In [ ]:
```
# EE183DA - LAB 4
# Group: Tiger
```

In [27]:
```
import numpy as np
import math
import matplotlib.pyplot as plt
import sys
import time
from numpy import cos, sin

class Node():
    # Define Node structure
    def __init__(self, state):
        self.state = state  # state include x,y,h
        self.parent = None  # the edges connected to this node

car_collision_d = 130
Lx = 600
Ly = 450
car_width = 85
car_length = 100

obstaclelist=[
    (0, 300, 200, 150),
    (400, 300, 200, 150)]
```

In [2]:
```
# 2.2a
def draw_Car(pos):
        rec_corner = [pos[0]+80*cos(pos[2]+np.pi)+(car_width/2)*cos(pos[2]-np.pi/2),
                      pos[1]+80*sin(pos[2]+np.pi)+(car_width/2)*sin(pos[2]-np.pi/2)]
        robot = plt.Rectangle(rec_corner, car_length, car_width, np.degrees(pos[2]), facecolor='w', edgecolor
='b')
        ax.add_patch(robot)
        plt.arrow(pos[0], pos[1], np.cos(pos[2]), np.sin(pos[2]), color='r', width=10)

def draw_Ospace(obstaclelist):
    fig = plt.figure(figsize = (9, 6))
    ax = fig.add_subplot(1,1,1)
    plt.xlim((0, Lx))
    plt.ylim((0, Ly))
    plt.grid()
    # Plot the obstacles
    for ob in obstaclelist:
        obstacle = plt.Rectangle(ob[0:2], ob[2], ob[3], color = 'k')
        ax.add_patch(obstacle)
    return ax




# Draw map
initial_state = [100, 100, 0]
goal_state = [300, 400, np.pi/2] #heading parking
ax = draw_Ospace(obstaclelist)
draw_Car(initial_state)
draw_Car(goal_state)
```

In [3]:

```
# 2.2b Given a set of points V in C-space and a single other target point, write and test a function to deter
mine
# which of the points in V is closest to the target.
def closest_node(node_list, target_point):
    # store distances from target point to all nodes
    d_list = []
    for i, node in enumerate(node_list):
        d_list.append([np.linalg.norm(target_point - node.state, ord = 2),i])
    # find the index of the smallest distance
    i_min = min(d_list)[1]
    return node_list[i_min], i_min
```

In [4]:

```python
# Testing, given a set V of 4 nodes
n1 = Node(np.array([0, 0, -2.0])) #0
n2 = Node(np.array([0, 4, -3.0])) #1
n3 = Node(np.array([4, 0, -2.0])) #2
n4 = Node(np.array([4, 4, -3.0])) #3
nodelist = [n1, n2, n3, n4]
Closestnode, index = closest_node(nodelist, [5, 1, -2.1])
print('Closest node of ', [5, 1, -2.1], ' is ', Closestnode.state,index)
Closestnode, index = closest_node(nodelist, [3, 2, -2.1])
print('Closest node of ', [3, 2, -2.1], ' is ', Closestnode.state,index)
Closestnode, index = closest_node(nodelist, [4, 3, -2.1])
print('Closest node of ', [4, 3, -2.1], ' is ', Closestnode.state,index)
```

```
Closest node of  [5, 1, -2.1]  is  [ 4.  0. -2.] 2
Closest node of  [3, 2, -2.1]  is  [ 4.  0. -2.] 2
Closest node of  [4, 3, -2.1]  is  [ 4.  4. -3.] 3
```

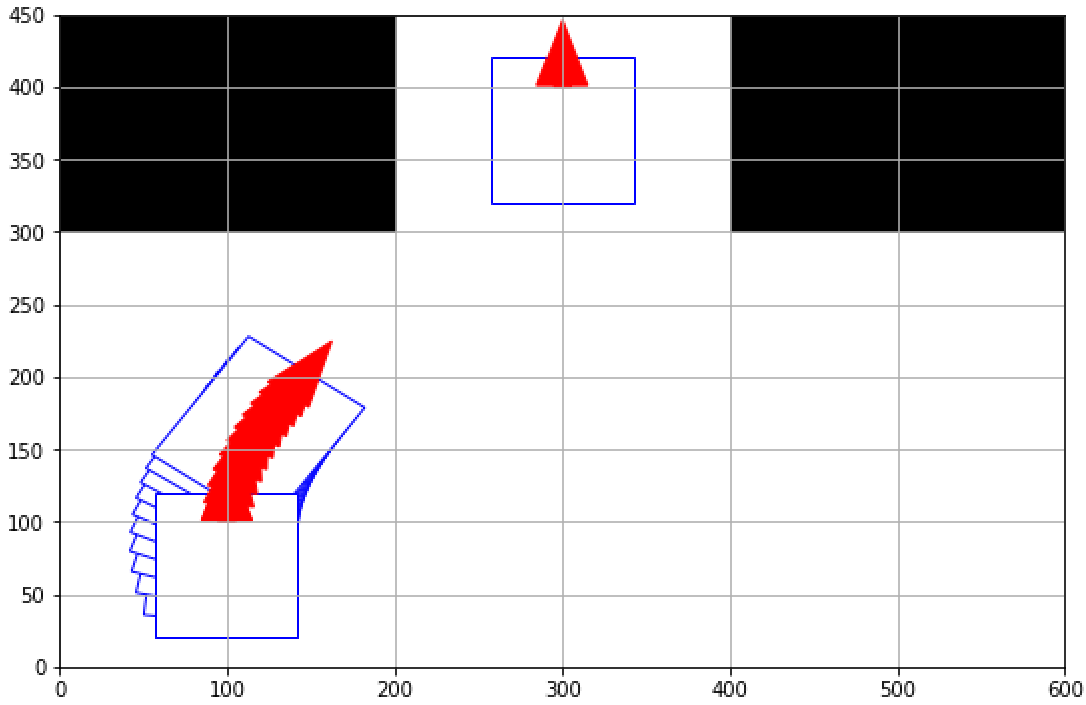
In [5]:
```
# 2.2c Given arbitrary initial and target robot states (in C-space), write and test a function to generate a
 smooth
# achievable trajectory from the inital state towards the target lasting 1 second.
# What are the control inputs for this trajectory?

# simulation parameters
Kp_d = 0.3
Kp_alpha = 1.5
Kp_beta = -0.3
dt = 0.1
# robot parameters
r = 20.0 #mm
w = 85.0 #mm

# given initial and target state of robot in C-space
# return the smooth trajectory and the control inputs in 1 second

def trajectory_generation(initial_state, target_state, time_limted=1):

    # Initilization
    curr_state = [initial_state[0],initial_state[1],initial_state[2]]

    traj = []
    input_u = []
    d2target = np.linalg.norm(target_state - initial_state, ord = 2)
    t = 0

    while (t < time_limted or time_limted == 0) and (d2target > 0.01):
        traj.append(np.array(curr_state))
        # distance from current state to the goal state
        d2target = np.linalg.norm(target_state - curr_state, ord = 2)
        # angle to the goal relative to the heading of the robot
        alpha = (np.arctan2(target_state[1]-curr_state[1], target_state[0]-curr_state[0]) -
                 curr_state[2] + np.pi) % (2 * np.pi) - np.pi
        # goal direction + different of car heading and goal direction
        beta = (target_state[2] - curr_state[2] - alpha + np.pi) % (2 * np.pi) - np.pi
        # velocity of robot, define by Kp_d constant
        v = Kp_d * d2target
        omega = Kp_alpha * alpha + Kp_beta * beta

        # Calculate the angular speed of 2 wheels
        omega_right = (2*v + omega*w)/(2*r)
```

```
        omega_left = (2*v - omega*w)/(2*r)
        input_u.append(np.array([omega_left, omega_right]))

        if alpha > np.pi / 2 or alpha < -np.pi / 2:
            v = -v

        curr_state[2] = curr_state[2] + omega * dt
        curr_state[0] = curr_state[0] + v * np.cos(curr_state[2]) * dt
        curr_state[1] = curr_state[1] + v * np.sin(curr_state[2]) * dt
        if (time_limted != 0):
            t += dt
        else:
            t = 0


    return traj, input_u
```

