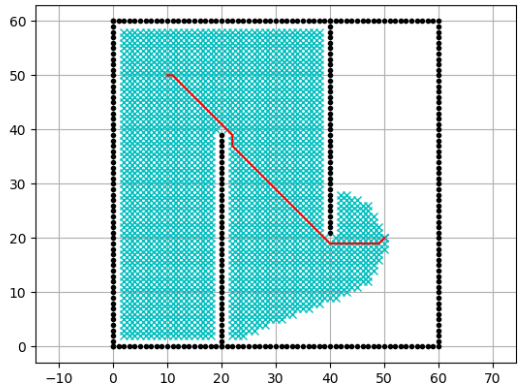
Ignacio Gonzalez Portillo

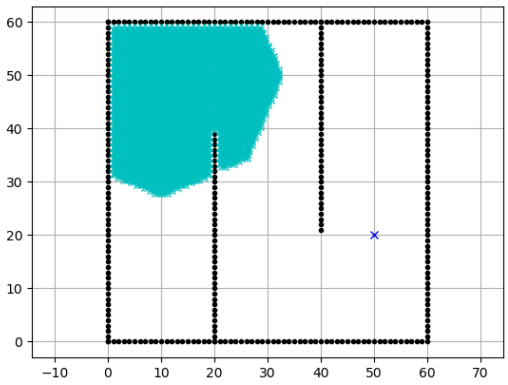
Ejercicio 1: Dijkstra Global Planner

1.2 Capturas

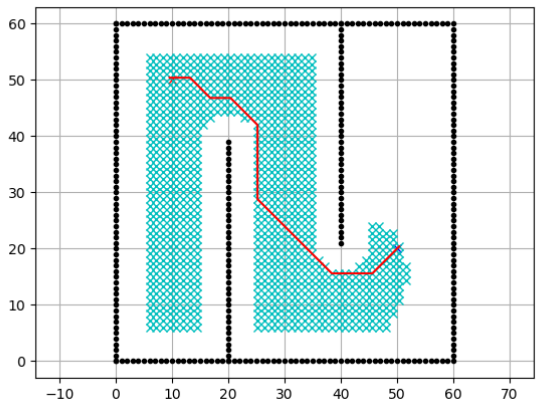
Modificación posición inicial y final



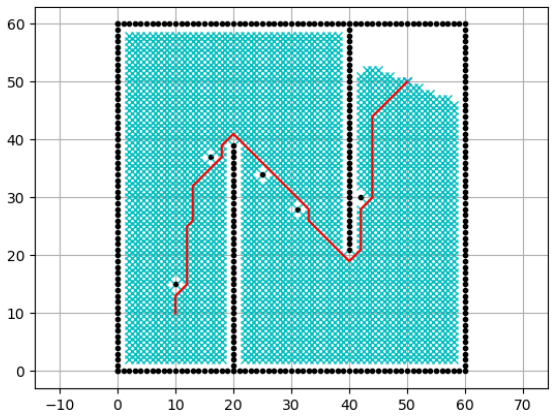
Modificación resolución



Modificación tamaño del robot



Obstaculos añadidos



En el caso en el que se añaden obstaculos podemos ver como la trayectoria generada es similar a la original, pero en este caso se evitan los obstaculos, pues en estas zonas no hay nodos y por lo tanto no hay caminos posibles para recorrer.

1.3 Explicación

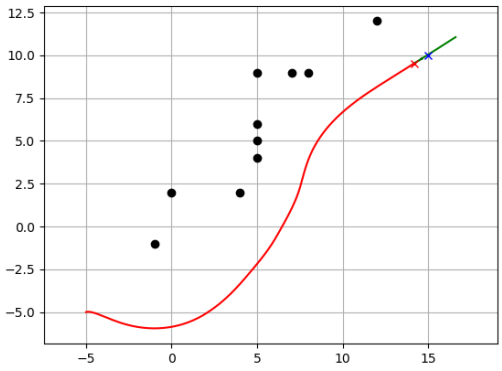
En general Dijkstra es un algoritmo capaz de producir taryectorias libres de colision entre dos puntos en un grafo de nodos, asegurando el camino más optimo posible. En un grafo de nodos el camino para ir de un nodo a otro (edge) es pesado segun diferentes factores, en este caso la distancia recorrida. El algoritmo se encarga de visitar todos los nodos adyacentes al nodo donde se encuentra y asignar un peso a cada camino segun la distancia recorrida para ir de un nodo a otro y de esta manera se expande hasta visitar todos los nodos del grafo. Una vez todos los nodos han sido visitados, y todos los caminos han sido pesados, dado un punto de origen y un punto final (dos nodos) el algoritmo puede dar el camino que conecte ambos nodos recorriendo la menor distancia, es decir, el camino con menor peso. En el caso concreto aqui tratado y en la mayoria de implementaciones, el algoritmo no recorre absolutamente todos los nodos, pues desde el inicio se conoce qual es la meta o nodo al que se desea llegar y el nodo de origen, por lo que el algoritmo se expande partiendo de la salida hacia los nodos adyacentes hasta que que llega al nodo de la meta y genera el camino más corto. Esto permite ahorrar coste computacional al no visitar todos los nodos del grafo.

