Trabajo Final.

Autonomous Robot.

Alumnos: Román Nahuel; Ferreri Delfina.

Materia: Modelling Brains**.**

Profesor: José A. Fernández-Leon Fellenz

Fecha de Entrega: 08/08/2024

Universidad Nacional Del Centro.

1. Resumen.

El presente trabajo se desarrolla en el marco del trabajo final de la cátedra Modelling Brains, perteneciente a la Facultad de Ciencias Exactas de la UNCPBA, que introduce los conceptos fundamentales sobre control neuro-inspirado y la robótica basada en comportamientos.

La consigna implica trabajar en Webots, un software de simulación de robots de código abierto desarrollado por Cyberbotics, con el objetivo de programar un agente autónomo capaz de lograr una meta específica mediante la coordinación de múltiples comportamientos durante la navegación.

El objetivo propuesto para el agente, un robot del tipo Husarion Rosbot, fue explorar el entorno hasta encontrar una señal de radio emitida por un segundo robot, el modelo Elisa3, cuya única función es emitir dichas señales. Una vez hallada la señal, el robot debería alternar a un comportamiento de navegación vectorial que lo llevaría a culminar su recorrido en la proximidad del robot emisor.

Para dicho trabajo se propuso que el agente *aprenda* a medida que explora el entorno, permitiéndole decidir qué comportamiento es el adecuado en cada momento según su estado actual. Para lograr esto se empleó la técnica de aprendizaje por refuerzo Q-Learning.

Los resultados serán comparados con otro robot con las mismas características y comportamientos, pero cuya coordinación de comportamientos no se basa en un aprendizaje, sino en respuestas a estímulos (Vehículo de Braitenberg).

Adicionalmente, al final de cada recorrido se computa un mapa que refleja la trayectoria del robot. Dicho mapa cumple únicamente una función de registro, es decir, el robot lo genera pero no lo utiliza ni para orientarse ni para planificar su ruta.

2. Objetivos.

El presente trabajo tiene por objetivos:

1. Consolidar los contenidos aprendidos a través de la experimentación en un simulador.
2. Implementar un algoritmo de aprendizaje por refuerzo (Q-Learning) para la navegación en el entorno.
3. Implementar mecanismo de mapeo básico del entorno.
4. Comparar el rendimiento del robot adaptativo con el de un robot basado en comportamientos

3. Vehículo de Braitenberg

Los vehículos de Braitenberg son robots teóricos que exhiben comportamientos complejos a partir de reglas muy simples y conexiones directas entre sensores y actuadores. Un robot que se considera un vehículo de Braitenberg reacciona únicamente en función de los valores recibidos de los sensores en un momento dado sin la necesidad de una planificación avanzada o toma de decisiones compleja. Por ejemplo: si detecta un obstáculo mediante el lidar, lo esquiva, y si detecta una señal de radio, sencillamente la sigue para acercarse a la fuente. Las decisiones se toman de manera estática e inmutable, sin variar ni adaptarse con el tiempo.

4. Aprendizaje por Refuerzo.

Dentro del área de inteligencia artificial se desprende una rama llamada *aprendizaje automático*, o *machine learning,* cuyo objetivo es desarrollar técnicas que permitan a un agente aprender a tomar decisiones, identificar patrones, o generar predicciones, entre otras cosas.

Una de las técnicas dentro de esta rama es el *aprendizaje por refuerzo* (*reinforcement learning*) que busca que el agente aprenda mediante la interacción con el entorno, sin necesidad de conocimientos previos o resultados conocidos, con excepción del objetivo final del problema y una señal de refuerzo, positiva o negativa, imitando así una forma de aprendizaje humano.

En marcos generales, esta técnica cuenta con dos componentes principales: el **agente** y el **entorno**. En nuestro caso, el agente será el robot (*Sección 5.1.*) y el entorno (*Sección 5.3*) será el lugar en donde aprenda.

Para lograr que un agente aprenda se necesita de un conjunto de **estados** S (representan al entorno), un conjunto de **acciones** A, una **recompensa positiva** r_p, una **recompensa negativa** r_n y **reglas** que determinen el tipo de recompensa que el agente recibirá. El funcionamiento paso a paso del algoritmo básico es:

1. El agente interacciona con el entorno.
2. Se define el estado S_t en el que se encuentra el agente.
3. El agente realiza una acción.
4. Esta acción afecta al estado del agente en el entorno.
5. La recompensa se determina por la conveniencia o no de las acciones llevadas a cabo.
6. Con la recompensa se busca mejorar el comportamiento posterior.

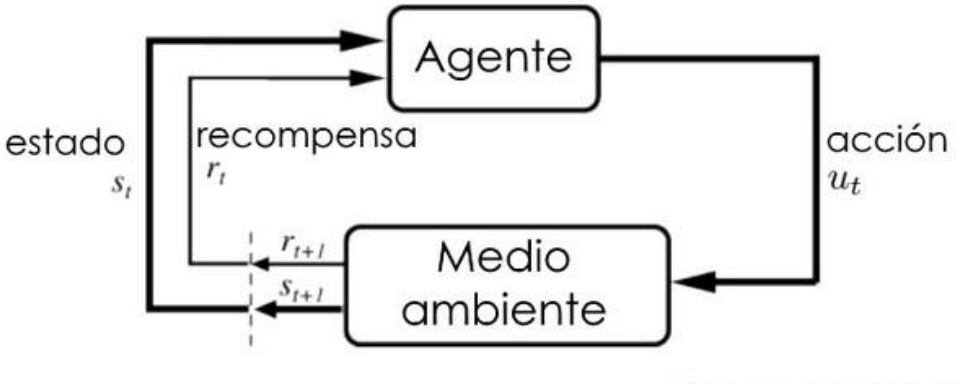

Figura 1. *Esquema del aprendizaje por refuerzo.*

Como se aprecia en la figura anterior, en cada paso t se determina en qué estado S_t se encuentra el agente, para luego ejecutar una acción u_t y recibir una recompensa r_t. Gracias a la acción el estado cambia a S_{t+1} y se brinda una recompensa r_{t+1}.

En el aprendizaje por refuerzo se trata de encontrar un comportamiento óptimo para la resolución del problema aprendiendo de las acciones pasadas y maximizando la recompensa acumulada en el tiempo, generando así una serie de **políticas** que indicarán qué acción tomar en cada estado en el que se encuentre el agente.

5. Componentes.

5.1 Husarion Rosbot.

El robot principal de la simulación es del modelo ROSbot, desarrollado por la compañía polaca Husarion. El ROSbot es un robot móvil autónomo con tracción 4x4, equipado con diversos sensores entre los que se encuentran un LIDAR, una cámara RGB-D, IMU, codificadores y sensores de distancia. Para poder abordar la problemática planteada, se configuró la antena propia del robot para ser únicamente un receptor de señales. Para especificaciones más detalladas sobre el modelo se recomienda visitar la documentación disponible en el sitio web de Husarion.