In [21]:
```
# Try to go from start to goal in 1 second
initial_state = np.array([100, 100, np.pi/2])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
ax = draw_Ospace(obstaclelist)

traj, u = trajectory_generation(initial_state, goal_state)

for node in traj:
    draw_Car(node)


draw_Car(initial_state)
draw_Car(goal_state)
print("The input control is (change after dt time) [left speed, right speed]:")
u
```

```
The input control is (change after dt time) [left speed, right speed]:
```

Out[21]:
```
[array([7.65743687, 3.15921695]),
 array([7.23151403, 3.29871052]),
 array([6.83806435, 3.40242866]),
 array([6.47443829, 3.47645325]),
 array([6.13816293, 3.52574584]),
 array([5.82692748, 3.55438079]),
 array([5.53857814, 3.56572667]),
 array([5.27111583, 3.56258706]),
 array([5.02269386, 3.54731007]),
 array([4.79161429, 3.52187374]),
 array([4.57632276, 3.48795306])]
```

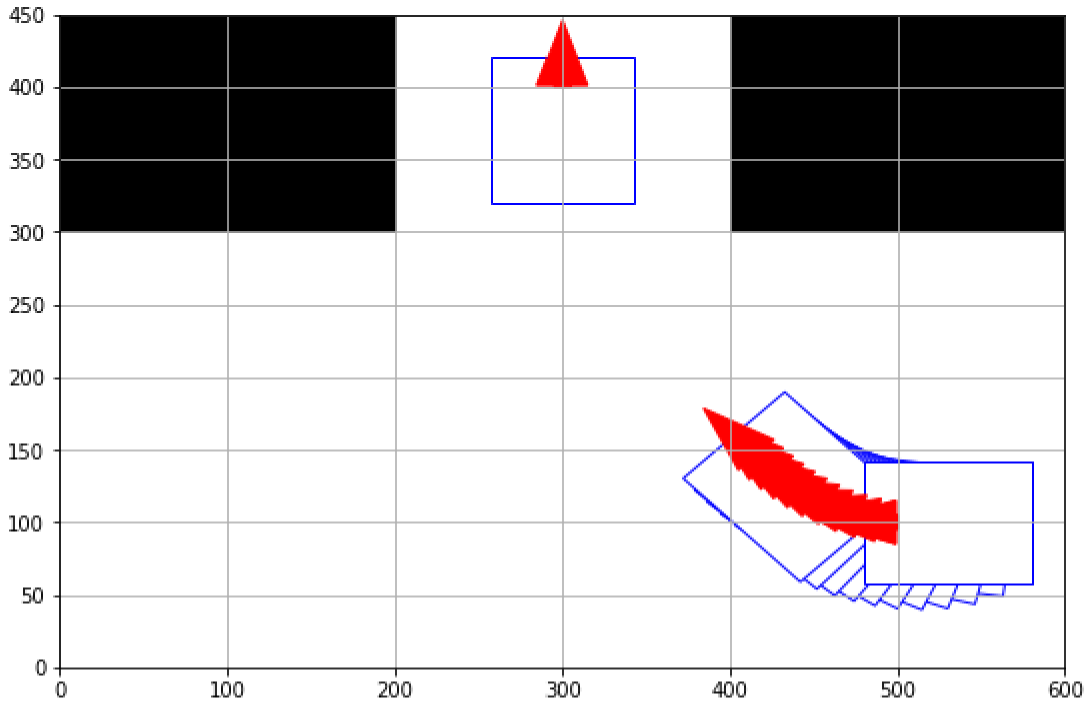


In [23]:
```
# # Try to go from start to goal in 1s
initial_state = np.array([500, 100, np.pi])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
ax = draw_Ospace(obstaclelist)

traj, u = trajectory_generation(initial_state, goal_state)
for node in traj:
    draw_Car(node)


draw_Car(initial_state)
draw_Car(goal_state)
print("The input control is (change after dt time) [left speed, right speed]:")
u
```

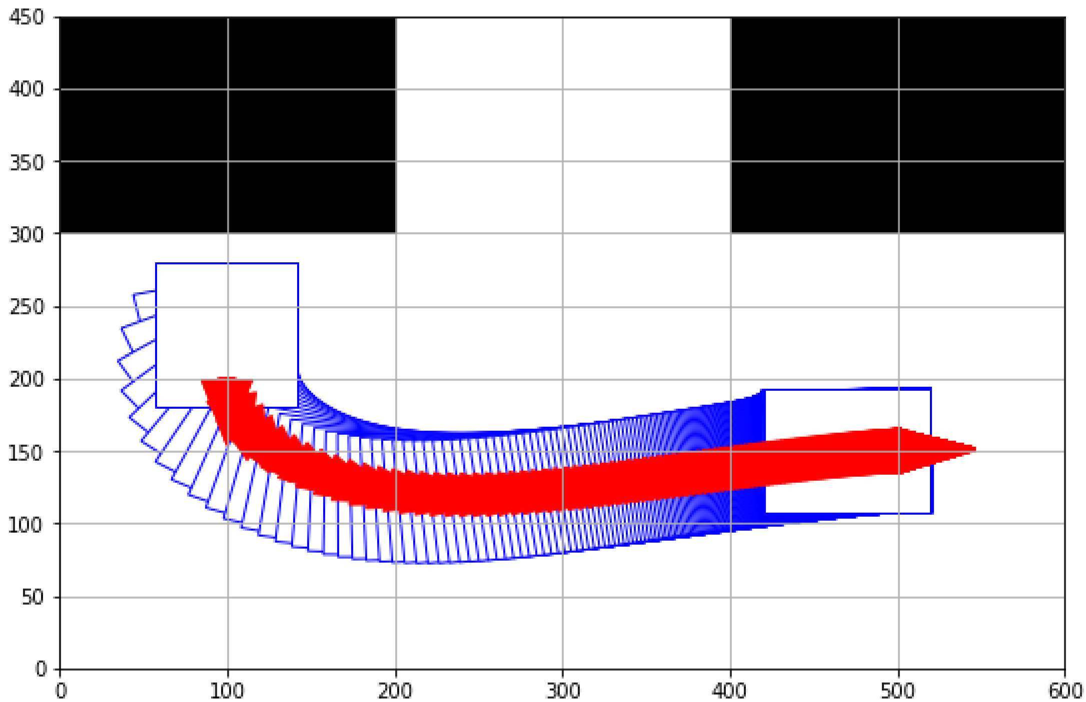
```
The input control is (change after dt time) [left speed, right speed]:
```

