It is unrealistic that the autonomous car knows the trajectories of all the cars in advance. To improve the situation, I implemented online version of the previous algorithm. The online version uses heuristics to predict the best possible path from current position of the autonomous car. The algorithm is similar to the previous algorithm in sense that it will still sample points in RRT style and find a path, but that path will just be used as a heuristic. The car will take one step according to the heuristics, reach a new state, sample again from new position and follow the heuristics path.

Let us look at the algorithm now then I will explain each design choice.

input: $(initial_{state}, final_{pos}, N)$ where $initial_{pos}$ is initial position, $final_{pos}$ is target position and N is the number of samples
output: $\begin{cases} (G, \quad controls) \\ \quad None \end{cases}$,
where graph, G, contains multiple paths from initial to final position
$control$ is the corresponding controls command for each transition from state s to s'.

```
$create\_roadmap(initial_{pos}, final_{pos}, N)$
$\Rightarrow$ $backup\_plan = get\_default\_policy()$
$\Rightarrow$ $m_g \leftarrow (initial_{pos}, 0,0)$
$\Rightarrow$ $roadmap \leftarrow m_g$
$\Rightarrow$ $repeat\ till\ final_{pos}\ is\ reached:$
$\Rightarrow$     $plan \leftarrow create\_heuristics(m_g, final_{pos}, N)$
$\Rightarrow$     $if\ plan\ is\ valid:$
$\Rightarrow$         $update\ backup\_plan$
$\Rightarrow$         $m_{new} \leftarrow next\ node\ in\ plan$
$\Rightarrow$         $add\ to\ roadmap(m_{new})$
$\Rightarrow$         $add\ to\ action\ (action\ corresponding\ to\ (m_g, m_{new})$
$\Rightarrow$     $else:$
$\Rightarrow$         $m_{new} \leftarrow next\ node\ in\ backup\ plan$
$\Rightarrow$         $add\ to\ roadmap(m_{new})$
$\Rightarrow$         $add\ to\ action\ (action\ corresponding\ to\ (m_g, m_{new})$
$\Rightarrow$ $return\ (roadmap, action)$
```

```
$create\_heuristics(initial_{state}, final_{pos}, N):$
$\Rightarrow$ $G \leftarrow initial_{state}$
$\Rightarrow$ $T \leftarrow \{\}$
$\Rightarrow$ $predicted_{model} = add\ actual\ car's\ position\ till\ timestep\ specified\ by\ initial_{state}$
$\Rightarrow$ $repeat\ for\ N\ steps:$
$\Rightarrow$     $m_g \leftarrow select\_node(G)$                 # select an existing node from the graph
$\Rightarrow$     $m_{new}, u \leftarrow sample\_new\_milestone(m_g)$   # sample new milestone using $m_g$
$\Rightarrow$     $satisfy\_constraints(m_{new})$            # check if $m_{new}$ satisfy dynamic constraints
$\Rightarrow$     $check\_collision(m_g, m_{new})$           # check if $m_g, m_{new}$ is collision free
$\Rightarrow$     $if\ check\_collision\ is\ successful:$
$\Rightarrow$         $add\ m_{new}\ to\ G\ with\ weight\ 1$
$\Rightarrow$         $if\ m_{new}\ lies\ in\ final_{pos}\ region:$
$\Rightarrow$             $add\ m_{new}\ to\ T$
$\Rightarrow$             $T_{max} \leftarrow m_{new}[t]$            # reduce the scope of expansion
$\Rightarrow$ $return\ G$
```

```
$select\_node(G):$
$\Rightarrow$ $s \leftarrow random\ value\ between\ [0, final_{pos})$   # get a random position s
$\Rightarrow$ $t \leftarrow random\ value\ between\ [0, T_{max})$     # get a random timestep in the given scope
```

```
⇒ $node \leftarrow (s', v', t')$ from $G$ such that $distance((s,t),(s',t'))$ is minimum
⇒ return node

sample_new_milestone($m_g$):
⇒ $\Delta t \leftarrow 0.1$                                    # as $f$ is given $10$, so $\frac{1}{f} = 0.1$
⇒ $(s_t, v_t, t) \leftarrow m_g$
⇒ $u \leftarrow$ random value of control (acceleration)        # random acceleration with bias
⇒ $v_{t'} \leftarrow v_t + a\Delta t$
⇒ $s_{t'} \leftarrow s_t + v_t\Delta t + \frac{a\Delta t^2}{2}$
⇒ $t' \leftarrow t + \Delta t$
⇒ return $(s_{t'}, v_{t'}, t')$

check_collision($m_g, m_{new}$):
⇒ $pos_t \leftarrow$ get robots' position at timestep $t$ using predicted model
⇒ $pos_{t'} \leftarrow$ get robots' position at timestep $t'$ using predicted model
⇒ if $pos_{t'}$ is invalid:
⇒     predict car's position at $t'$ and add to predicted model
⇒ for each robot:
⇒     check if $m_{new}$ is free horizontally (by expanding size of robots and including their velocity)
⇒     check if $m_{new}$ is free vertically (by expanding size of robots and including our velocity(safe distance)
⇒     if $m_{new}$ is not free horizontally and vertically:
⇒         return False
⇒ return True
```