Ignacio Gonzalez Portillo

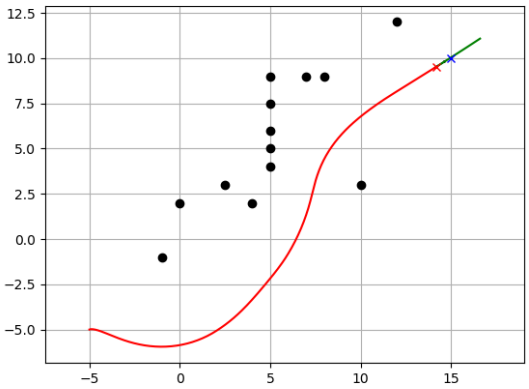
Ejercicio 2: Dynamic Window Approach local planner (Ventana Dinámica)

2.2 Capturas

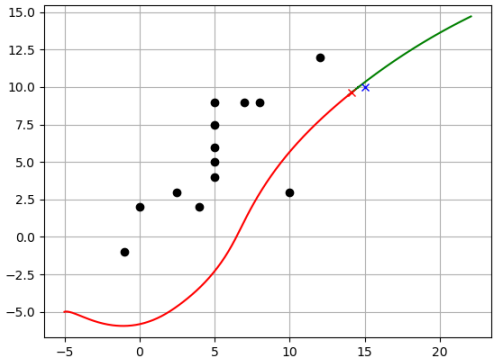
Modificación salida y meta



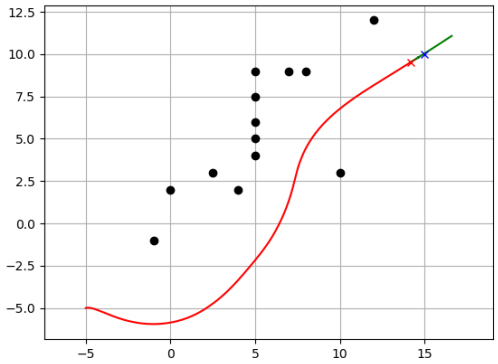
Obstaculos añadidos



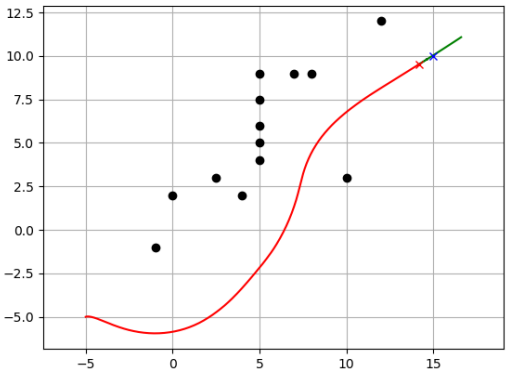
Augmento velocidad máxima



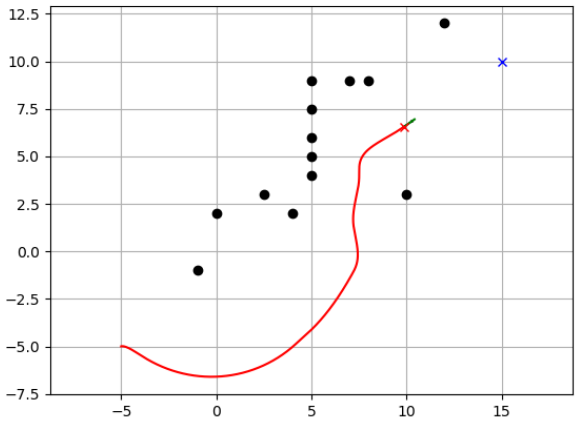
Disminución velocidad mínima



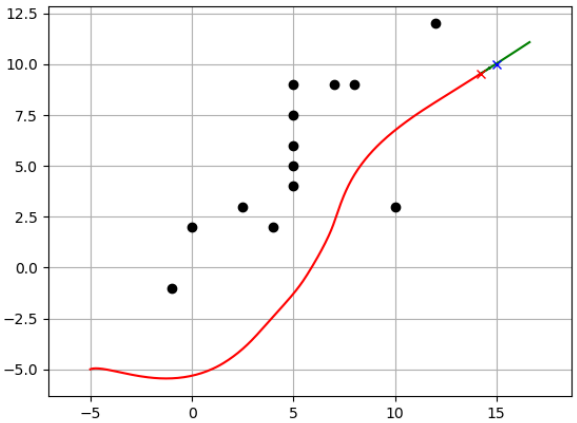
Augmento velocidad angular máxima



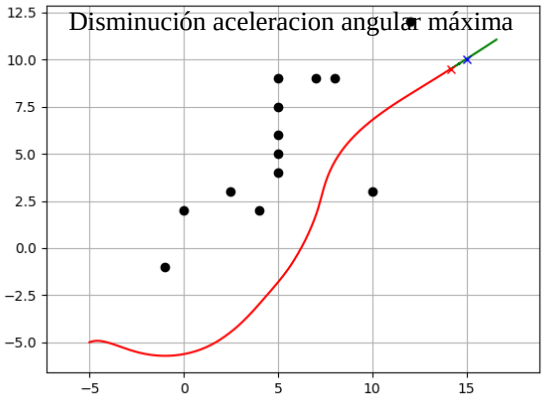
Disminución aceleración máxima



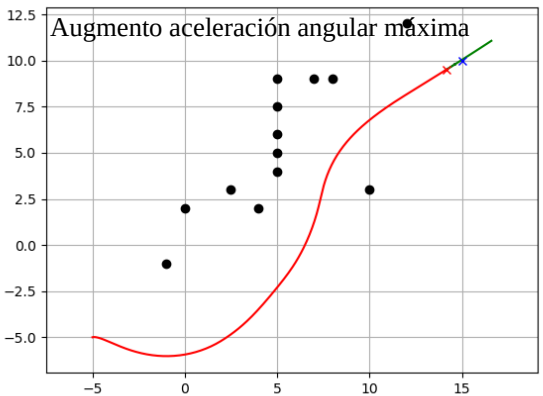
Augmento aceleración máxima



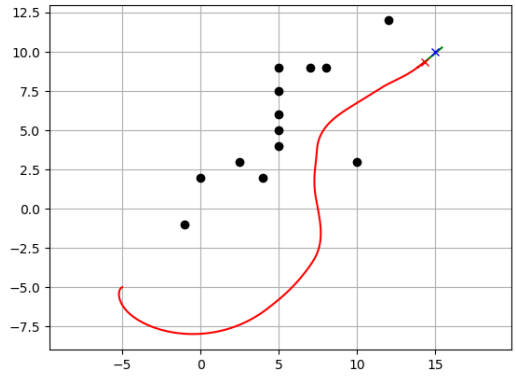
Disminución aceleracion angular máxima



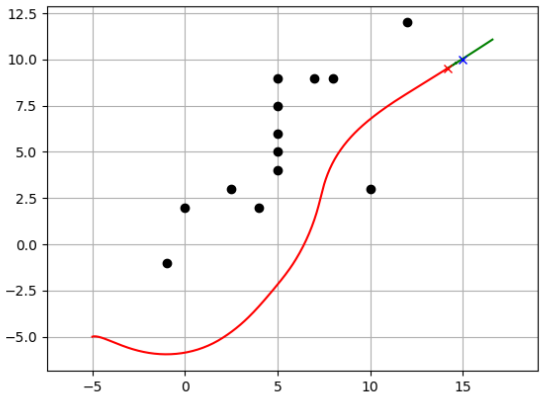
Augmento aceleración angular máxima



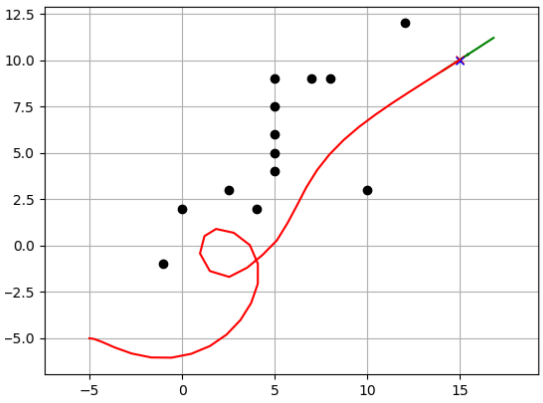
Augmentar resolución velocidad



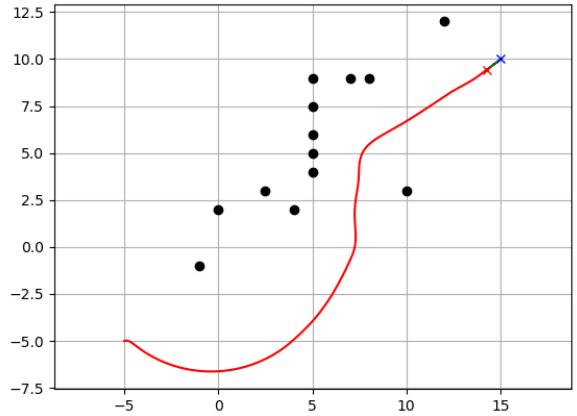
Augmentar coste velocidad



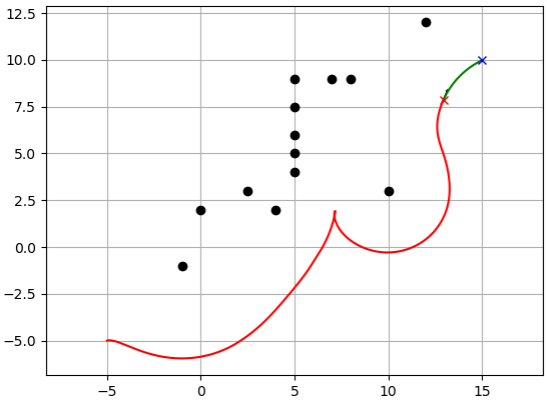
Augmentar diferencial de tiempo



Disminución predicción tiempo



Augmentar radio robot



Se adjuntan algunos de los resultados de introducir variaciones en los parametros de serie del planificador local.

Primero se han variado los parámetros referentes a la dinámica del movimiento del robot (velocidades y aceraciones maximas), podemos apreciar pocas variaciones en la trayectoria seguida. Estos parámetros amplian o restringen el aspectro de velocidades y aceleraciones que seran evaluadas por el algoritmo para proponer soluciones, entoces si no se excluye fuera de los limites ninguna solución valida, las trayectorias no deberian verse afectadas por estos parametros.

