Reporte Proyecto Final

Mildred Limón Santamaría
177229

Renata Monsalve Rubí
176371

***Abstract*—El documento detalla la implementación de un algoritmo que automatiza el desplazamiento de una tortuga en ROS desde un punto inicial aleatorio hasta un punto objetivo determinado. Considera la posibilidad de obstáculos y define cómo debe ser sorteado. A pesar de que sólo se desarrolla a un nivel de simulación de turtlesim, el código final resultante puede ser aplicado a la automatización de un robot móvil que se desplace en un ambiente real.**

I. Introduction

Al trabajar con robots se busca que estos puedan realizar tareas de manipulación en entornos humanos. Existen múltiples sistemas que ayudan a los desarrolladores de software a crear aplicaciones de robots. Un sistema base que proporciona puntos de partida y estructura básica para entender la robótica es el sistema ROS, antes llamado switchyard . ROS se describe como un entorno de trabajo que cuenta con una variedad de herramientas, librerías y paquetes que busca la creación de software complejo para tener robots robustos y con comportamientos variados. En este proyecto final se buscó tener el mayor acercamiento posible con el desarrollo de aplicaciones de robots. A través del simulador 2D para robots diferenciales en ROS: turtelsim pudimos tener un primer acercamiento con la robótica. Turtelsim es un simulador que permite, de forma simple, aprender los conceptos básicos de ROS. El simulador consiste en una ventana gráfica que muestra un robot en forma de tortuga (turtlebots). De esta manera realizamos tres tareas principales con el turtlebot, a saber, un recorrido rectangular, uno circular y uno coordinado con otros turtlebots.

II. Marco Teórico

Dado que todo el sistema funciona sobre la plataforma ROS, turtlesim y la suscripción y publicación de tópicos, no se ahondará en la teoría detallada en el documento anterior. Esta puede ser vista en el siguiente reporte https://github.com/monrubi/PM-Practica-1/blob/master/Practica5/Practica5.pdf

Sin embargo, dada la implementación que se realizará de un algoritmo de automatización similar a las búsquedas (llegar de un punto A a un B dados), es importante que se platee la diferencia entre un algoritmo de fuerza bruta y un heurístico. Por un lado, el algoritmo de fuerza bruta es el algoritmo más simple posible. Consiste en probar todas las posibles combinaciones hasta lograr un resultado positivo. Su coste de ejecución es proporcional al número de soluciones candidatas, el cual es exponencialmente al tamaño del problema. Por ello, el algoritmo se usa habitualmente cuando el número de soluciones candidatas no es elevado, o bien cuando este puede reducirse previamente usando algún otro método heurístico. Para poder utilizar el algoritmos de fuerza bruta a un tipo específico de problem, se deben implementar las funciones primero, siguiente, válido y mostrar. La principal desventaja del método de fuerza bruta es que, para la mayoría de problemas reales, el número de soluciones candidatas es prohibitivamente elevado. Por otro lado, los algoritmos heurísticos es una técnica diseñada para resolver problemas más rápidamente cuando los métodos clásicos son demasiados lentos, o para encontrar una solución aproximada cuando los métodos clásicos no encuentran una solución exacta. Esto se logra mediante el intercambio de la optimización, integridad, precisión o precisión de la velocidad. En cierto modo, puede considerarse un atajo. Una función heurística clasifica las alternativas en los algoritmos de búsqueda en cada paso de ramificación en función de la información disponible para decidir qué rama seguir. Es posible que la solución dada por la heurística no sea la más óptima de las soluciones, pero intenta aproximarse a la solución exacta. Sin embargo, sigue siendo valiosa porque encontrarlo no requiere un tiempo largo. La heurística subyace en todo el campo de la Inteligencia Artificial y la simulación por computadora del pensamiento, ya que pueden usarse en situaciones en las que no se conocen algoritmos.

III. Desarrollo

El proyecto retoma de la práctica 5. Por ello, de forma previa ya se contaba con una máquina virtual que nos permite usar Linux en su versión Ubuntu 18, con Ros melodic funcionando y la librería dependencia del turtlesim instalada. En esta práctica ya se había hecho entrega de la simulación de la tortuga realizando un círculo y un cuadrado.