Out[23]:

```
[array([8.16618157, 2.6505749 ]),
 array([7.73535964, 2.87076907]),
 array([7.33746424, 3.04069623]),
 array([6.96928502, 3.16899629]),
 array([6.62806084, 3.26273771]),
 array([6.31135968, 3.32772634]),
 array([6.0170058 , 3.36875228]),
 array([5.74303459, 3.38978605]),
 array([5.4876632 , 3.39413432]),
 array([5.24927028, 3.38456338]),
 array([5.02638065, 3.36339732])]
```



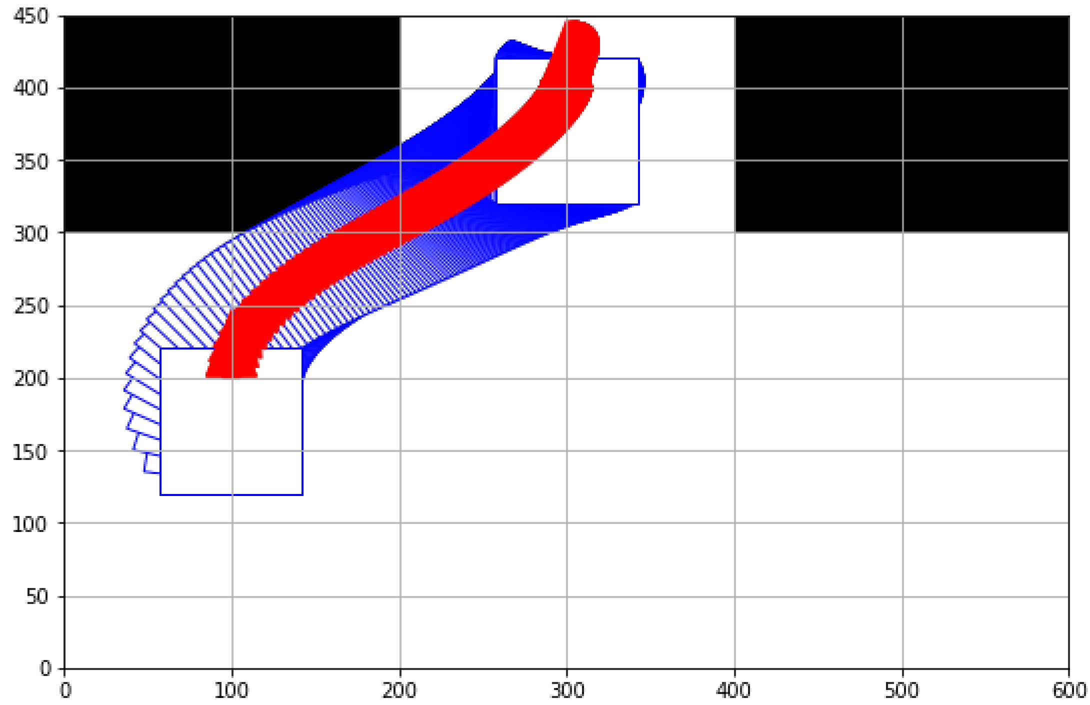
In [25]:
```
# Try to go from start to goal without limit 1 second
# Choose start point and goal which can go to by just one smooth path.
initial_state = np.array([100, 200, -np.pi/2])
goal_state = np.array([500, 150,0]) #heading parking
ax = draw_Ospace(obstaclelist)

traj, u = trajectory_generation(initial_state, goal_state, 0)
for node in traj:
    draw_Car(node)


draw_Car(initial_state)
draw_Car(goal_state)
```



In [9]:
```
# 2.2d Given your C-space map and an arbitrary robot trajectory, write and test a function to determine wheth
er
# this trajectory is collision free.
def collision(traj, obstacle_list):
    collision = 0
    for node in traj:
        robot_center = node

        for (ox, oy, wx, wy) in obstacle_list:
            rectangle_center = [ox + wx / 2, oy + wy / 2]
            v = [abs(robot_center[0] - rectangle_center[0]), abs(robot_center[1] - rectangle_center[1])]
            h = [wx / 2, wy / 2]
            u = [v[0] - h[0], v[1] - h[1]]
            if u[0] < 0:
                u[0] = 0
            if u[1] < 0:
                u[1] = 0
            if u[0] ** 2 + u[1] ** 2 - (car_collision_d / 2) ** 2 < 0:
                collision = 1
                break

    return collision
```

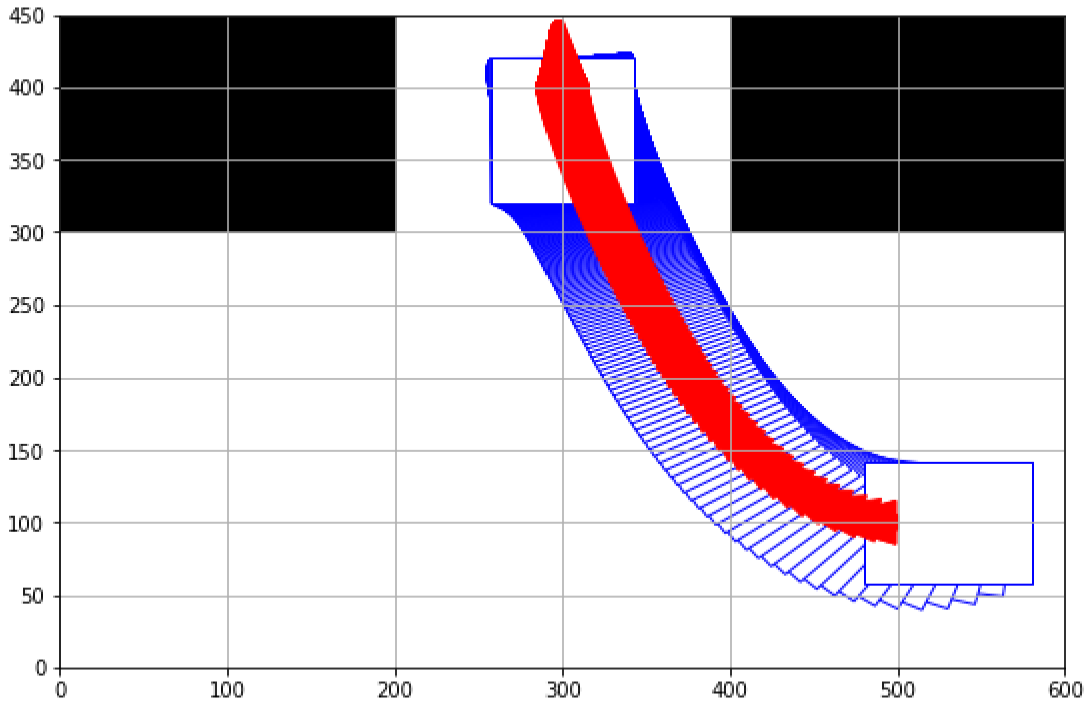
In [10]:
```
# Create the trajectory from start to end, then check collision free for this traj
initial_state = np.array([100, 200, np.pi/2])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
ax = draw_Ospace(obstaclelist)

traj, input_traj = trajectory_generation(initial_state, goal_state, 0)
for node in traj:
    draw_Car(node)


draw_Car(initial_state)
draw_Car(goal_state)

if collision(traj,obstaclelist) == 0:
    print('The trajectory is free collision')
else:
    print('The trajectory is not free collision')
```

```
The trajectory is not free collision
```