En cuanto al resto de parámetros podriamos decir que sí tienen una mayor influencia en la geometría de la trayectoria. La resolución de las velocidades afecta en como se realiza el muestreo del aspectro de velocidades a evaluar, a cuanta más resolución menor sera la discretización del aspectro de velocidades y menos soluciones se podran evaluar y proponer. Disminuir la resolución hara que la trayectoria sea más "precisa" a costa de augmentar el coste computacional. El mismo concepto se aplica para la resolución de la velocidad angular.

El diferencial de tiempo determina el tiempo que se aplica cada solución, és decir cuanto tiempo se recorre cada arco de circumferencia, por este motivo al augmentar este parámetro las trayectorias son cada vez más circulares pues cada familia de velocidades escojidas como solución, se aplican de manera constante durante más tiempo.

Las ganancias del coste de la orientación a la meta y del coste de velocidad, sirven para pesar ambos factores en la *Objective Function* que se optimiza para encontrar las soluciones. Por defecto ambos factores tienen el mismo peso y deben fijarse en función de la geometria del mapa.

El radió del robot tiene un claro efecto en la trayectoria por restricciones geometricas obvias.

Cabe destacar que algunos de los parámetros presentan limitaciones, y al subir por encima de un cierto valor el algoritmo no es capaz de encontrar una trayectoria valida para llegar a la meta, esto sucede por ejemplo con la predicción de tiempo. Este parámetro controla para cuanto tiempo se va a simular la trayectoria de cada familia de velocidades