Con estas bases se debían desarrollar ahora tres etapas para el proyecto:

La trayectoria del cuadrado esta vez incluía requerimientos de dimensiones y sentido: debe poder realizarse tanto en el sentido del reloj con a contra sentido. El código entregado para la práctica número 5 ya nos permitía realizar el cuadrado, el código fue simplemente adaptado para que cumpliera los requerimientos del proyecto final: tener nueve unidades de cada lado.

A. Primera Parte

En la primera parte se realizó el código para que el turtlebot siguiera una trayectoria cuadrada en dirección horario y antihorario. El desarrollo se encuentra en la práctica número 5 del laboratorio de Principios de Mecatrónica.

B. Segunda Parte

En la segunda parte se realizó el circulo. El código generado en la práctica previa ya nos permitía hacer un cŕiculo con radio variable; se inicia ubicando a la tortuga en el (5,5,0), ya que permitía más facilmente posicionar una coordenada inicial cercana al centro. El proyecto recomendaba que se use (5.5,5.5,0), sin embargo, el diámetro requerido para el círculo podía ser completado en las coordenadas seleccionadas en la práctica, por ello no fue necesario recentrar el círculo y modificar el código. Otro requerimiento marcaba que debe iniciarse en el ángulo 0 y poder realizar un círculo de diámetro 9 (por tanto 4.5 de radio), en cualquiera de los dos sentidos. Se ha anexado un script para cada sentido, para evitar que requiera modificar el código. Una mejora posible a esta etapa sería recibir el sentido del círculo y el radio como parámetro para que sea controlado por el usuario desde la línea de comando (en vez de pre definirse en el código). Uno de los aciertos del código es que la tortuga avanza desde el centro del círculo hasta el radio dado y se orienta hacia el sentido en el que se va a desplazar, por lo que esto no necesita ser manejado por el usuario.

Vale la pena rescatar del reporte previo, que para realizar el circulo se calculan seno y coseno de un ángulo que tiene un incremento fijo en cada iteración. Posteriormente se somete a ecuaciones de escalamiento y traslación (para mover el centro del círculo al punto (5,5) y darle el tamaño del radio solicitado), así, se evitó el uso de funciones trigonométricas que tienen indeterminaciones (como la tangente), ya que coseno y seno tienen un rango acotado en todo su dominio. El algoritmo parte la circunferencia total entre el número de grados que desea que avance cada iteración (entre menor el ángulo de incremento, más fina es la curvatura del círculo). Para no modificar esto, lo único que se requirió al invertir el sentido del traslado es multiplicar por -1 el valor en Y, de ese modo inicia en valores menores a 0 y finaliza en el lado positivo del eje.

C. Tercera Parte

La tercer, y última, etapa del proyecto, requiere de la interacción de las tortugas con otras. Cada equipo cuenta con una tortuga propia, son posicionadas de forma aleatoria en una línea de forma contigua entre ellas, y tienen todas un punto meta al que deben desplazarse sin colisionar entre ellas. El punto de inicio y fin se desconocen, así como la solución implementada por los otros equipos, la cual genera el accionar de sus tortugas. Por ello, debe pensarse en una estrategia que cubra varios escenarios:

- Tortuga con un avance lento o rápido.
- Tortugas atrapadas en una posición, por lo que deberán evadirse por un método diferente a detenerse y esperar que terminen su avance.
- Tortugas que colisionaron entre ellas y requerirán ser sorteadas por su periferia.
- Tortugas con un avance errático, por lo que calcular su trayectoria considerando su velocidad angular y lineal no aporta un valor heurístico.
- También estaba penalizado permanecer estático por más de 3 segundos, por lo que detenerse al percibir otra tortuga transitando no es una solución válida.
- Detenerse cerca de la posición meta si esta está siendo estorbada por otra tortuga inmóvil.