In [11]:
```
# Create the trajectory from start to end, then check collision free for this traj
initial_state = np.array([500, 100, np.pi])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
ax = draw_Ospace(obstaclelist)

traj, input_traj = trajectory_generation(initial_state, goal_state, 0)
for node in traj:
    draw_Car(node)


draw_Car(initial_state)
draw_Car(goal_state)

if collision(traj,obstaclelist) == 0:
    print('The trajectory is free collision')
else:
    print('The trajectory is not free collision')
```

```
The trajectory is free collision
```

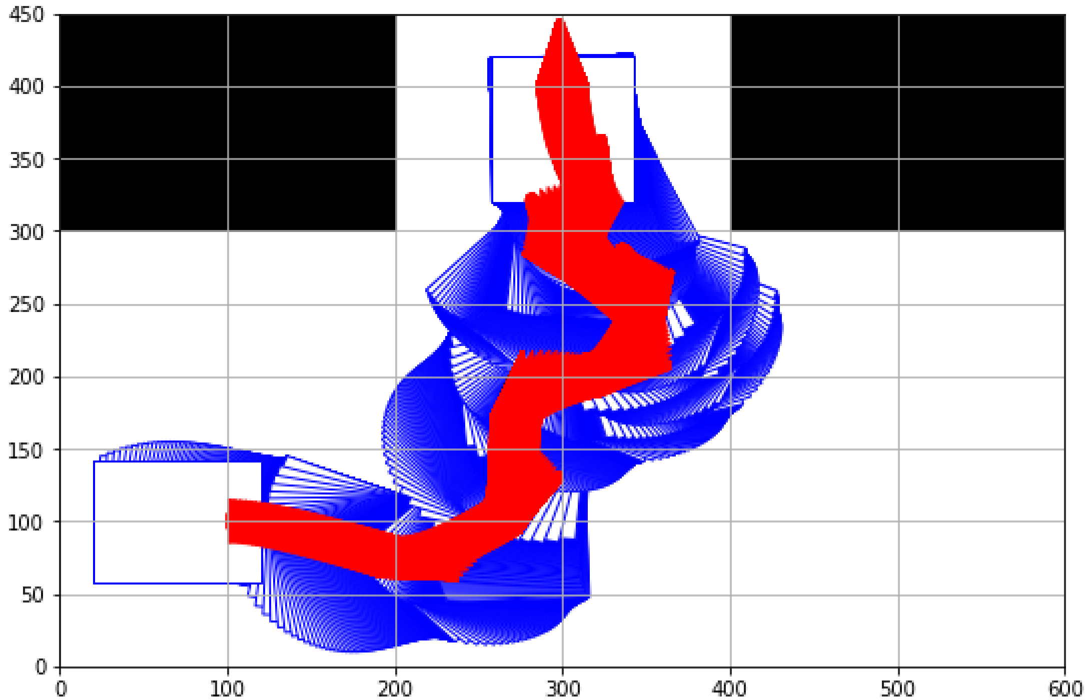


In [12]:
```
# 2.2e Put these functions together to implement an RRT planner on your map to generate a trajectory from a
# specified initial state to the desired goal state. Visualize the evolution of the RRT as well as the result
ing
# trajectory

import random

def rrt(initial_state, goal_state, boundary, max_step=2000):
    start_node = Node(initial_state)
    end_node = Node(goal_state)
    nodelist = [start_node]

    np.random.seed(10)
    done = False
    for i in range(max_step):
        # Distance at the lastest node to goal
        d2goal = np.linalg.norm(goal_state - nodelist[-1].state.copy(), ord = 2)
        # If the goal is inside the circle around car means reach the goal, stop expand the tree
        if d2goal < car_collision_d/2:
            done = True
            break

        # Take a random point in C_space (the point is collision free)
        pnt_random = np.zeros(3)
        while True:
            for i in range(3):
                pnt_random[i] = np.random.uniform(sample_boundary[i][0], sample_boundary[i][1])
            tmp = []
            tmp.append(np.array(pnt_random))
            if (collision(tmp,obstaclelist) == 0):
                break;

         # Find nearest node in the nodelist to the random point
        nearestNode, min_index = closest_node(nodelist, pnt_random)
        # make a trajectory from this nearest point to the random point
        traj_temp, _ = trajectory_generation(nearestNode.state.copy(), pnt_random, time_limted=1)

        # Check this trajectory for collision free, add this node to the nodelist if collision free
        if collision(traj_temp, obstaclelist)==0:
            newNode = Node(traj_temp[-1])
            newNode.parent = min_index
            nodelist.append(newNode)
```

```
        # print("Number of node in Nodelist:", len(nodelist))

    if (done==False):
        print ("Cannot find the path to the goal.")
        return [],[]
    else:
        # Find the path from start point to goal
        path = [goal_state]
        i = len(nodelist) - 1
        while nodelist[i].parent is not None:
            node = nodelist[i]
            path.append(node.state)
            i = node.parent
        path.append(initial_state)

        # Generate the traj from start to end point
        traj = []
        i = len(path) - 1
        while i >= 1:
            tmp, _ = trajectory_generation(path[i], path[i-1], time_limted=0)
            traj = traj + tmp
            i = i - 1
        return traj, path
```