En general cada uno de estos parametros tiene una influencia particular sobre el comportamiento del algoritmo, pero es difícil dar una explicación exacta e unívoca de como afectan individualmente cada uno de ellos a la trayectoria final. En mi opinión, para cada aplicación se deberia realizar un "tunning" exhaustivo y exclusivo de estos parametros para ajustarlos al comportamiento deseado en cada caso y a la geometria del entorno.

2.3 Explicación

Este algoritmo funciona proponiendo soluciones en el espacio de velocidades (angular y lineal), estas soluciones cumplen por un lado con las restricciones del modelo dinamico del robot y por otro lado solo se incluiran aquellas que no den lugar a colisión o bien que el robot pueda frenar antes de esta. Las soluciones propuestas son trayectorias circulares que se caracterizan por una velocidad lineal y angular unicas. Estas velocidades se aplicaran durante un intervalo de tiempo determinado recorriendo solo una parte del arco. A cada intervalo de tiempo se proponen nuevas soluciones. A cada vuelta del loop, el algoritmo solo tiene en cuenta las velocidades del primer intervalo, asumiendo que la velocidad es constante en el resto. Entre las velocidades clasificadas como admisibles, la ventana dinámica restringe el espacio de la solución a aquellas a las que el robot pueda llegar en el siguiente intervalo de tiempo con su aceleración actual es por este motivo que para cada intervalo de tiempo se proponen nuevas soluciones.