Figura 2. *Husuarion Rosbot*

5.2 Elisa3

El segundo robot, cuya función se limita únicamente a emitir señales para que el primero las capte, es del modelo Elisa3. Se trata de un pequeño robot de dos ruedas magnéticas, que le permiten moverse de manera horizontal o vertical. Posee sensores infrarrojos de distancia, un acelerómetro y una radio RF, entre otros. Este último componente es el de mayor relevancia para la simulación, y se refiere a un dispositivo que transmite y recibe señales de radiofrecuencia, que son ondas electromagnéticas que se encuentran en el rango de frecuencias de radio. Naturalmente su ubicación en el entorno marca la meta para el Husarion Rosbot, cuya misión es hallar la señal y acercarse a la fuente siguiendo el gradiente de la misma.

5.3 Entorno.

El entorno consiste en una arena rectangular de 8x8 metros, rodeada por paredes negras de baja altura. En el centro de la arena hay una pared adicional con las mismas características, dejando suficiente espacio en ambos extremos para que el robot pueda pasar. El robot debe superar esta dificultad para alcanzar su objetivo. Además de este obstáculo principal, hay algunas cajas de madera distribuidas en la arena que también están allí para desafiar al agente en su navegación hacia el objetivo.

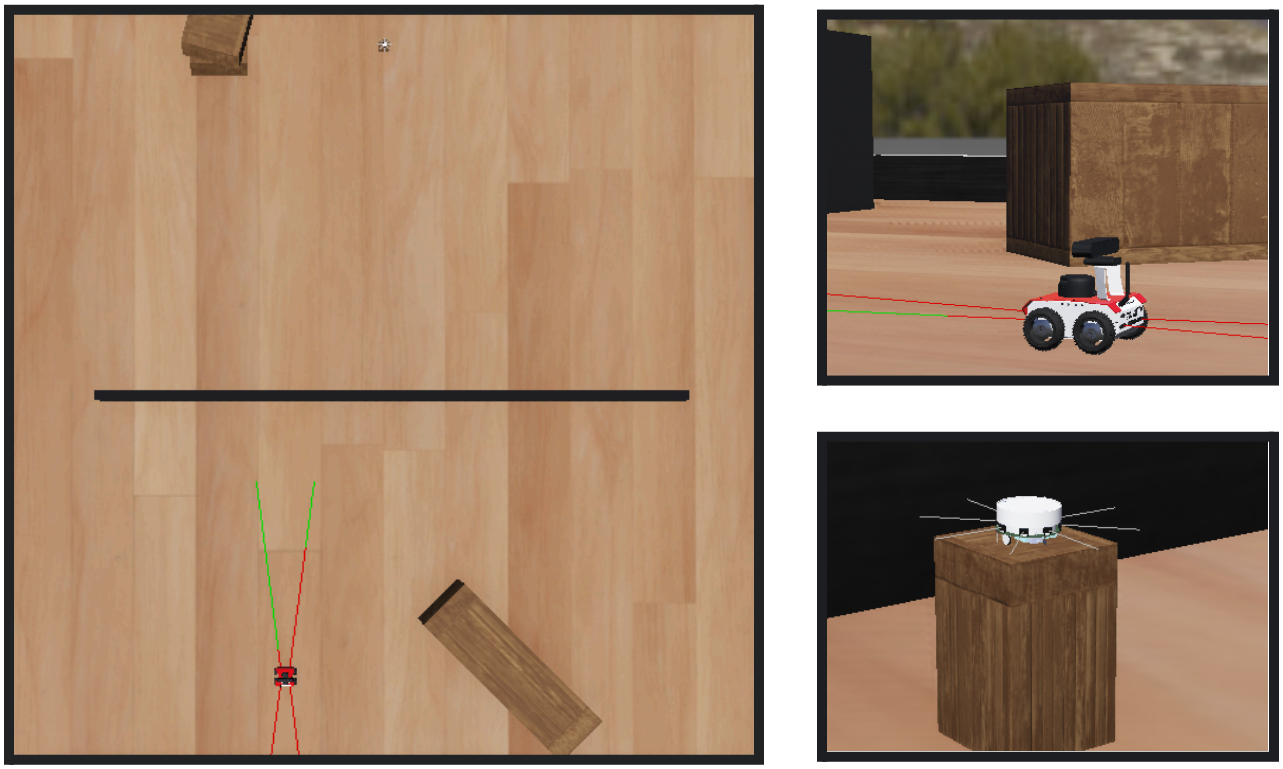

Figura 3. *Entorno*

6. Robot Autónomo Móvil.

Un robot es un agente artificial activo capaz de percibir y actuar en un mundo físico con cierta autonomía. Dentro de esta definición existen varios tipos de robots, de los cuales solo serán mencionados dos tipos que fueron la base para el desarrollo del presente trabajo: *robot autónomo* y *robot móvil*.

Cuando se dice que un agente que tiene autonomía se hace alusión a que el mismo puede realizar tareas por un largo periodo de tiempo sin intervención o supervisión humana. A su vez debe ser capaz de adaptar estrategías en base a la información captada del entorno y podrían tener una capacidad de aprendizaje.

Un robot móvil es aquel que tiene la capacidad de controlar su desplazamiento en el entorno.

En este trabajo se desarrolló un robot autónomo móvil con un criterio de *atracción* y *exploración*, en donde dicho agente se sentirá atraído hacia una señal de radio buscando acercarse lo más posible al emisor. A su vez se vale de una capacidad de exploración.

En la codificación de dicho robot se utilizó el paradigma de programación orientada a objetos en donde se planteó la división de tres clases:

1. HROSbot: tiene la capacidad de ejecutar acciones básicas y tener una lectura e interpretación del entorno. A su vez, declara e inicializa todos los dispositivos del robot.
2. HROSbotComportamental: posee los comportamientos cruciales para el robot.
3. HROSbotInteligente: brinda al robot la capacidad de aprendizaje y entendimiento del entorno.

6.1 Acciones.

Las acciones que puede realizar el robot se encuentran dentro de la clase *HROSbot()*, y representan los movimientos básicos que puede realizar el mismo.

6.1.1 Avanzar.

El robot tendrá la capacidad de avanzar una cantidad determinada de metros siempre y cuando no haya un obstáculo cercano en frente.