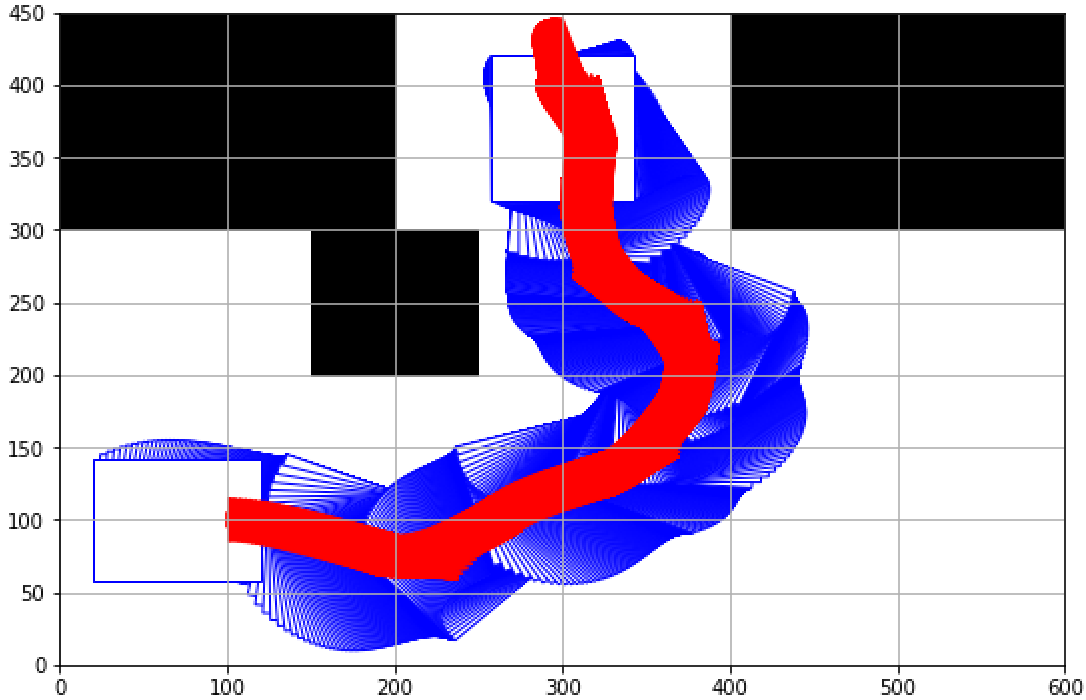
In [29]:
```
initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
```

```
Time to calculate for the path: 0.3262195587158203
```



In [30]:
```
# Add more obstacle on the map
obstaclelist=[
    (0, 300, 200, 150),
    (150, 200, 100, 100),
    (400, 300, 200, 150)]

initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
```

```
Time to calculate for the path: 0.5026412010192871
```



In [31]:
```
# Add more obstacle on the map so that there are no path to the goal
obstaclelist=[
    (0, 300, 200, 150),
    (250, 200, 100, 100),
    (400, 300, 200, 150)]

initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
```

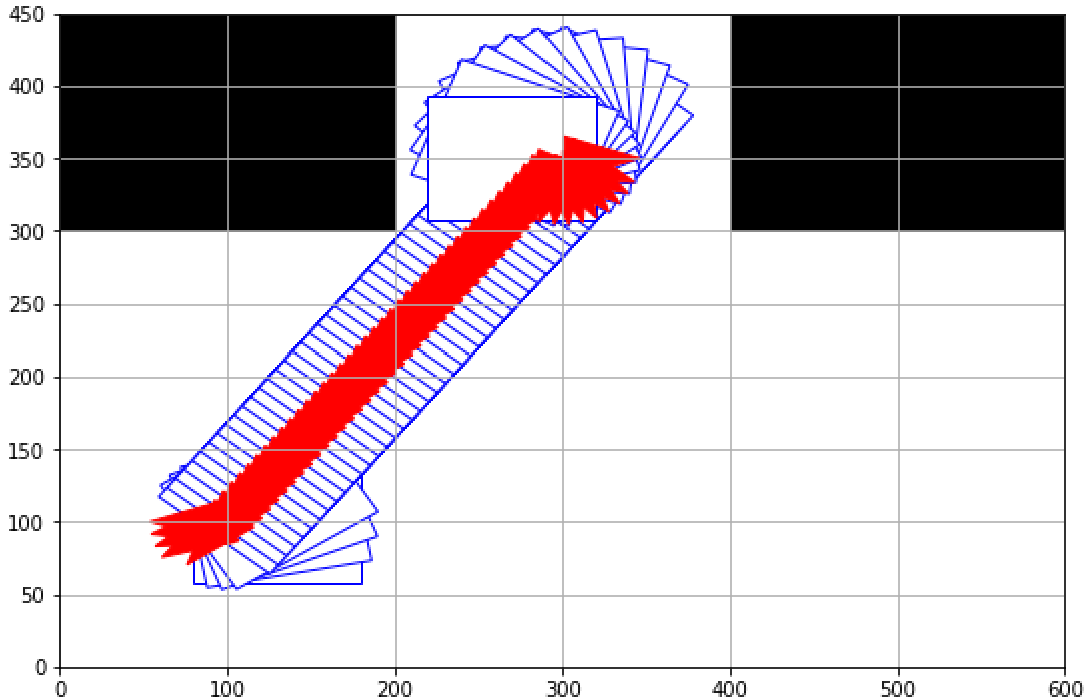
```
Cannot find the path to the goal.
Time to calculate for the path: 21.899013996124268
```