The algorithm starts by picking a default policy. From the initial state, this online version calls create heuristics function, which creates a rough estimate of the positions of the other car and make a rough roadmap. At t=0, autonomous car only knows the starting positions of other cars. It has no idea of their speed or acceleration. It just picks random velocity and acceleration for the obstacle cars and creates a mapping of obstacles. It then finds a solution according to this mapping and returns the path according to heuristics. The planner takes an action according to the heuristics and reach a new state s'. Now, the planner again repeats the same process, creates a heuristics roadmap and selects the best possible action from the heuristics. This goes on until final position is reached.

Design choices:

There are few design choices made by me which can be altered to change the performance of the planner.

⇒ **Default policy**: I used constant acceleration of 0 as the default policy. It is a very simple policy because at t=0, autonomous car does not know much about the environment. So, the best thing is to wait and observe the environment if you could not find a good heuristic.

⇒ **Number of Samples**: The accuracy of the algorithm defined above depends on the number of samples drawn at each step. As the samples increases, algorithm finds a heuristic path with high success. But, computation time is also directly proportional to the number of samples. So, as the samples increases, planner takes more time to take a step. So, it is trade-off decision that one has to take. I sampled 15000 points for first 5 iteration as I believe that a good initial heuristic map is important to get the car into right position from the start. I reduced the subsequent samples to 1000 per step to speed up the computation. However, as we can notice, if *create_heuristic* in current timestep could not find a good heuristic, the planner will pick action from previous heuristics. If planner encounters more None in line, it will keep on picking actions from the last successful heuristic. It will result in planner picking very old value for action, however the environment might have changed. To mitigate this, we can increase the number of samples for each step. It will increase the likelihood of planner finding a good heuristic for current timestep according to current environment.

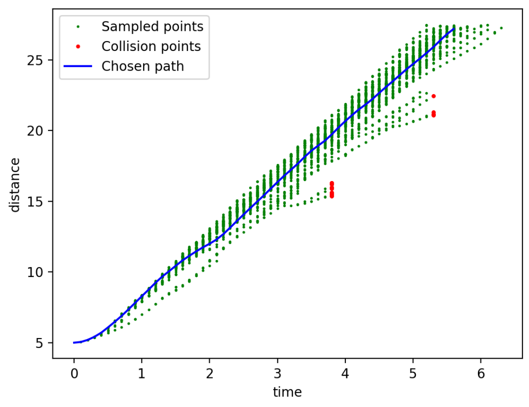
⇒ **Probabilistic model**: The performance of planner depends on heuristics which in turn depends on how accurate were the predictions of obstacle car position's probability model. I assumed that car's next position depends more on its last 3 positions. So, I used last 3 positions (discarding older positions) of the obstacle cars to find the acceleration and velocity of cars at current timestep. I then used it to predict the position of car at next timestep. One more point to note, I used the predicted positions at $t+1$ to predict its position at $t+2, t+3$ *and so on*. This helped to create an approximate roadmap till target. One might improve the action chosen at current timestep by improving the probability model (like taking more observations into account). But for the given problems, this was a reasonably reliable assumption.

⇒ **Biasing towards goal**: Value for the control input was not uniformly selected. It was biased to make the car choose more actions which will take it to the goal faster (like choosing values closer to max acceleration allowed). This particularly helped in converging to the best path quickly. But there was a negative impact of doing this. If a high acceleration value resulted in an invalid state, it will have more tendency to choose that high value again resulting in a collision again. This was a trade-off that I made which resulted fine in the current problem set.

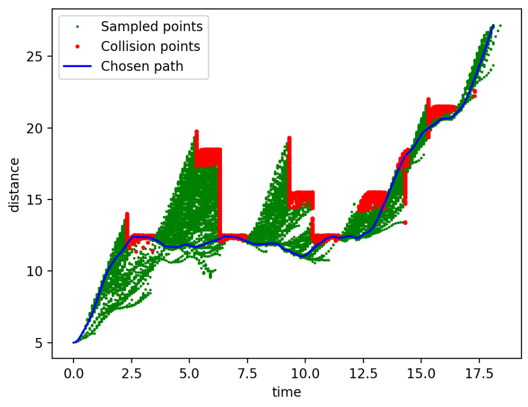
⇒ **What to do when goal state is reached in heuristic search:** Generally, the algorithm is given a maximum value of number of samples to be made. If the algorithm finds a suitable path, it stops searching anymore and returns the heuristic path. This resulted in lesser running time of the algorithm as it was able to find a possible path (may not be the best) in very less iterations. We could set the algorithm to keep running and find a better path (like I did in offline version). This will result in in better heuristic but will require more running time. In my experimentation, this design choice worked fine and resulted in near optimal results most of the time.