Una vez el espacio de soluciones (velocidades) esta definido y delimitado, se utiliza una función llamada "*Objective Function"* que realiza una suma ponderada de los siguientes factores; "target *heading*" (progreso/ orientación del sistema respecto a la meta, mayor como mejor sea la orientación respecto a la meta), "*clearance*" (distancia al objeto de colision más proximo de la trayectoria, mayor como mayor sea la distancia a la colisión), y "*velocity*" (medida del avance del robot dentro de la trayectoria circular evaluada).

El objetivo final es optimizar esta función de manera que se encuentre la velocidad (lineal y angular) que maximize esta función.

Objective Function

$$G(v; !) = \sigma * (\alpha * heading(v; w) + \beta * dist(v; w) + \gamma * velocity(v; w))$$

En el codigo trabajado en el ejercicio, la "*Objective Function*" se encuentra implementada de la siguiente forma:

$$to\_goal\_cost = calc\_to\_goal\_cost(traj, goal, config)$$

$$speed\_cost = config.speed\_cost\_gain * /(config.max\_speed - traj[-1, 3])$$

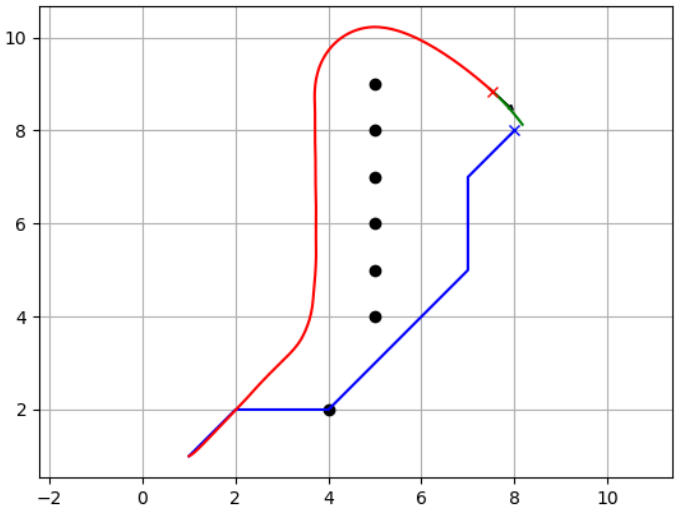
$$ob\_cost = calc\_obstacle\_cost(traj, ob, config)$$

$$final\_cost = to\_goal\_cost + speed\_cost + ob\_cost$$

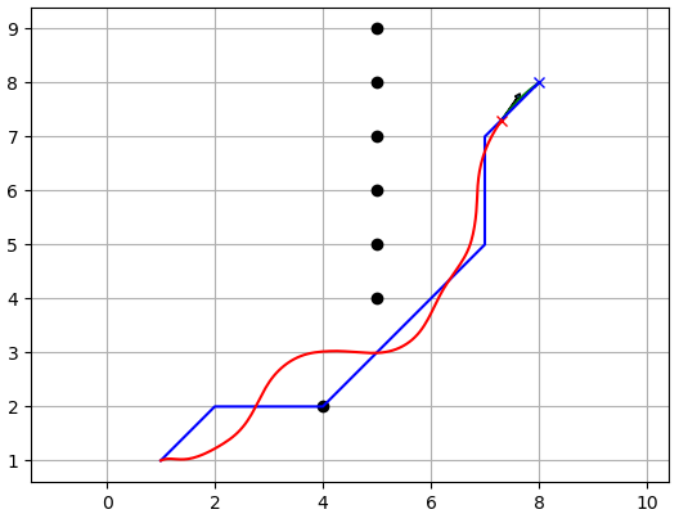
Podemos ver como la expresión que encontramos implementada es identica a la original, excepto porque no aparece un termino analogo a la beta que se utiliza para pesar el factor de distancia de colisión, y ademas tampoco aparece el termino sigma (σ) multiplicando la suma de todos los factores.