Para la realización de dicha acción se codificó un método que recibe por parámetro la distancia y la velocidad con la que se quiere que el robot avance, acto seguido con el lidar se verifica si no hay un obstáculo frente al robot, y por último se activan todos los motores de las ruedas a la velocidad indicada. Por cada paso se miran los sensores de

posición, se determina la distancia recorrida y se vuelve a verificar si no hay un obstáculo al frente. El proceso se repite hasta recorrer los metros indicados. Una vez terminado el proceso los motores se frenan.

```
def avanzar(self, distancia, velocidad):
    print("  ⬆ Avanzar")
    dist = [0, 0]
    dist[0] = 0
    dist[1] = 0

    self.robot.step(self.robotTimestep)

    while ((self.getObstaculoAlFrente()==None)and
           (dist[0]<distancia or dist[1]<distancia)and
           (self.robot.step(self.robotTimestep) != -1)):
        dist =  self.metrosRecorridos()
        self.ruedaDerechaSuperior.setVelocity(velocidad)
        self.ruedaDerechaInferior.setVelocity(velocidad)
        self.ruedaIzquierdaInferior.setVelocity(velocidad)
        self.ruedaIzquierdaSuperior.setVelocity(velocidad)


    self.ruedaDerechaSuperior.setVelocity(0)
    self.ruedaDerechaInferior.setVelocity(0)
    self.ruedaIzquierdaInferior.setVelocity(0)
    self.ruedaIzquierdaSuperior.setVelocity(0)
    self.anteriorValorPositionSensor[0] = self.frontLeftPositionSensor.getValue()
    self.anteriorValorPositionSensor[1] = self.frontRightPositionSensor.getValue()

    self.mapping.update({'type': 'avance', 'value': max(dist)})

    if((dist[0]>=distancia) or (dist[1]>=distancia)):
        return True
    else:
        return False
```

Figura 4. *Método "Avanzar".*

6.1.2 Retroceder.

Al igual que el la acción de avanzar, el robot tendrá un método que le permitirá retroceder.

Utilizando únicamente los sensores de infrarrojos traseros el robot verificará que no haya un obstáculo cerca y de no haberlo se desplazará hacia atrás a una velocidad y por una distancia determinada.

```
def retroceder(self, distancia, velocidad):
    print("  ⬇ Retroceder")
    dist = [0, 0]
    distancia = -1*distancia
    self.robot.step(self.robotTimestep)
    self.anteriorValorPositionSensor[0] = self.frontLeftPositionSensor.getValue()
    self.anteriorValorPositionSensor[1] = self.frontRightPositionSensor.getValue()

    rls = self.rearLeftSensor.getValue()
    rrs = self.rearRightSensor.getValue()

    dist[0] = 0
    dist[1] = 0

    if((rls>distancia and rrs>distancia)):
        while ((dist[0]>distancia or dist[1]>distancia)and
               (self.robot.step(self.robotTimestep) != -1)and
               (rls>distancia and rrs>distancia)):

            dist =  self.metrosRecorridos()
            self.ruedaDerechaSuperior.setVelocity(-velocidad)
            self.ruedaDerechaInferior.setVelocity(-velocidad)
            self.ruedaIzquierdaInferior.setVelocity(-velocidad)
            self.ruedaIzquierdaSuperior.setVelocity(-velocidad)
            rls = self.rearLeftSensor.getValue()
            rrs = self.rearRightSensor.getValue()

    self.ruedaDerechaSuperior.setVelocity(0)
    self.ruedaDerechaInferior.setVelocity(0)
    self.ruedaIzquierdaInferior.setVelocity(0)
    self.ruedaIzquierdaSuperior.setVelocity(0)
```

Figura 5. *Método "retroceder"*

En cada paso se verifica que no hay obstáculo y los metros recorridos, para finalizar cuando esté último parámetro llegue a la distancia determinada.

6.1.3 Girar a la izquierda y girar a la derecha.

La capacidad de giro en un robot móvil es imprescindible, por ello se implementaron dos métodos que cumplen con este requisito.

Para poder generar esta acción el robot verifica con el lidar si es que no hay un obstáculo en la zona hacia donde debe girar. Luego comienza activa los motores de las ruedas necesarios (si el giro es hacia la derecha activa las ruedas izquierdas; si es para la izquierda las ruedas derechas), en cada paso se observará el valor del giroscopio y la acción concluirá cuando el ángulo de giro realizado sea igual al ángulo de giro dado por parámetro.

```
def giroDerecha(self, angulo):
    print("  ➡ Derecha")
    velocidad = 2.0
    ang_z = 0
    giro = False

    self.robot.step(self.robotTimestep)
    lidar_data = self.lidar.getRangeImage()
    lidar_right = lidar_data[25:99]
    obstaculo, min = self.getObstaculo(lidar_right)
    if(obstaculo==None):
        giro = True
        while ((self.robot.step(self.robotTimestep) != -1)and
               (ang_z>(angulo))):
            gyroZ =self.giroscopio.getValues()[2]
            ang_z=ang_z+(gyroZ*self.robotTimestep*0.001)

            self.ruedaDerechaSuperior.setVelocity(0.0)
            self.ruedaDerechaInferior.setVelocity(0.0)
            self.ruedaIzquierdaInferior.setVelocity(velocidad)
            self.ruedaIzquierdaSuperior.setVelocity(velocidad)


    self.ruedaDerechaSuperior.setVelocity(0)
    self.ruedaDerechaInferior.setVelocity(0)
    self.ruedaIzquierdaInferior.setVelocity(0)
    self.ruedaIzquierdaSuperior.setVelocity(0)

    self.mapping.update({'type': 'giro', 'value': ang_z})

    return giro
```

Figura 6. *Método "giroDerecha".*

```
def giroIzquierda(self, angulo):
    velocidad = 2.0
    ang_z = 0
    giro = False

    self.robot.step(self.robotTimestep)
    lidar_data = self.lidar.getRangeImage()
    lidar_left = lidar_data[300:375]
    obstaculo, min = self.getObstaculo(lidar_left)

    if(obstaculo==None):
        giro = True
        while ((self.robot.step(self.robotTimestep) != -1)and
               (ang_z<(angulo))): #0.5*np.pi

            gyroZ =self.giroscopio.getValues()[2]
            ang_z=ang_z+(gyroZ*self.robotTimestep*0.001)

            self.ruedaDerechaSuperior.setVelocity(velocidad)
            self.ruedaDerechaInferior.setVelocity(velocidad)
            self.ruedaIzquierdaInferior.setVelocity(0.0)
            self.ruedaIzquierdaSuperior.setVelocity(0.0)

    self.ruedaDerechaSuperior.setVelocity(0)
    self.ruedaDerechaInferior.setVelocity(0)
    self.ruedaIzquierdaInferior.setVelocity(0)
    self.ruedaIzquierdaSuperior.setVelocity(0)

    self.mapping.update({'type': 'giro', 'value': ang_z})

    return giro
```

Figura 7. *Método "giroIzquierda".*

6.2 Comportamiento reactivo.

El comportamiento reactivo es una respuesta o *reacción* inmediata a una percepción dentro del entorno, llámese *estímulo*. En este contexto un estímulo puede ser una entrada sensorial que el robot percibe del entorno. Por otra parte la reacción comprende acciones que el robot realiza son en respuesta al estímulo percibido, y operan en base a reglas simples que mapean directamente las percepciones sensoriales a acciones motoras.