In [16]:
```
# 2.2f The car cannot control the speed of 2 wheels at a variable speed, only stable a max speed
# write a new trajectory generation that replace a curve trajectory by a turn, go straight and then turn to g
oal heading angle
def line_trajectory(initial_state, target_state, time_limted=1):

    # Initilization
    curr_state = [initial_state[0],initial_state[1],initial_state[2]]
    v= 80    # max speed of the car mm/s
    traj = []
    input_u = []
    d2target = np.linalg.norm(target_state - initial_state, ord = 2)
    t = 0
    StartTurn = True
    GoFW = False
    GoBW = False
    EndTurn = False
    alpha = (np.arctan2(target_state[1]-curr_state[1], target_state[0]-curr_state[0]) -
             curr_state[2] + np.pi) % (2 * np.pi) - np.pi
    while (t < time_limted or time_limted == 0) and (d2target > 0.01):
        traj.append(np.array(curr_state))

        # Turning speed
        omega = 2*v/car_width

        # turning to the angle of start to end point

        if StartTurn:
            if alpha > np.pi / 2 or alpha < -np.pi / 2:
                if (alpha > 0):
                    curr_state[2] = curr_state[2] - omega * dt
                    input_u.append(np.array([-v, v]))
                else:
                    curr_state[2] = curr_state[2] + omega * dt
                    input_u.append(np.array([v, -v]))
            else:
                if (alpha > 0):
                    curr_state[2] = curr_state[2] + omega * dt
                    input_u.append(np.array([v, -v]))
                else:
                    curr_state[2] = curr_state[2] - omega * dt
                    input_u.append(np.array([-v, v]))
```

```
        # angle to the goal relative to the heading of the robot
        alpha = (np.arctan2(target_state[1]-curr_state[1], target_state[0]-curr_state[0]) -
                 curr_state[2] + np.pi) % (2 * np.pi) - np.pi

        if StartTurn and np.abs(alpha % np.pi) < omega*dt:
            if np.abs(alpha) < np.pi/2:
                curr_state[2] = np.arctan2(target_state[1]-curr_state[1], target_state[0]-curr_state[0])
                GoFW = True
            else:
                curr_state[2] = np.arctan2(target_state[1]-curr_state[1], target_state[0]-curr_state[0]) + np
.pi
                GoBW = True
            StartTurn = False

        # Go forward
        if GoFW:
            curr_state[0] = curr_state[0] + v * np.cos(curr_state[2]) * dt
            curr_state[1] = curr_state[1] + v * np.sin(curr_state[2]) * dt
            input_u.append(np.array([v, v]))
            d2target = np.linalg.norm(target_state - curr_state, ord = 2)
            if d2target < v*dt:
                curr_state[0] = target_state[0]
                curr_state[1] = target_state[1]
                GoFW = False
                EndTurn = True


        # Go backward
        if GoBW:
            curr_state[0] = curr_state[0] + v * np.cos(curr_state[2]-np.pi) * dt
            curr_state[1] = curr_state[1] + v * np.sin(curr_state[2]-np.pi) * dt
            input_u.append(np.array([-v, -v]))
            d2target = np.linalg.norm(target_state - curr_state, ord = 2)
            if d2target < v*dt:
                curr_state[0] = target_state[0]
                curr_state[1] = target_state[1]
                GoBW = False
                EndTurn = True

        # Turning to the ending state heading
        if EndTurn:
            alpha = (target_state[2]-curr_state[2]) % (2 * np.pi)
```

```
            if (np.abs(alpha) < np.pi):
                if alpha < 0:
                    curr_state[2] = curr_state[2] - omega * dt
                    input_u.append(np.array([v, -v]))

                else:
                    curr_state[2] = curr_state[2] + omega * dt
                    input_u.append(np.array([-v, v]))
            else:
                if alpha < 0:
                    curr_state[2] = curr_state[2] + omega * dt
                    input_u.append(np.array([-v, v]))

                else:
                    curr_state[2] = curr_state[2] - omega * dt
                    input_u.append(np.array([v, -v]))


            # angle to the goal relative to the heading of the robot
            alpha = (target_state[2]-curr_state[2]) % (2 * np.pi)

        if EndTurn and np.abs(alpha) < omega*dt:
            curr_state[2] = target_state[2]
            break;

        if (time_limted != 0):
            t += dt
        else:
            t = 0

    return traj, input_u
```

In [17]:
```
obstaclelist=[
    (0, 300, 200, 150),
    (400, 300, 200, 150)]
initial_state = np.array([100, 100, np.pi])
goal_state = np.array([300, 350, 0])

traj, input_u=line_trajectory(initial_state,goal_state,0)

ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
#draw_Car(initial_state)
draw_Car(goal_state)
```



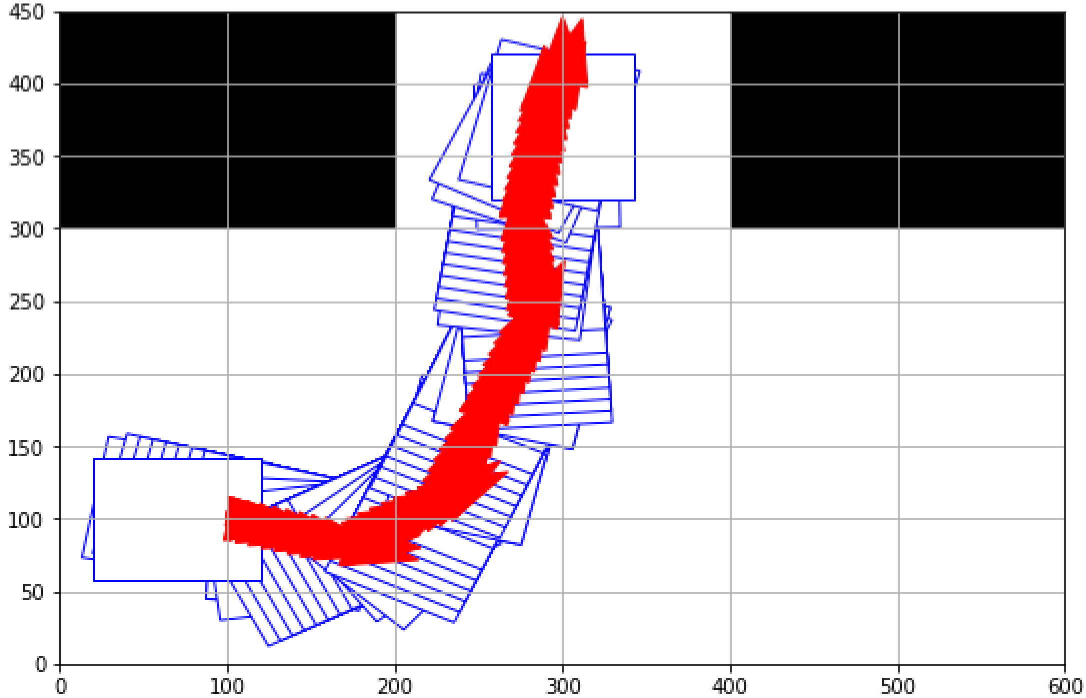
In [18]:
```
# new rrt using line_trajectory generation
def new_rrt(initial_state, goal_state, boundary, max_step=2000):
    start_node = Node(initial_state)
    end_node = Node(goal_state)
    nodelist = [start_node]

    np.random.seed(10)
    done = False
    for i in range(max_step):
        # Distance at the lastest node to goal
        d2goal = np.linalg.norm(goal_state - nodelist[-1].state.copy(), ord = 2)
        # If the goal is inside the circle around car means reach the goal, stop expand the tree
        if d2goal < car_collision_d/2:
            done = True
            break

        # Take a random point in C_space (the point is collision free)
        pnt_random = np.zeros(3)
        while True:
            for i in range(3):
                pnt_random[i] = np.random.uniform(sample_boundary[i][0], sample_boundary[i][1])
            tmp = []
            tmp.append(np.array(pnt_random))
            if (collision(tmp,obstaclelist) == 0):
                break;

         # Find nearest node in the nodelist to the random point
        nearestNode, min_index = closest_node(nodelist, pnt_random)
        # make a trajectory from this nearest point to the random point
        traj_temp, _ = line_trajectory(nearestNode.state.copy(), pnt_random, time_limted=1)

        # Check this trajectory for collision free, add this node to the nodelist if collision free
        if collision(traj_temp, obstaclelist)==0:
            newNode = Node(traj_temp[-1])
            newNode.parent = min_index
            nodelist.append(newNode)
        # print("Number of node in Nodelist:", len(nodelist))

    if (done==False):
        print ("Cannot find the path to the goal.")
        return [],[]
    else:
```

```
    # Find the path from start point to goal
    path = [goal_state]
    i = len(nodelist) - 1
    while nodelist[i].parent is not None:
        node = nodelist[i]
        path.append(node.state)
        i = node.parent
    path.append(initial_state)

    # Generate the traj from start to end point
    traj = []
    i = len(path) - 1
    while i >= 1:
        tmp, _ = line_trajectory(path[i], path[i-1], time_limted=0)
        traj = traj + tmp
        i = i - 1
    return traj, path
```