Ejercicio 3:

DWA sin seguir el path de Dijkstra



DWA siguiendo el path de Dijkstra



Ignacio Gonzalez Portillo

Script de Python

```
"""
Dijkstra grid based planning

author: Atsushi Sakai(@Atsushi_twi)
"""

import matplotlib.pyplot as plt
import math
import numpy as np
import sys



show_animation = True


class Node:

    def __init__(self, x, y, cost, pind):
        self.x = x
        self.y = y
        self.cost = cost
        self.pind = pind

    def __str__(self):
        return str(self.x) + "," + str(self.y) + "," + str(self.cost) + "," + str(self.pind)


def dijkstra_planning(sx, sy, gx, gy, ox, oy, reso, rr):
    """
    gx: goal x position [m]
    gy: goal y position [m]
    ox: x position list of Obstacles [m]
    oy: y position list of Obstacles [m]
    reso: grid resolution [m]
    rr: robot radius[m]
    """

    nstart = Node(round(sx / reso), round(sy / reso), 0.0, -1)
    ngoal = Node(round(gx / reso), round(gy / reso), 0.0, -1)
    ox = [iox / reso for iox in ox]
    oy = [ioy / reso for ioy in oy]

    obmap, minx, miny, maxx, maxy, xw, yw = calc_obstacle_map(ox, oy, reso, rr)

    motion = get_motion_model()
```

```
    openset, closedset = dict(), dict()
    openset[calc_index(nstart, xw, minx, miny)] = nstart

    while 1:
        c_id = min(openset, key=lambda o: openset[o].cost)
        current = openset[c_id]
        #  print("current", current)

        # show graph
        if show_animation:
            plt.plot(current.x * reso, current.y * reso, "xc")
            if len(closedset.keys()) % 10 == 0:
                plt.pause(0.001)

        if current.x == ngoal.x and current.y == ngoal.y:
            print("Find goal")
            ngoal.pind = current.pind
            ngoal.cost = current.cost
            break

        # Remove the item from the open set
        del openset[c_id]
        # Add it to the closed set
        closedset[c_id] = current

        # expand search grid based on motion model
        for i in range(len(motion)):
            node = Node(current.x + motion[i][0], current.y + motion[i][1],
                        current.cost + motion[i][2], c_id)
            n_id = calc_index(node, xw, minx, miny)

            if not verify_node(node, obmap, minx, miny, maxx, maxy):
                continue

            if n_id in closedset:
                continue
            # Otherwise if it is already in the open set
            if n_id in openset:
                if openset[n_id].cost > node.cost:
                    openset[n_id].cost = node.cost
                    openset[n_id].pind = c_id
            else:
                openset[n_id] = node

    rx, ry = calc_final_path(ngoal, closedset, reso)

    return rx, ry


def calc_final_path(ngoal, closedset, reso):
```

```
    # generate final course
    rx, ry = [ngoal.x * reso], [ngoal.y * reso]
    pind = ngoal.pind
    while pind != -1:
        n = closedset[pind]
        rx.append(n.x * reso)
        ry.append(n.y * reso)
        pind = n.pind

    return rx, ry


def verify_node(node, obmap, minx, miny, maxx, maxy):

    if obmap[int(node.x)][int(node.y)]:
        return False

    if node.x < minx:
        return False
    elif node.y < miny:
        return False
    elif node.x > maxx:
        return False
    elif node.y > maxy:
        return False

    return True


def calc_obstacle_map(ox, oy, reso, vr):

    minx = round(min(ox))
    miny = round(min(oy))
    maxx = round(max(ox))
    maxy = round(max(oy))


    xwidth = round(maxx - minx)
    ywidth = round(maxy - miny)


    # obstacle map generation
    obmap = [[False for i in range(int(xwidth))] for i in range(int(ywidth))]
    for ix in range(int(xwidth)):
        x = ix + minx
        for iy in range(int(ywidth)):
            y = iy + miny
            #  print(x, y)
            for iox, ioy in zip(ox, oy):
                d = math.sqrt((iox - x)**2 + (ioy - y)**2)
```

```
                if d <= vr / reso:
                    obmap[ix][iy] = True
                    break

    return obmap, minx, miny, maxx, maxy, xwidth, ywidth


def calc_index(node, xwidth, xmin, ymin):
    return (node.y - ymin) * xwidth + (node.x - xmin)


def get_motion_model():
    # dx, dy, cost
    motion = [[1, 0, 1],
              [0, 1, 1],
              [-1, 0, 1],
              [0, -1, 1],
              [-1, -1, math.sqrt(2)],
              [-1, 1, math.sqrt(2)],