La metodología utilizada para el diseño de comportamientos fue "Diseño de Actividades Asociado a una Situación" (Situation-Based Activity Design), el cual se basa en identificar situaciones específicas en las que un robot puede encontrarse y desarrollar los comportamientos pertinentes. La forma en la que se desarrolla el controlador con esta metodología se puede apreciar en la *figura 8.*

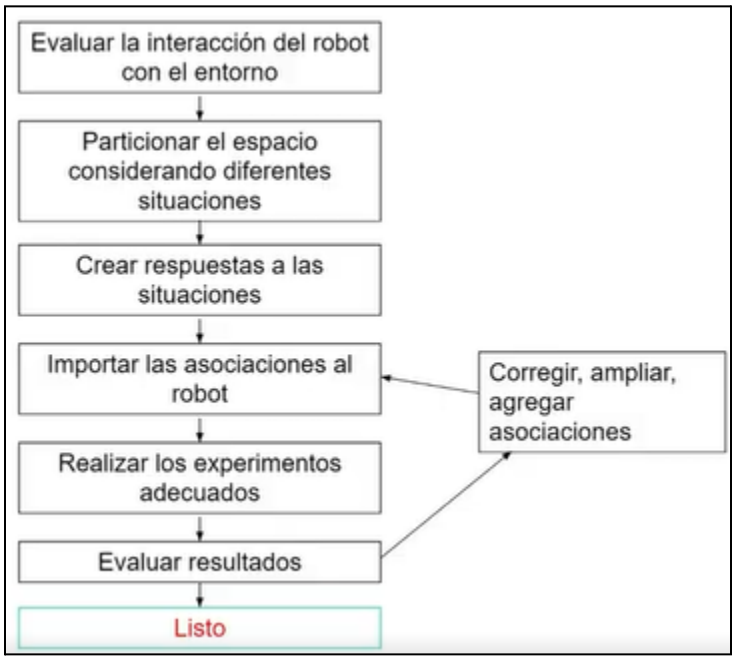

Figura 8. *Diagrama de desarrollo "Guiados por la actividad asociada a una situación".*

Para poder brindarle comportamientos al robot se creó la clase *HROSbotComportamental()* la cual hereda los atributos y métodos de la clase *HROSbot()*.

El agente cuenta con tres comportamientos definidos los cuales se verán influidos por la señal de radio, debido a que el objetivo principal del robot es encontrar al emisor de dicha señal. Por este motivo es que se agrega un atributo el cual almacena la dirección de la última señal encontrada

6.2.1 Ir a la señal.

Este comportamiento reacciona al estímulo de una señal de radio y busca dirigirse hacia ella. Para lograr esto utiliza la antena receptora y en base al último paquete recibido determina la intensidad de la señal, con ella la distancia hacia el emisor y por último la dirección estimada de donde recibió el paquete.

Una vez definida esas variantes, utiliza las acciones predefinidas (explicadas en la sección anterior), girando hacia el emisor y avanzando la distancia calculada.

```
def ir_estimulo(self):
    #Detecta la señal e intenta dirigirse a ella.
    self.robot.step(self.robotTimestep)
    print("-----> Ir_estimulo")
    velocidad = self.speed
    finaliza = False
    giro = False

    if (self.haySeñal()):
        direccion = self.receiver.getEmitterDirection() #1: x; 2: y; 3:z;
        self.actualizarSeñal()
        if (direccion[0]<1):
            if(direccion[1]>0):
                self.robot.step(self.robotTimestep)
                if(self.getObstaculoAIzquierda()==None):
                    giro = self.giroIzquierda(math.atan2(direccion[1], direccion[0]))
            else:
                self.robot.step(self.robotTimestep)
                if(self.getObstaculoADerecha()==None):
                    giro = self.giroDerecha(math.atan2(direccion[1], direccion[0]))

            if(giro):
                self.actualizarOrientación(math.atan2(direccion[1], direccion[0]))
                distancia = self.distanciaSeñal()
                self.vaciarCola()
                finaliza = self.avanzar(distancia,velocidad)
                self.actualizarSeñal()
        self.vaciarCola()
        self.robot.step(self.robotTimestep)

    return finaliza
```

Figura 9. *Método "ir_estimulo".*

Luego de girar el robot almacena la dirección del último paquete recibido. Información que será de utilidad para los demás comportamientos en caso que no se llegue al objetivo.

6.2.2 Evitar Obstaculo.

Al momento en el que el robot se encuentra con un obstáculo en frente, este lo detecta a través del lidar que no solo indica si hay un objeto enfrente a una determinada distancia, sino que brinda información sobre el punto específico en el que la distancia es menor. Cada punto del lidar puede ser convertido a grados y con ellos determinar un ángulo de

giro. El lidar implementado tiene un total de 400 puntos, en donde el punto cero(frente exácto del robot) es el grado cero.

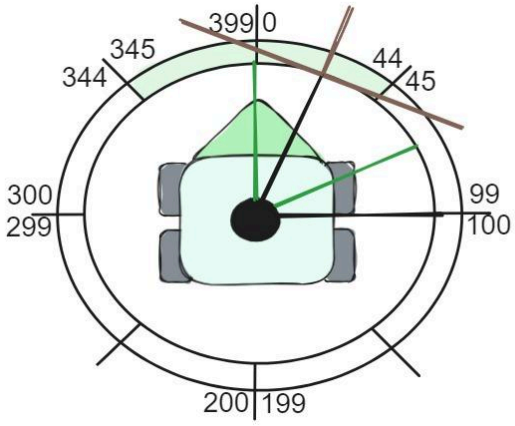

Figura 10. *Representación gráfica de los puntos del lidar y sus grados.*

Con el punto específico en donde la distancia es menor se calcula el ángulo hasta el punto cero.

Para determinar el giro, se observa la dirección del último paquete recibido y se almacena el giro. En caso de no haber encontrado ninguna señal durante el recorrido se optara por girar para el lado contrario a donde se encontró la mínima distancia a un obstáculo.

En el caso que el obstáculo se encuentre en el punto cero del lidar el robot girará 90º hacia donde haya indicado la dirección del paquete. Caso contrarió el robot girará el ángulo calculado entre el punto en donde se encontró la mínima distancia y el punto cero. Este giro lo hará hasta que no se detecte un obstáculo en frente.

Una vez que el giro se concretó, almacena la dirección del último paquete recibido (si es que existe), y avanza hasta que el obstáculo que quedó a un lado ya no esté.

6.2.3 Explorar.

Explorar es un comportamiento fundamental para conocer el entorno. Con esta premisa se codificó un método el cual induce al robot a cumplir este objetivo.

En primer lugar, el robot decide de forma aleatoria si girar o no. En el caso de que se opte por girar, el giro se determinará por la dirección de la última señal encontrada, si no

es posible girar para esa dirección lo intentará para la contraría. El ángulo de giro sólo puede ser de un valor entre 0 y 45 grados.

Ya sea que se concrete o no un giro, el robot siempre intentará avanzar un metro si es que no detecta un obstáculo en frente.

Nuevamente se almacena la dirección del último paquete recibido(si es que existe).

6.3 Aprendizaje.

En búsqueda que el agente tenga la capacidad de tener una representación del entorno en base a la interpretación de lo que percibe, y a su vez una capacidad de aprendizaje sobre las decisiones que debe tomar en cada situación, se codificó la clase *HROSbotInteligente()* que hereda los atributos y métodos de la clase *HROSbotComportamental()*.