Para entender la solución debe explicarse primero que existen 2 constantes dentro del código: ka y kv. Son constantes por las que se multiplican la distancia faltante y el ángulo faltante, respectivamente. Es decir, tras obtener la distancia entre el punto meta y el actual de la tortuga, y el ángulo entre la orientación en la que debe estar y en la que se encuentran, se multiplican por una constante por lo que funcionan como factores de amplificación o reducción de estos valores y sirven para. El resultado corresponde, en el caso de Ka, a la velocidad lineal que tendrá X y a la velocidad angular que tendrá Z. Sólamente se tiene velocidad lineal en X debido a que la tortuga se mueve de forma frontal únicamente, no se mueve por sus costados (Y) ni se eleva o hunde (Z) ya que representa un robot posicionado sobre el piso. Para resolver el traslado deben proponerse y probarse distintos valores para estas constantes y analizar su conveniencia ¿es preferible una estrategia donde la tortuga avance rápidamente o es preferible en la estrategia permitir que otras tortugas lleguen primero a su destino para trabajar en un área menos ocupada?.

Con ello se constituyó el marco del algoritmo a generar. Se optó por una solución que primero contempla la posición inicial de la tortuga y posteriormente llama a 1 de 3 algoritmos:

1) Si el punto al que se debe desplazar se encuentra directamente al frente, la tortuga sólo debe caminar de frente. En caso de encontrar un obstáculo puede regresar sobre la marcha 1 unidad y volver a intentarlo.
2) Si el punto al que debe desplazarse se encuentra a su derecha debe intentar desplazarse en una trayectoria escalonada, intercalando avances entre a 90° y 0°. Si se atora, debe intentar el otro ángulo, si realiza 4 intentos, por ejemplo, avanzar a la derecha (obstruido), girar hacia arriba (obstruido), avanzar nuevamente a la derecha (obstruido (avanzar nuevamente hacia arriba (obstruido), se asumirá que la tortuga está en una situación donde el obstáculo está detenido (una tortuga atascada o una colisión) e intentará bajar (270°), para volver a intentar avanzar en el ángulo 0°.
3) Si el punto meta se encuentra a la izquierda del punto de origen, se realiza la versión análoga al punto dos, pero usando 90° y 180° grados. El retroceso se realiza de igual manera.

Partiendo de este análisis se realizó el código en Python. Posteriormente se realizaron pruebas sin otras tortugas, para evaluar si el movimiento de la tortuga era el esperado. Lamentablemente, este código funcionaba de forma lenta e ineficiente. En desplazamientos donde no había obstáculos resulta innecesario recorrer la distancia horizontal y luego la vertical. La suma de esas dos distancias es mayor que la que se requiere para un movimiento diagonal. Y, como último detrimento del código, es una búsqueda que se asemeja

más a un algoritmo de fuerza bruta que a uno de búsqueda heurística. Dada la extensión de la fecha de entrega se optó por hacer una segunda propuesta. No trabaja en una mejora del código previo, sino que plantea un nuevo enfoque tras anexar al análisis las siguientes consideraciones:

1) El código que trabaja en rectas no hace un aprovechamiento de la capacidad de la tortuga de hacer trayectorias curvas.
2) Es preferible no buscar ángulos fijos, sino ir buscando consecutivamente un ángulo, que desvíe la trayectoria lo menos posible, en el que la tortuga pueda evadir el obstáculo, en vez de regresar sobre la marcha perdiendo una sección del tramo perdido.
3) La orientación del movimiento es corregida mientras la tortuga se desplaza por lo que no se necesario que avance en ángulos ortogonales.
4) Conviene que revise si va a sortear el obstáculo por el costado izquierdo o el derecho. Si la otra tortuga se desplaza sobre un incremento en las Y, parecería conveniente sortearla por su costado derecho, sin embargo, si su velocidad es muy baja y por ese costado hay un mayor número de tortugas, se encontrarán menos obstáculos si se sortea por la izquierda. Por otro lado puede que ya esté la mayoría en una posición de Y superior a la propia y/o la otra tortuga se desplace rápidamente, por lo que conviene sortearla por el costado derecho