              [1, -1, math.sqrt(2)],
              [1, 1, math.sqrt(2)]]

    return motion


"""

Mobile robot motion planning sample with Dynamic Window Approach

author: Atsushi Sakai (@Atsushi_twi)

"""



sys.path.append("../../")


show_animation = True


class Config():
    # simulation parameters

    def __init__(self):
        # robot parameter
        self.max_speed = 0.7  # [m/s]
        self.min_speed = -0.5  # [m/s]
        self.max_yawrate = 40.0 * math.pi / 180.0  # [rad/s]
        self.max_accel = 0.3  # [m/ss]
```

```
        self.max_dyawrate = 40.0 * math.pi / 180.0  # [rad/ss]
        self.v_reso = 0.01  # [m/s]
        self.yawrate_reso = 0.1 * math.pi / 180.0  # [rad/s]
        self.dt = 0.1  # [s]
        self.predict_time = 1.5  #[s]
        self.to_goal_cost_gain = 1.5
        self.speed_cost_gain = 1.0
        self.robot_radius = 1.0  # [m]


def motion(x, u, dt):
    # motion model

    x[2] += u[1] * dt
    x[0] += u[0] * math.cos(x[2]) * dt
    x[1] += u[0] * math.sin(x[2]) * dt
    x[3] = u[0]
    x[4] = u[1]

    return x


def calc_dynamic_window(x, config):

    # Dynamic window from robot specification
    Vs = [config.min_speed, config.max_speed,
          -config.max_yawrate, config.max_yawrate]

    # Dynamic window from motion model
    Vd = [x[3] - config.max_accel * config.dt,
          x[3] + config.max_accel * config.dt,
          x[4] - config.max_dyawrate * config.dt,
          x[4] + config.max_dyawrate * config.dt]

    #  [vmin,vmax, yawrate min, yawrate max]
    dw = [max(Vs[0], Vd[0]), min(Vs[1], Vd[1]),
          max(Vs[2], Vd[2]), min(Vs[3], Vd[3])]

    return dw


def calc_trajectory(xinit, v, y, config):

    x = np.array(xinit)
    traj = np.array(x)
    time = 0
    while time <= config.predict_time:
        x = motion(x, [v, y], config.dt)
        traj = np.vstack((traj, x))
        time += config.dt
```

```
    return traj


def calc_final_input(x, u, dw, config, goal, ob):

    xinit = x[:]
    min_cost = 10000.0
    min_u = u
    min_u[0] = 0.0
    best_traj = np.array([x])

    # evalucate all trajectory with sampled input in dynamic window
    for v in np.arange(dw[0], dw[1], config.v_reso):
        for y in np.arange(dw[2], dw[3], config.yawrate_reso):
            traj = calc_trajectory(xinit, v, y, config)

            # calc cost
            to_goal_cost = calc_to_goal_cost(traj, goal, config)
            speed_cost = config.speed_cost_gain * \
                (config.max_speed - traj[-1, 3])
            ob_cost = calc_obstacle_cost(traj, ob, config)
            # print(ob_cost)

            final_cost = to_goal_cost + speed_cost + ob_cost

            #print (final_cost)

            # search minimum trajectory
            if min_cost >= final_cost:
                min_cost = final_cost
                min_u = [v, y]
                best_traj = traj

    return min_u, best_traj


def calc_obstacle_cost(traj, ob, config):
    # calc obstacle cost inf: collistion, 0:free

    skip_n = 2
    minr = float("inf")

    for ii in range(0, len(traj[:, 1]), skip_n):
        for i in range(len(ob[:, 0])):
            ox = ob[i, 0]
            oy = ob[i, 1]
            dx = traj[ii, 0] - ox
            dy = traj[ii, 1] - oy
```

```
            r = math.sqrt(dx**2 + dy**2)
            if r <= config.robot_radius:
                return float("Inf")  # collision

            if minr >= r:
                minr = r

    return 1.0 / minr  # OK


def calc_to_goal_cost(traj, goal, config):
    # calc to goal cost. It is 2D norm.

    goal_magnitude = math.sqrt(goal[0]**2 + goal[1]**2)
    traj_magnitude = math.sqrt(traj[-1, 0]**2 + traj[-1, 1]**2)
    dot_product = (goal[0] * traj[-1, 0]) + (goal[1] * traj[-1, 1])
    error = dot_product / (goal_magnitude * traj_magnitude)
    error_angle = math.acos(error)
    cost = config.to_goal_cost_gain * error_angle

    return cost


def dwa_control(x, u, config, goal, ob):
    # Dynamic Window control

    dw = calc_dynamic_window(x, config)

    u, traj = calc_final_input(x, u, dw, config, goal, ob)

    return u, traj


def plot_arrow(x, y, yaw, length=0.5, width=0.1):
    plt.arrow(x, y, length * math.cos(yaw), length * math.sin(yaw),
              head_length=width, head_width=width)
    plt.plot(x, y)


def main():
    print(__file__ + " start!!")

    follow_dijkstra = True