Esta nueva clase está basada en la técnica de machine learning *aprendizaje por refuerzo* en post de que el robot pueda tener un aprendizaje supervisado cualitativo y un entendimiento del entorno en base a los estados definidos. Dentro de la clase nos encontramos con los siguientes métodos

- Definir estado: método que define el estado en el que se encuentra el robot en base a la distancia con un objeto y a si hay o no una señal de radio.
- Definir siguiente acción: método que elige una acción correspondiente que debe tomar el agente en base al estado en el que se encuentra. Estas acciones pueden ser cualquiera de los tres comportamientos que contiene la clase padre.
- Actualizar políticas: actualiza la tabla de políticas Q-Table.

Para que el agente pueda ser entrenado se desarrolló la clase *EntornoEntrenamiento()* cuyo objetivo es brindar una forma de entrenamiento que sea compatible con el aprendizaje por refuerzo.

Una vez entrenado el robot podrá tener la capacidad de definir el estado en el que se encuentra y tomar una decisión sobre qué comportamiento tomar. Esta forma de actuar es un tipo de *coordinación híbrida*, ya que trae consigo dos características fundamentales:

- **Componente Reactiva**: Cuando se selecciona un comportamiento este actúa de manera reactiva ante los estímulos del entorno.
- **Componente Deliberativo**: El algoritmo de aprendizaje proporciona un nivel de planificación al seleccionar el comportamiento adecuado basado en experiencia previa y buscando maximizar la recompensa futura.

6.4 Navegación

Para lograr la generación del mapa de la trayectoria, se implementó la clase *HROSbotMapping()*. La misma contiene un atributo principal *map*, integrado por una lista ordenada de las acciones que realizó el robot desde el punto de inicio. Cada elemento conserva información adicional sobre dicho movimiento, por ejemplo, si se trata de una acción de tipo "avance", se conserva la cantidad de metros que efectivamente avanzó el robot. Lo mismo con las acciones de retroceso y de giro a izquierda y a derecha.

Como resultado, al reconstruir la trayectoria una vez que el robot ha llegado a su destino, se obtiene un mapa calculado por odometría cuya precisión es lo suficientemente buena para la utilidad que tiene, que es únicamente la de registro.

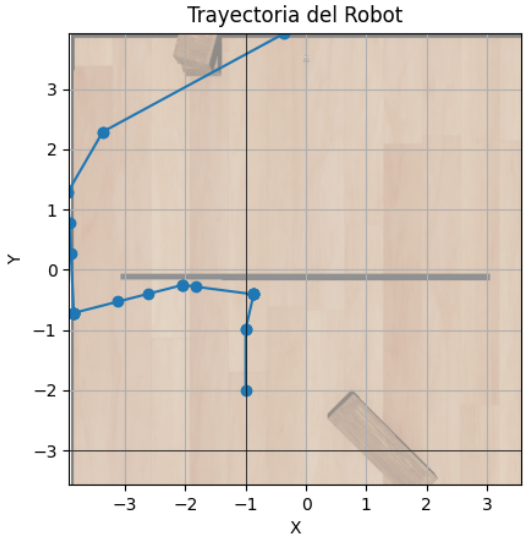

Figura 11. *Mapa generado.*

Cabe aclarar que, por supuesto, el mapa odométrico no se encuentra libre de error. La diferencia entre la distancia o el ángulo de giro estimado en contraposición al real, si bien suele ser mínima, a la larga puede afectar la exactitud del recorrido aproximado en el mapa. La imágen de fondo contribuye a esto por no ajustarse a la perfección a la escala, sin embargo su presencia mejora la visualización más de lo que la perjudica.

7. Tipos de controladores.

El controlador es el encargado de determinar el comportamiento del robot dentro de las diversas situaciones, dentro del entorno, en la que puede encontrarse.

Según el libro de *Stefano Noli* [1] existen tres tipos de controladores:

1. The behavior-based approach o Enfoque basado en comportamiento.
2. The adaptive approach o Enfoque adaptativo
3. The deliberative approach o Enfoque deliberativo.

dentro de los cuales nos enfocaremos en los dos primeros, pudiendo implementarlos y compararlos en base a sus rendimientos.

7.1 Behavior-based robot

Los robots basados en el comportamiento no dependen de un modelo interno del mundo exterior y no simulan mentalmente el efecto de sus acciones antes de actuar en el entorno. Determinan cómo actuar en función de las observaciones actuales[1]. Durante su recorrido extraen información de lo que logran percibir a través de sus dispositivos.

Para el diseño de un robot basado en comportamientos se utilizó el concepto de *Vehículo de Braitenberg*, desarrollando un algoritmo simple el cual sigue el siguiente diagrama de flujo:

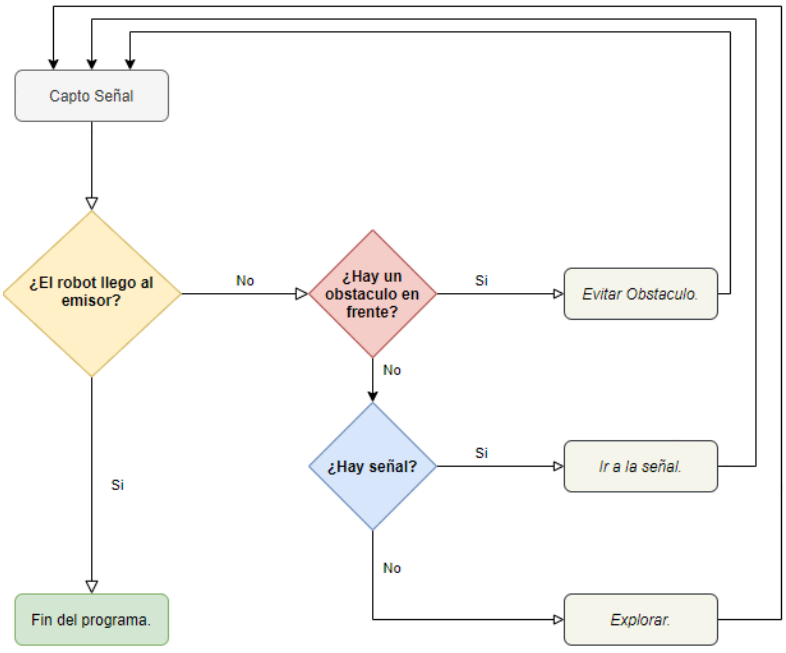

Figura 12. *Diagrama de flujo.*

[1] *Behavioral and Cognitive Robotics: An Adaptive Perspective*, 2021, pag. 12

Visto en código:

```
while(robot.step(timestep) != -1)and(not llegue):
    print("------Comportamiento ",i,"----------")
    i+=1
    if(rosbotComp.getObstaculoAlFrente() != None):
        print("Evitar Obstaculo")
        rosbotComp.evitarObstaculoGuiado()
    elif(rosbotComp.get_receiver() > 0):
        print("Ir Estimulo")
        rosbotComp.ir_estimulo()
    else:
        print("Explorar")
        rosbotComp.explorar()

    llegue =  rosbotComp.estimuloEncontrado(0.1)

rosbotComp.displayMapa()
```

Figura 13. *Controlador basado en comportamiento.*

Esta forma de actuar del robot responde al tipo de *coordinación competitiva* en donde un comportamiento será ejecutado hasta que se determine un cambio en el entorno que active a otro comportamiento.