In [94]:
```
# testing the line_trajectory with rrt
initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = new_rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
path
```

```
Time to calculate for the path: 0.03998684883117676
```

Out[94]:
```
[array([300.        , 400.        ,   1.57079633]),
 array([290.41253417, 379.97360754,   1.58889201]),
 array([277.9496223 , 307.43069857,   1.63548513]),
 array([282.60396835, 235.58129312,   1.15400122]),
 array([253.45606331, 169.74513565,   1.14335429]),
 array([226.92522581, 111.50327217,   0.46955325]),
 array([169.85191208,  82.54404461,  -0.24488404]),
 array([100, 100,   0])]
```

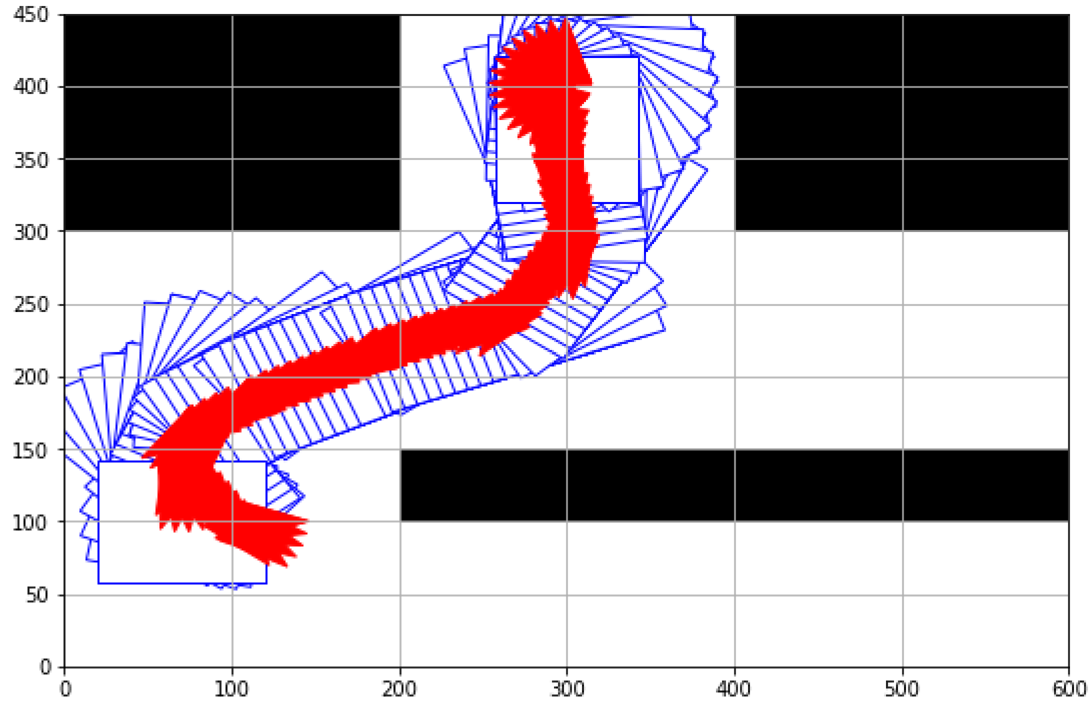


In [95]:
```
# testing the line_trajectory with rrt
obstaclelist=[
    (0, 300, 200, 150),
    (200,100,400,50),
    (400, 300, 200, 150)]

initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = new_rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
path
```

```
Time to calculate for the path: 0.027989864349365234
```

Out[95]:

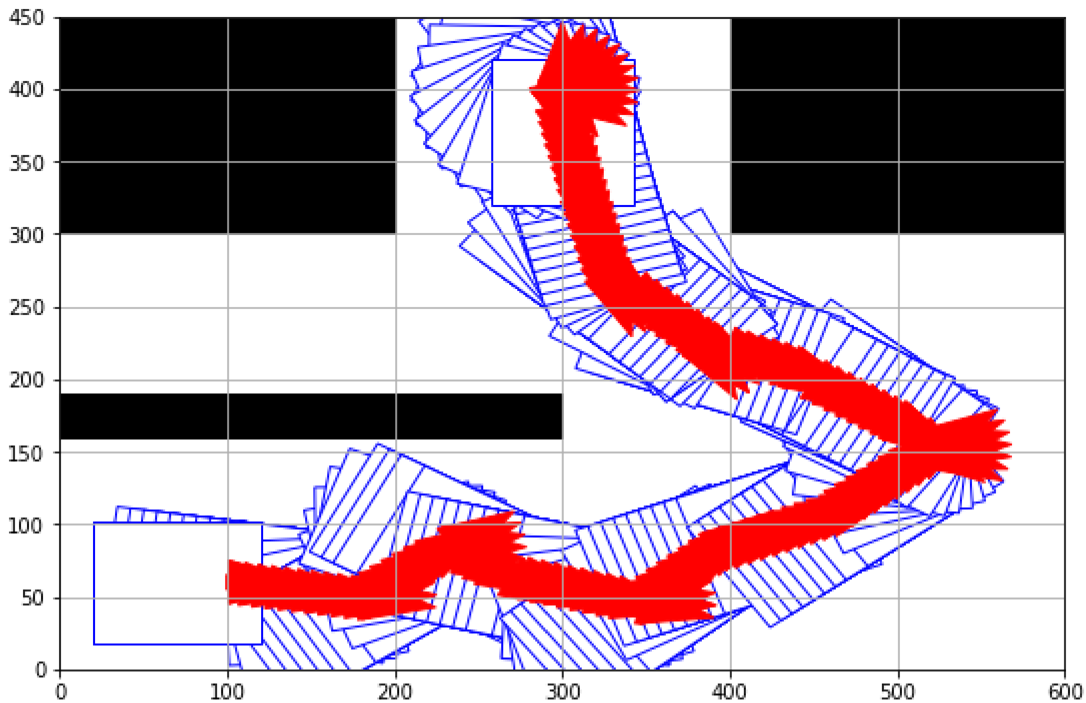
```
[array([300.        , 400.        ,   1.57079633]),
 array([294.9513512 , 353.97082969,   4.88519919]),
 array([304.58062806, 298.80492512,   4.12714594]),
 array([269.2269008 , 245.45594229,   3.47361251]),
 array([193.59602923, 219.37968366,   3.55077326]),
 array([120.20025869, 187.55106654,   3.65218384]),
 array([ 85.30202856, 168.00334202,   4.64507927]),
 array([ 71.24410023, 138.43303564,   5.35474214]),
 array([100, 100,   0])]
```