Following are the plots for distance-time plots for the problems in the problem set. You may observe that these plots are densely sampled due to a lot of cumulative sample points compared to previous algorithm.

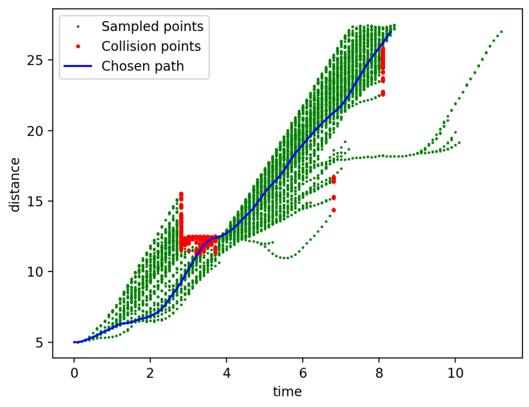
Data 1



Data 2

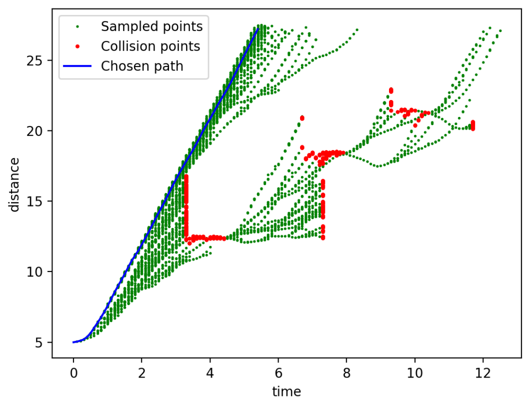


Data 3

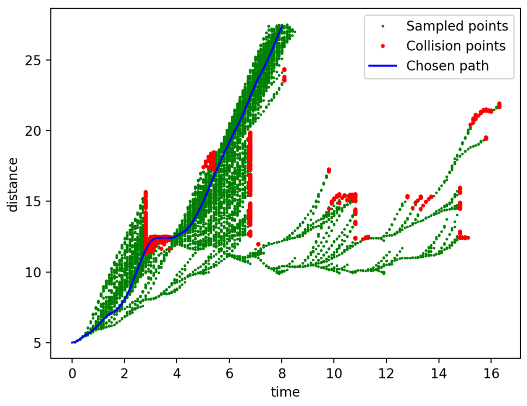


Test 1

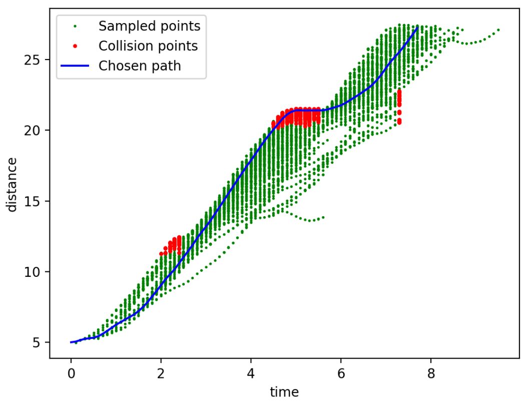
Data 4



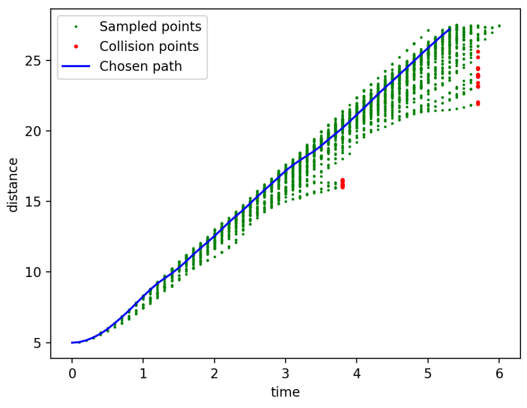
Test 2

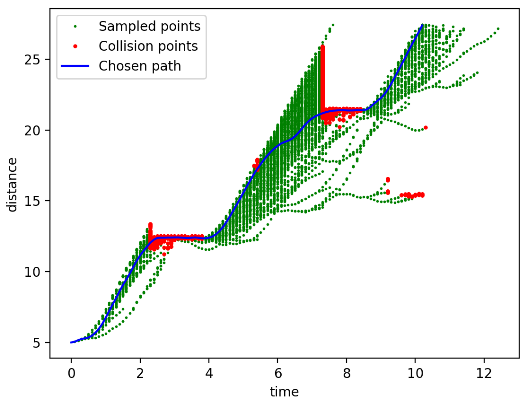

Sampled points
Collision points
Chosen path
distance