7.2 Adaptive approach robot

El enfoque adaptativo pretende crear robots capaces de desarrollar las habilidades conductuales y cognitivas necesarias para realizar una tarea de forma autónoma, mientras interactúan con su entorno, a través de procesos evolutivos y/o de aprendizaje[2]. Durante este proceso no debe haber interrupción o ayuda humana.

Para el diseño de un robot adaptativo se utilizó un tipo de algoritmo de *aprendizaje por refuerzo*: **Q-Learning**.

El Q-Learning es un algoritmo de aprendizaje automático utilizado para resolver tareas que pueden ser enfocadas como procesos de decisión Markovianos y cuyo objetivo es encontrar la política óptima y maximizar los valores de las funciones anteriores. Dentro de esta técnica existen dos variantes según el tipo de aprendizaje. Por un lado, se encuentra el *aprendizaje con modelo* en donde el entorno es conocido, junto con las probabilidades de acción y sus recompensas. Por otra parte, el *aprendizaje sin modelo* que es un tipo de aprendizaje en donde el agente no tiene un modelo explícito del entorno, sino que aprende directamente interaccionando con él.

[2] *Behavioral and Cognitive Robotics: An Adaptive Perspective*, 2021, pag. 15

Para este trabajo se utilizó un aprendizaje sin modelo basado en el método *on-policy Sarsa*[3] cuyo objetivo es el aprendizaje en base a la interacción real del agente con el entorno actualizando las políticas según las acciones que se toman. El funcionamiento consiste en:

1. Determinar el estado actual.
2. Observar la mejor acción a tomar en la tabla de políticas.
3. Ejecutar la acción.
4. Obtener la recompensa.
5. Repetir paso 1, 2 y 3.
6. Actualizar políticas.

Otros dos factores a tener en cuenta son el *learning rate* y el *factor de descuento*. El learning rate o tasa de aprendizaje es un parámetro que determina en qué medida el agente ajusta las estimaciones de los valores de la tabla de políticas en respuesta a las nuevas experiencias, generalmente este valor está entre cero y uno indicando una actualización parcial que balancea la nueva información y la estimación previa. Por otra parte, el factor de descuento controla la importancia de los refuerzos a largo plazo, cuanto mayor el valor más importancia se le da a las recompensas futuras.

Para actualizar la tabla de políticas(o *Q-Table*) se utiliza la siguiente fórmula:

$$Q(S,A) = Q(S,A) + \alpha \cdot (r + (\gamma \cdot Q(S',A') - Q(S,A)))$$

donde

- $Q(S,A)$: valor del estado actual y la acción tomada.
- α : learning rate.
- γ : factor de descuento.
- $Q(S',A')$: valor del estado siguiente y la acción tomada.

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state},\cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ϵ-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Figura 14. *Pseudocódigo.* (Sutton & Barto, 1998, pag 155).

[3] Sutton & Barto, 1998, pag. 168

Un factor importante que viene arraigado con el método Q-learning es el concepto de *exploración-explotación*. Cuando el agente se encuentra en un estado particular, la tabla de políticas indica cual es la mejor acción que se puede tomar para llegar al éxito, si el agente toma esta acción en base a las políticas se determina *explotación*. Si por otra parte el agente toma la decisión de tomar otra acción diferente a la que indica las políticas se determina *exploración*.
La exploración es muy útil en pos de buscar un camino o una solución óptima que a priori es desconocida por las políticas. La combinación de ambos conceptos a la hora de aprender es fundamental, y la forma en que se determina que acción tomar puede ser abordada por varios métodos.

En las secciones siguientes se especifica cómo fue abordado este método de aprendizaje en el trabajo.

7.2.1 Estados.

La definición de estados viene dada por las capacidades del robot (*husarion rosbot*) para percibir el ambiente. Por un lado, el robot se vale del LIDAR y los sensores infrarrojos frontales para poder medir la distancia a un objeto, a su vez utilizará la antena para captar señales.

Se definieron tres situaciones posibles en el que el robot puede encontrarse dentro del entorno.

1. Sin obstaculo: no se detecta un obstáculo en los próximos dos metros frente al robot. Distancia máxima de los sensores frontales.
2. Obstáculo lejano: obstáculo detectado por los sensores frontales a una distancia superior a treinta centímetros.
3. Obstáculo cercano: obstáculo detectado por el lidar a treinta centímetros o menos.

Dentro de estas situaciones se distingue si recibe o no señal. En el caso de que haya señal se verifica

A. *Señal lejana*: distancia del robot al emisor mayor a tres metros.
B. *Señal cercana*: distancia del robot al emisor menor a tres metros.

Si se detecta una señal y el agente está en la situación uno o dos se verifica si hay un obstáculo a la derecha o a la izquierda a una distancia de treinta centímetros.

En total hay **trece estados posibles** en los que se puede encontrar el robot.

```
def estadoActual(self):
    if((frs>=self.limiteSensor)and(fls>=self.limiteSensor)): #No Hay Obstaculo
        if((Queque<=0)): #No Hay señal
            indice = 0
        else:   #Hay señal
            if(obstaculo):
                if(distS>distacia): #Señal Lejana
                    indice = 1
                else:
                    indice = 2 #Señal Cercana
            else:
                if(distS>distacia): #Señal Lejana
                    indice = 3
                else:
                    indice = 4 #Señal Cercana
elif(((frs>self.minDistancia)or(fls>self.minDistancia))and(of==None)): #Obstaculo lejos
    if((Queque<=0)): #No Hay señal
        indice = 5
    else:   #Hay señal
        if(obstaculo):
            if(distS>distacia): #Señal Lejana
                indice = 6
            else:
                indice = 7 #Señal Cercana
        else:
            if(distS>distacia): #Señal Lejana
                indice = 8
            else:
                indice = 9 #Señal Cercana
elif(of!=None): #Obstaculo Cerca
    if((Queque<=0)): #No Hay señal
        indice = 10
    else:   #Hay señal
        if(distS>3): #Señal Lejana
            indice = 11
        else:
            indice = 12 #Señal Cercana

return indice
```

Figura 15. *Método de definición de estado.*

7.2.2 Acciones.

Las acciones que puede tomar el robot son aquellos comportamientos mencionados en la *sección 6.2* : *Ir al estímulo; Evitar obstáculo;* y *Explorar*.

Para elegir qué acción a realizar en cada situación se utiliza una política ϵ-*greedy* en donde se toma un valor entre cero y uno de forma aleatoria con una distribución uniforme continua, por lo que cada valor del intervalo tiene la misma probabilidad de salir. En caso de que el valor del intervalo sea mayor al *epsilon* ϵ se realizará la acción indicada por la *Q-table*, caso contrario se tomará una acción al azar que puede, o no, coincidir con la de la tabla de políticas.