A grosso modo, el código se escribió adaptando el "go to goal" para incluir un if que determina si avanza a un espacio libre o no. Si está libre avanza de forma normal. En caso de que no esté vacío revisa si se está desplazando a la izquierda o a la derecha para asignar el signo en el que va a rotar. Una vez que se asigna esta nueva orientación se permite que continúe avanzando conscientes de que enderezará la trayectoria conforme avance hacia la meta. Existen 5 parámetros que manipulan el comportamiento de este algoritmo:

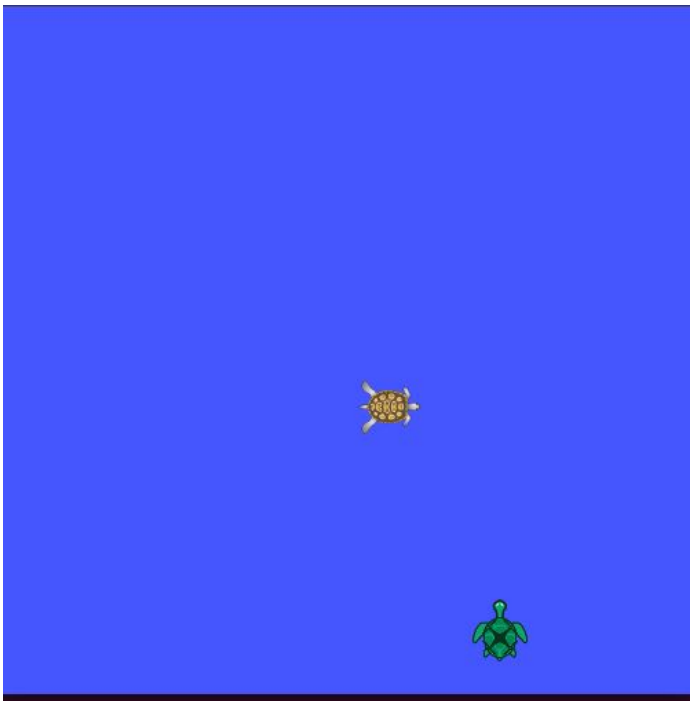
1) El ángulo que sondea la tortuga. Revisa si hay objetos en su proximidad, pero en caso de que sea así sólo se detendrá si se encuentra "frente a la tortuga". Dado que podría revisarse que no haya nada cercano en los primeros 180° que rodean la tortuga, sin embargo, en las pruebas realizadas con un ángulo de visión tan amplio, la tortuga tendía a quedar ciclada (dando vueltas sobre su eje) cuando topaba con tortugas estacionadas que simulaban una colisión. Se optó por cerrar el margen de distancia que la tortuga podía tener al pasar cerca de otra haciendo más angosto este ángulo. Varias pruebas determinaron que, en conjunto con otras variables, el valor que conviene son 50 grados a cada costado, lo que da un ángulo total de 100°.
2) La "sana distancia". Es el radio mínimo de distancia que la tortuga debe tener de separación. Esta distancia está determinada por varios factores. Primero, debe ser mayor que el tamaño de la tortuga. Segundo, debe ser menor que el "cajón" de llegada que es la posición meta y se encuentra entre otras tortugas previamente acomodadas. Tercero, debe contemplar la velocidad a la que se desplaza, es decir, si se revisa que no haya nada a 2 unidades de la tortuga pero se desplaza lentamente, se un margen innecesario que la obliga a girar bajo circunstancias que no lo ameritan; por otro lado si el desplazamiento es muy rápido y se revisa que esté despejada una unidad a la redonda, pero se avanzan dos (debido a la velocidad lineal) puede generar una colisión. Finalmente, esta distancia en conjunto con el ángulo aumenta el área total que se requiere esté libre y puede ser mayor que el cajón de llegada. Tras concluir las pruebas con varios valores, se determinó que a esta variable se le asignaría un 1.4, dado que se planea avanzar lentamente y ese valor de un área pequeña necesaria, pero es suficiente para el tamaño de los avances asignados por la velocidad lineal.
3) El ángulo de corrección. Al topar con un obstáculo se debe virar hasta encontrar una posición donde se pueda avanzar libremente. A diferencia del algoritmo anterior, que es por fuerza bruta, en esta implementación el ángulo de rotación no es una constante. La tortuga gira la cantidad de grados necesarios para librar el camino modificando en la menor medida posible su trayectoria inicial. El factor que debe decidirse es el tamaño del incremento en el que revisa. Si avanza de 5 en 5 grados tendrá una ventaja en movimiento, dado que si la tortuga que obstruye el camino se desplaza se podrá avanzar con una menor desviación. Sin embargo, se debe programar pensando en el peor escenario, en caso de que una tortuga esté varada en una posición, avanzar cinco grados puede resultar muy lento y se expone a ser chocada por otra tortuga cuyo algoritmo no considere que las demás pueden pausar por periodos largos de tiempo. Se asignó, tras realizar pruebas con varios valores, que 30° serán incrementos óptimos, dado que la velocidad angular en la que se orienta la tortuga es alta, este ángulo puede ser avanzado rápidamente y es posible que con únicamente dos iteraciones pueda evadir cualquier bloqueo. Si la tortuga contraria se mueve rápidamente en una única iteración se deberá poder avanzar libremente. Se prevé que el mayor ángulo de desviación sea por tanto 60°.
4) Kv, es el valor de la velocidad lineal, determina la cantidad que se avanzará en el ángulo dado como velocidad angular. Inicialmente se usaba como una constante que multiplica la distancia faltante, de tal manera que se mueve rápidamente en un inicio y disminuye al acercarse a la meta. Sin embargo, desplazamientos amplios generan, sobre la marcha, mayores desviaciones de la ruta original. Al usar velocidades bajas, se puede hacer una mejor revisión del entorno para evitar coaliciones, la desventaja es que al acercarse a la meta avanza en extremo despacio.
5) Ka es la velocidad angular. Determina en qué ángulo se avanzará, es decir, la tortuga no avanza hacia donde está orientada, sino que avanza en el ángulo indicado la cantidad indicada por Kv.