    # start and goal position
    sx = 1.0  # [m]
    sy = 1.0  # [m]
    gx = 8.0  # [m]
    gy = 8.0  # [m]
    grid_size = 1.0  # [m]
    robot_size = 1.0  # [m]
```

```
    ox = [5.0]
    oy = [5.0]

    for i in range(12):
        ox.append(i)
        oy.append(0.0)
    for i in range(13):
        ox.append(12.0)
        oy.append(i)
    for i in range(12):
        ox.append(i)
        oy.append(12.0)
    for i in range(12):
        ox.append(0.0)
        oy.append(i)


    if show_animation:
        plt.plot(ox, oy, ".k")
        plt.plot(sx, sy, "xr")
        plt.plot(gx, gy, "xb")
        plt.grid(True)
        plt.axis("equal")

    rx, ry = dijkstra_planning(sx, sy, gx, gy, ox, oy, grid_size, robot_size)

    if show_animation:
        plt.plot(rx, ry, "-b")
        #plt.show()


#"""if __name__ == '__main__':"""




# initial state [x(m), y(m), yaw(rad), v(m/s), omega(rad/s)]
    x = np.array([1.0, 1.0, math.pi / 8.0, 0.0, 0.0])
# goal position [x(m), y(m)]

# obstacles [x(m) y(m), ....]
    ob = np.array([[4, 2],
                   [5, 4],
                   [5, 5],
                  [5, 6],
                  [5, 7],
                  [5, 8],
                  [5, 9]])
```

```
    u = np.array([0.0, 0.0])
    config = Config()
    traj = np.array(x)



    if follow_dijkstra:

        #Pasamos como goal la mitad de los puntos de rx y ry recorriendo estos elementos con un bucle
        for i in range(len(rx)/2):

            goal = np.array([rx[len(rx)-1-i*2], ry[len(ry)-1-i*2]])

            for i in range(1000):
                u, ltraj = dwa_control(x, u, config, goal, ob)

                x = motion(x, u, config.dt)
                traj = np.vstack((traj, x))  # store state history

                # print(traj)

                if show_animation:
                    plt.cla()
                    plt.plot(ltraj[:, 0], ltraj[:, 1], "-g")
                    plt.plot(x[0], x[1], "xr")
                    plt.plot(goal[0], goal[1], "xb")
                    plt.plot(ob[:, 0], ob[:, 1], "ok")
                    plt.plot(rx, ry, "-b")
                    plot_arrow(x[0], x[1], x[2])
                    plt.axis("equal")
                    plt.grid(True)
                    plt.pause(0.0001)

                # check goal
                if math.sqrt((x[0] - goal[0])**2 + (x[1] - goal[1])**2) <= config.robot_radius:
                    print("Goal!!")
                    break

        #Aseguramos que el ultimo punto del path (goal) se pasa al dwa planner, el primer punto de la lista corresponde al ultimo de la trayectoria.
        goal = np.array([rx[0], ry[0]])


        for i in range(1000):
            u, ltraj = dwa_control(x, u, config, goal, ob)

            x = motion(x, u, config.dt)
            traj = np.vstack((traj, x))  # store state history

            # print(traj)
```

```
        if show_animation:
            plt.cla()
            plt.plot(ltraj[:, 0], ltraj[:, 1], "-g")
            plt.plot(x[0], x[1], "xr")
            plt.plot(goal[0], goal[1], "xb")
            plt.plot(ob[:, 0], ob[:, 1], "ok")
            plt.plot(rx, ry, "-b")
            plot_arrow(x[0], x[1], x[2])
            plt.axis("equal")
            plt.grid(True)
            plt.pause(0.0001)

        # check goal
        if math.sqrt((x[0] - goal[0])**2 + (x[1] - goal[1])**2) <= config.robot_radius:
            print("Goal!!")
            break



else:

    goal = np.array([8, 8])

    for i in range(1000):
        u, ltraj = dwa_control(x, u, config, goal, ob)

        x = motion(x, u, config.dt)
        traj = np.vstack((traj, x))  # store state history

        # print(traj)

        if show_animation:
            plt.cla()
            plt.plot(ltraj[:, 0], ltraj[:, 1], "-g")
            plt.plot(x[0], x[1], "xr")
            plt.plot(goal[0], goal[1], "xb")
            plt.plot(ob[:, 0], ob[:, 1], "ok")
            plt.plot(rx, ry, "-b")
            plot_arrow(x[0], x[1], x[2])
            plt.axis("equal")
            plt.grid(True)
            plt.pause(0.0001)

        # check goal
        if math.sqrt((x[0] - goal[0])**2 + (x[1] - goal[1])**2) <= config.robot_radius:
            print("Goal!!")
            break
```

```
    print("Done")
    if show_animation:
        plt.plot(traj[:, 0], traj[:, 1], "-r")
        plt.pause(0.0001)

    plt.show()


if __name__ == '__main__':

    main()
```