El valor de ϵ que se utilizó es de: $\epsilon = 0.2$.

```python
def siguienteAccion(self, estadoActual):
    explorar = np.random.uniform()
    sigAccion = 0

    #Determino si la siguiente acción es explorar.
    if(explorar<=self.prob_exploracion):
        print("-> Comportamiento de Exploración <-")
        sigAccion = np.random.randint(self.cantidadAcciones)
    else:
        sigAccion = np.argmax(self.qLearning[:,estadoActual])

    return sigAccion
```

Figura 16. *Método que define la siguiente acción.*

7.2.3 Q-table.

La tabla de políticas *Q-table* se define como una matriz de tres filas (cantidad de acciones) por trece (estados posibles).

Una *Q-table* puede comenzar tanto con valores distintos de cero como con valores iguales a cero, pero hay que tener ciertas consideraciones. Si se inicializa con ceros garantiza que no haya un sesgo para la elección de la siguiente acción en el entrenamiento, pero hace que tenga una exploración lenta. Mientras que si se inicializa con valores distintos de cero hay que tener cuidado de que dichos valores no sean muy grandes, debido a que pueden llevar a un sesgo significativo.

En el presente trabajo se optó por inicializar la tabla de políticas con diferentes valores entre [0, 0.05] de forma aleatoria con una distribución uniforme continua. Los valores elegidos son suficientemente bajos como para que no haya un sesgo significativo y al ser positivos mayores o iguales a cero ayuda a una exploración temprana en el entrenamiento.

7.2.4 Entrenamiento.

El robot es entrenado gracias a la clase *Entorno de Entrenamiento* presentada anteriormente, en donde se define una cantidad de épocas específica junto con una cantidad determinada de pasos. Por cada época el robot leerá los valores de sus dispositivos y comenzará a entrenar, pudiendo ejecutar una cantidad de acciones equivalentes a la cantidad de pasos. Por cada paso el robot define en qué estado se encuentra y qué acción debe tomar(en base a lo visto en la sección previa), para luego actualizar la tabla de políticas. La época termina cuando el agente llega al estímulo o cuando llega al límite de pasos.

Cuando termina una época se analiza la posición del entorno en la que terminó el robot, si no llegó al estímulo dicha posición se almacena. En la próxima época el robot podrá aparecer en la posición inicial de entrenamiento o en alguna de las posiciones almacenadas.

El entrenamiento finaliza cuando se completaron todas las épocas.

8. Resultados

En esta sección se presentarán los resultados obtenidos por los dos tipos de robots implementados (*Behavior-based robot* y *Adaptive approach robot*), considerando:

1. Desempeño y efectividad a la hora de encontrar el estímulo.
2. Situaciones en las que el algoritmo no funcione correctamente o como se esperaría.
3. Costo de desarrollo.

8.1 Desempeño.

En esta sección se evaluará el desempeño de cada robot para cumplir el objetivo (encontrar al emisor de la señal de radio). Para ello se ejecutará un total de cinco veces la simulación en donde cada robot iniciará desde una posición en la cual no tendrá ningún obstáculo en frente y no recibirá una señal del emisor.

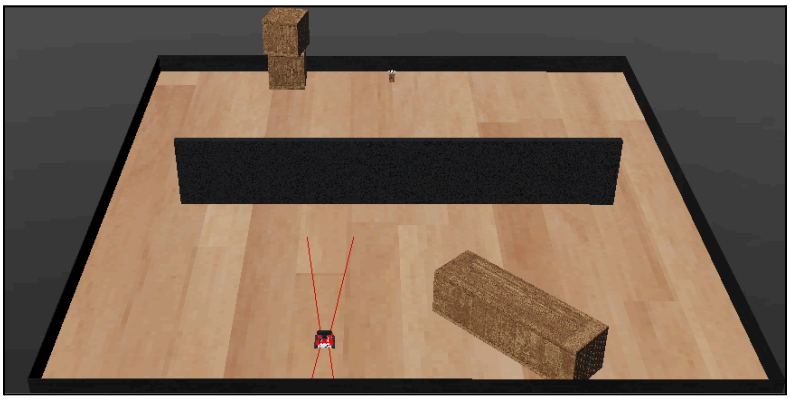

Figura 17. *Posición inicial.*

Por otro lado, se planteará a cada robot una situación muy difícil e inusual, en donde no hay ninguna señal y hay dos obstáculos a una distancia media.

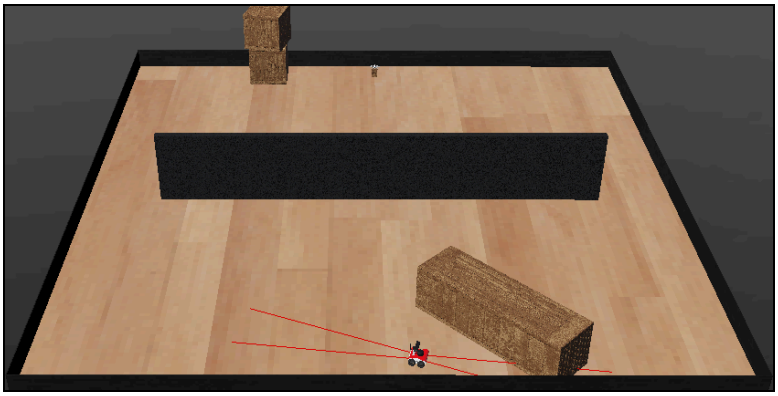

Figura 18. *Posición inusual.*

En la siguiente tabla se visualizará el desempeño de cada robot en base a las simulaciones, en donde se registraron; cantidad de simulaciones hasta llegar cinco veces al objetivo; cantidad de comportamientos ejecutados; y cantidad de fallos(simulaciones en las que el robot no pudo llegar al objetivo).

		Adaptive approach robot	***Behavior-based robot***
Cantidad de comportamientos ejecutados	*Primer simulación exitosa.*	21	12
	Segunda simulación exitosa.	14	11
	Tercera simulación exitosa.	18	10
	Cuarta simulación exitosa.	11	9
	Quinta simulación exitosa.	12	11
Cantidad	**Fallos**	0	3
	Simulaciones	5	8

Como se aprecia en la tabla anterior, en promedio el robot con enfoque adaptativo ejecutó aproximadamente 15 comportamientos antes de llegar al objetivo, mientras que el robot basado en comportamientos ejecutó en promedio 10 comportamientos aproximadamente.

Si bien es cierto que el agente basado en comportamientos puede llegar a ejecutar una cantidad igual o menor de comportamientos que el otro agente, hay que tener en cuenta que hubo tres simulaciones en la que no pudo completar la simulación.

En las siguientes figuras se aprecia el recorrido general que tuvieron ambos robots.