In [96]:
```
# testing the line_trajectory with rrt
obstaclelist=[
    (0, 300, 200, 150),
    (0,160,300,30),
    (400, 300, 200, 150)]

initial_state = np.array([100, 60, 0])
goal_state = np.array([300, 400, np.pi/2]) #heading parking
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
start_time = time.time()
traj, path = new_rrt(initial_state, goal_state, sample_boundary, max_step=2000)
end_time = time.time()
print("Time to calculate for the path: %s " %str(end_time - start_time))

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
path
```

```
Time to calculate for the path: 0.11996340751647949
```

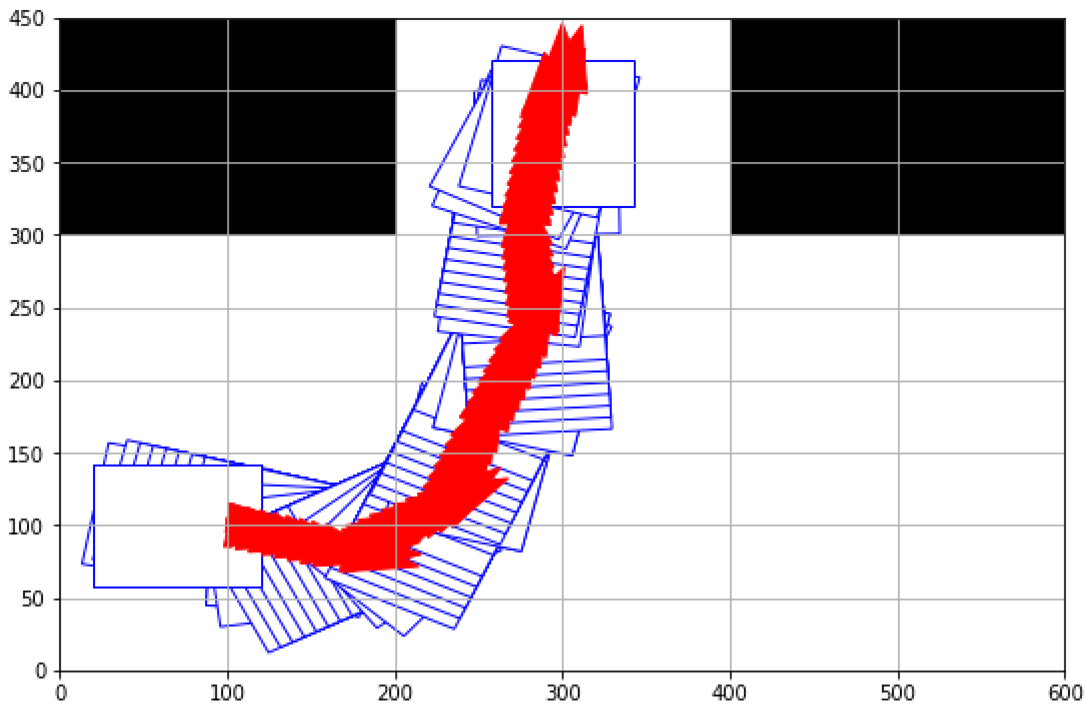
Out[96]:
```
[array([300.        , 400.        ,   1.57079633]),
 array([295.05183436, 404.84398533,   4.93991913]),
 array([313.09759629, 326.9058658 ,   4.97282302]),
 array([327.51759514, 272.79428078,   5.58677586]),
 array([382.75242045, 226.60863287,   5.19820528]),
 array([435.18559614, 208.52943678,   5.75036532]),
 array([497.20481517, 171.95599737,   5.61950371]),
 array([522.41216344, 152.24332142,   0.62549457]),
 array([4.57558271e+02, 1.05403459e+02, 3.50739497e-01]),
 array([389.94171179,  80.66480807,   0.66432892]),
 array([ 3.45851192e+02,  4.61390770e+01, -2.33389767e-01]),
 array([268.02015156,  64.64121376,  -0.49935047]),
 array([232.90440044,  83.79543061,   0.57600824]),
 array([ 1.79231213e+02,  4.89358709e+01, -1.38746343e-01]),
 array([100,  60,   0])]
```



In [125]:
```
def path2actions(initial_state, target_state):

    GoFW = False
    GoBW = False

    action = [0,0]
    alpha = (np.arctan2(target_state[1]-initial_state[1], target_state[0]-initial_state[0]) -
                 initial_state[2] + np.pi) % (2 * np.pi) - np.pi

    if alpha > np.pi / 2 or alpha < -np.pi / 2:
        GoBW = True
        if (alpha > 0):
            action[0] =  (alpha - np.pi)
        else:
            action[0] = (np.pi + alpha)
    else:
        GoFW = True
        if (alpha > 0):
            action[0] = alpha
        else:
            action[0] = alpha



    # Go forward
    if GoFW:
        action[1] = np.linalg.norm(target_state - initial_state, ord = 2)
    # Go backward
    if GoBW:
        action[1] =  -np.linalg.norm(target_state - initial_state, ord = 2)
    return action
```

In [126]:
```
def action_array(path):
    A=[]
    for i in range(len(path)-1):
        A.append(path2actions(path[len(path)-i-1],path[len(path)-i-2]))
    return A
```

In [127]:
```
obstaclelist=[
    (0, 300, 200, 150),
    (400, 300, 200, 150)]

initial_state = np.array([100, 100, 0])
goal_state = np.array([300, 400, np.pi/2])
sample_boundary = [(0, 600), (0, 450), (-np.pi, np.pi)]
traj, path = new_rrt(initial_state, goal_state, sample_boundary, max_step=2000)

a=path2actions(initial_state,goal_state)

A = action_array(path)

# Draw the path on map
ax = draw_Ospace(obstaclelist)
for node in traj:
    draw_Car(node)
draw_Car(initial_state)
draw_Car(goal_state)
```



In [128]:
```
print("action array to control robot [turning angle, go straight distance]")
A
```

```
action array to control robot [turning angle, go straight distance]
```

Out[128]:
```
[[-0.24488403875443465, 72.0004164445764],
 [0.7144372870188045, 64.00398753700496],
 [0.6738010464317008, 64.00354683804773],
 [0.010646921930092645, 72.00000078720093],
 [0.48148391236315247, 72.00160989004242],
 [-0.2348284155806084, 73.60570620102806],
 [-0.464592342668813, 22.203067880552975]]
```

In [ ]: