first group of passengers

Data: $P \leftarrow \{p_1, ..., p_n\}$ *(output of algorithm 1)*
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 ← |leg| ÷ 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]|;
```

2.3 Addition of passengers on the road

While the shuttle 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 shuttle 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 1). 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 shuttle has already visited. This will mean that the shuttle 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 shuttle, are removed from the initial list of potential new destinations (line 3) whereas the

related passengers are removed from the list of new potential passengers (lines 4 to 6).

Algorithm 4: Removal of passengers due to direction and previously visited stops

```
   Data: P ← {p_1, p_2, ..., p_n};
         destinations ← d of ∀p ∈ P;
         new_or;
   Result: P';
1  rem_angle;
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 ∈ destinations ∩ backwards do
6  |  |  destinations ← destinations - destinations[k];
```

If after this first filtering, there are still new passengers, the origin is added (Algorithm 5) to the to the list of stops of the shuttle 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 5: Insertion of the origin in the list of the stops

```
   Data:   stops;
   Result: new_or ∈ 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 shuttle among those that are on board is computed (Algorithm 6). 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 6: Computation of time for the path of the first added passengers that is at the time in the shuttle

```
   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;
```

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 shuttle or go alone to their destination (Algorithm 8). In order to do so, the remaining seats are computed and the loop is started. First of all, the last index of the shuttles *targets* is noted, (line 7 of Algorithm 8), then a check (Algorithm 7) 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 7, line 2). If there are not passengers left (Algorithm 7, lines 3,4) or if indeed there are no open seats left (lines 8,9 of Algorithm 8), 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 7: Check on remaining available seats in the shuttle 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 shuttle 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 shuttle is following will be recomputed to reach this new stop. Otherwise, once the shuttle has completed the insertion of the stops and all the new passengers that meet the condition have boarded the shuttle, 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 shuttle 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 8: Decentralised algorithm for adding passengers after the shuttle 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                         change ← t_a(original) - t_a(t2n) + time_a(t2d) + t_a(d2n);
26                         check ← (change<(t_a(original)×3/2)) and (av_time≥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 not 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 9: Addition of new passengers

Data: *destinations*;
Result: $P' \leftarrow P + P_{new}$;
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 9. 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 2.4. Now the passengers can board the shuttle (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.

2.4 Cost update algorithms

Each time a new passenger or a group of passengers can be added to the shuttle'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 shuttle, 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 11 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 10, 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 10: 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 11: Decentralised algorithm for updating the costs after the addition of passengers to a shuttle

```
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 not 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 not as_last then
        if (origin_exists and i > or_ind and i < dest_ind - 1 and diff > 0)
        or (not 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 (not 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 (not origin_exists and i ≥ dest_ind-1) then
            cost_legs ← cost_legs + ⟨k,tmp[k]⟩;
    else
        if (i ≥ or_ind and not 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 11 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 12: add_leg function

input : p_1; p_2; on_board;
output: $cost\_legs \leftarrow leg_{new}$;

```
1 add_leg (p1, p2, On_board){
2     leg ← path(p1, p2);
3     cost ← |leg| ÷ 1000 × cost_km;
4     cost_legs ← cost_leg + langle[p1, p2 ],[cost, On_board, time(leg)]rangle;
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 13: add_dev function

input : Origin; Middle; Destination; On_board_1; On_board_2;
output: $cost\_legs \leftarrow cost\_legs + [leg_{new1}, leg_{new2}]$;

```
1 add_dev (Origin, Middle, On_board1, On_board2){
2     add_leg (Origin, Middle,On_board1);
3     add_leg (Middle, Destination, On_board2);
4 }
```

After updating the costs for the legs of the path, it is possible to update the costs for each passenger. The update is applied over the list of passengers that are currently in the shuttle, hence, those whose price has changed with the addition of new passengers.