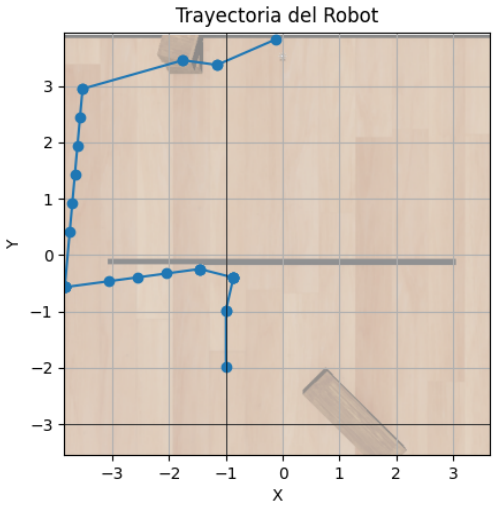

Figura 19. *Recorrido del robot comportamental.*

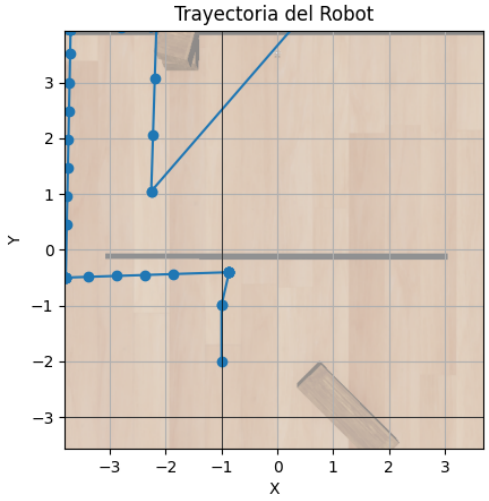

Figura 20. *Recorrido del robot adaptativo.*

Para el segundo escenario planteado se realizó una única simulación por cada agente, cuyos resultados fueron:

	Adaptive approach robot	***Behavior-based robot***
Pasos para evitar los obstáculos	21	>50
Pasos para ir al objetivo	50	-

En esta situación ambos robots tuvieron dificultades para evitar los obstáculos. Pero mientras que el agente con enfoque adaptativo tuvo que ejecutar 21 comportamientos para salir, el otro agente no pudo completar la prueba.

8.2 Fallos.

Como se pudo apreciar en la sección anterior no todas las simulaciones fueron completadas por ambos robots. Mientras el robot con enfoque adaptativo logró completar cada una de las simulaciones, el robot basado en comportamientos no pudo hacerlo.

Esto se debe a una situación particular en la que el agente no percibe obstáculo alguno y decide ir hacia la señal pero no puede girar, lo que genera que no pueda cumplir el objetivo.

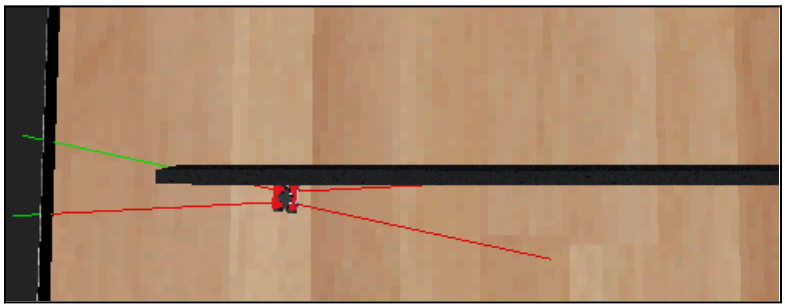

Figura 21. *Situación de fallo.*

La presente situación no tiene efecto sobre el robot de enfoque adaptativo debido a la representación interna del entorno manifestada en la definición de estados, en donde se contempla el hecho de no haber un obstáculo enfrente pero si a un lado. A su vez si hubiera un escenario en el que no pueda avanzar, la propia toma de decisiones tiene un factor de exploración que puede ayudar al agente a salir de esa situación.

8.3 Costo de desarrollo.

El costo de desarrollo y el costo computacional es primordial para la comparación de ambos robots. Como ambos agentes tienen los mismos comportamientos se analizará los algoritmos propios del funcionamiento de cada controlador.

El agente basado en comportamientos tiene un algoritmo simple, el cual solo requiere observar los datos recibidos de los dispositivos del propio robot y actuar en consecuencia. Este tipo de funcionamiento no requiere memoria, ni un entrenamiento previo por lo que no es muy costoso.

Por otra parte el agente con enfoque adaptativo utiliza un algoritmo de aprendizaje por lo cual necesita un tipo de entrenamiento específico lo que genera un costo computacional y de tiempo real alto. A su vez, requiere un algoritmo para definir el estado en que se encuentra y otro para tomar la decisión de qué acción tomar.

9. Conclusiones

Al contrastar ambos modelos planteados, logramos concluir en los siguientes puntos:

- El enfoque basado en Q-Learning posee la ventaja de posibilitar el aprendizaje, la optimización y adaptación por parte del robot, lo que deriva en una mayor flexibilidad en la navegación. Esto es algo particularmente útil en entornos dinámicos.
- Otro punto a favor de dicho enfoque es que facilita considerablemente la escalabilidad del agente al agregar nuevos comportamientos, ya que - si bien habría que programar este comportamiento adicional y reiniciar el entrenamiento - no es preciso modificar la lógica del controlador, resultando más ajustable y conveniente en ese aspecto.
- En cuanto al rendimiento, el agente adaptativo posee la cualidad de ser más lento al momento de lograr su objetivo, superando en cantidad de pasos al otro agente en todas las pruebas realizadas con éxito. Sin embargo, demuestra una mayor eficacia al haber logrado su objetivo en todos los casos, a diferencia del otro agente.
- Por otro lado, la desventaja de pasar de un modelo basado en los vehículos de Braitenberg, caracterizados por su simpleza, a uno basado en Q-Learning es que este último presenta un incremento en la complejidad y en los requerimientos de potencia computacional, así como un tiempo adicional para el entrenamiento del agente.

Determinar cuál de los dos agentes es mejor es una tarea difícil si se realiza de manera aislada, ya que el contexto, el entorno, los recursos y los objetivos son factores cruciales que influyen en el rendimiento de cada modelo. Es fundamental considerar estas variables para tomar una decisión efectiva, por lo que la elección del agente más adecuado dependerá de las condiciones específicas y de las prioridades del sistema en cuestión.

En el marco de este trabajo y observando el desempeño de ambos robots, llegamos a la conclusión de que el robot con un controlador con enfoque adaptativo cumple de mejor manera el objetivo, pudiendo tomar decisiones acertadas en cada simulación.

10. Referencias

[1] . *Behavioral and Cognitive Robotics: An Adaptive Perspective*. Stefano Nolfi. (2021).

[2]. *Reinforcement Learning: An Introduction.* Sutton, R. S., & Barto, A. G. (1998).

[3]. *Aprendizaje por Refuerzo Elementos básicos y algoritmos*. Lorién Lascorz Lozano. Trabajo de fin de grado en Matemáticas. Universidad de Zaragoza.