time

Test 3

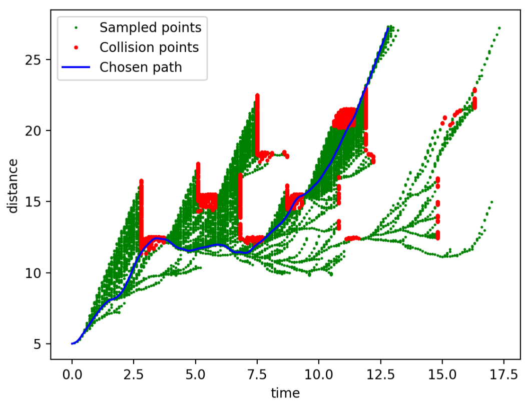

Sampled points
Collision points
Chosen path
distance
time

Test 4

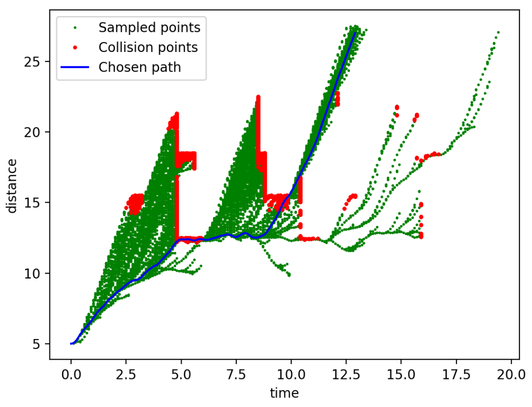

Sampled points
Collision points
Chosen path
distance
time


Sampled points
Collision points
Chosen path
distance
time

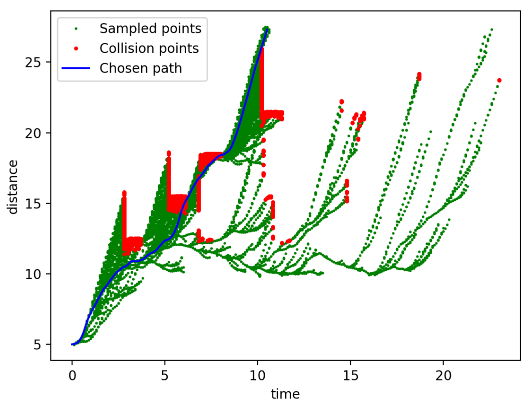
Test 5



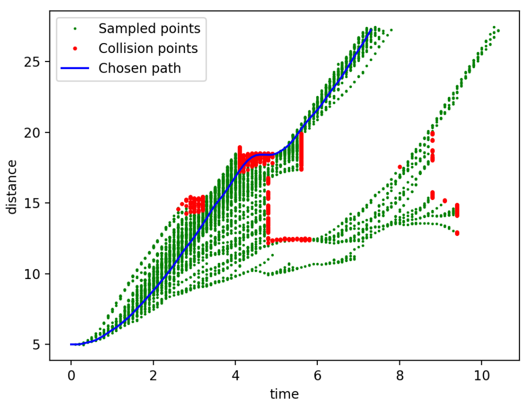
Test 6



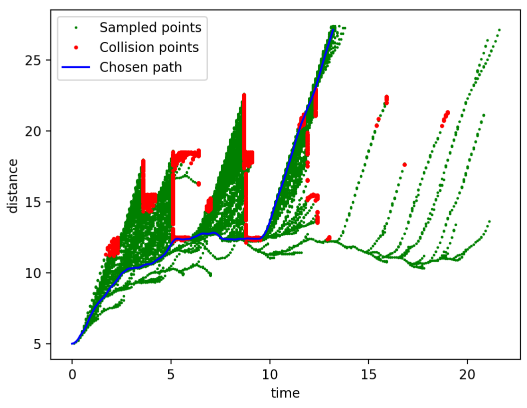
Test 7



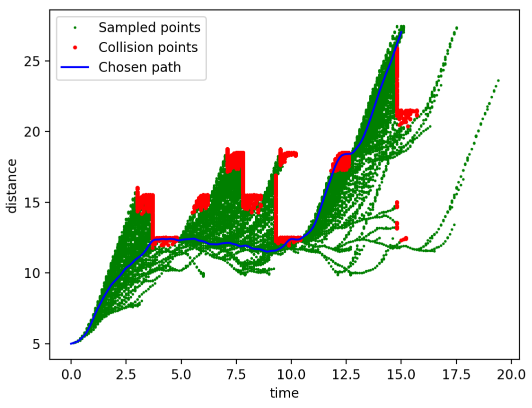
Test 8



Test 9



Test 10


Sampled points
Collision points
Chosen path
distance
time