Estas dos últimas variables tienen una dinámica importante, si el ka es alto y el kv es bajo, la tortuga podrá enderezar rápidamente la trayectoria, desviándose menos, pero con ello no consigue evadir el obstáculo. Si el kv es alto y el ka es bajo, la curvatura es mayor, por tanto se evade con mayor margen el obstáculo pero el desvío es mayo y la probabilidad de que encuentre otro obstáculo también. Por tanto, se buscaron valores medio, que permita evadir tortugas estáticas con un pequeño margen de distancia, pero que efectivamente no colisionen. Estos valores se obtuvieron por medio de pruebas, el resultado final nos da una ka de 0.27 y un kv de 0.5. Debido al bajo valor de ka, la tortuga tardaba un tiempo considerable en llegar a la meta, por ello se definió ka como la velocidad linear de forma directa, sin ser multiplicada por la distancia faltante.

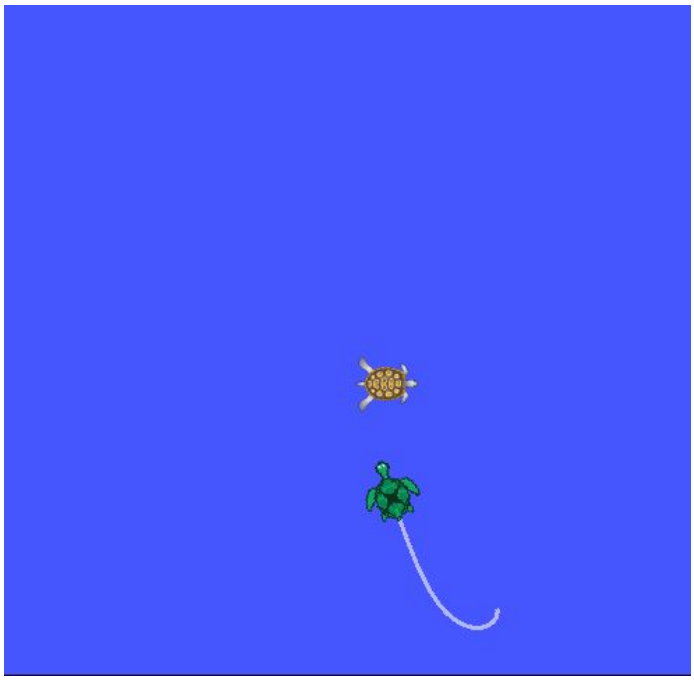
RESULTADOS

Lamentablemente el tiempo invertido en el primer algoritmo que fue desechado mermó el que se podía invertir para hacer pruebas exhaustivas en el segundo. No se pudo generar que dos scripts corrieran simultáneamente y corroborar qué sucedía cuando interactuaba con tortugas en movimiento. Sin embargo, dado que se estaba manejando el caso de las tortugas estáticas como el peor escenario posible, se consideró que si se resolvían estos casos particulares se podía esperar un resultado favorable durante las simulaciones.

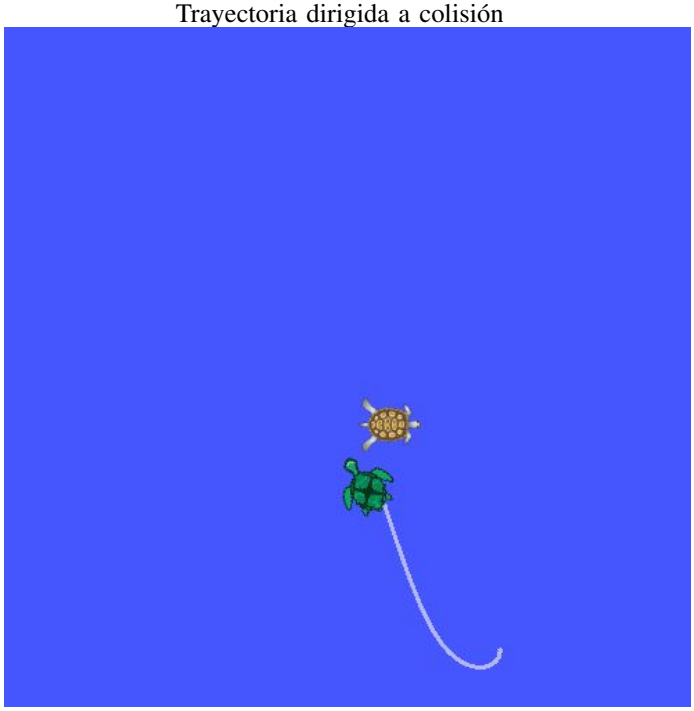
A continuación se muestra la evasión de una tortuga por el lado izquierdo por etapas y su correspondiente lado contrario:



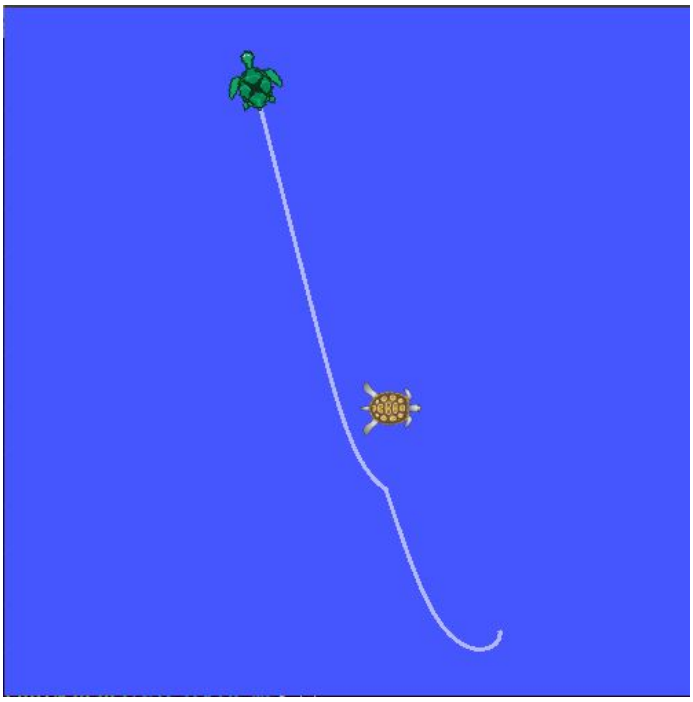
Posicion inicial de las tortugas



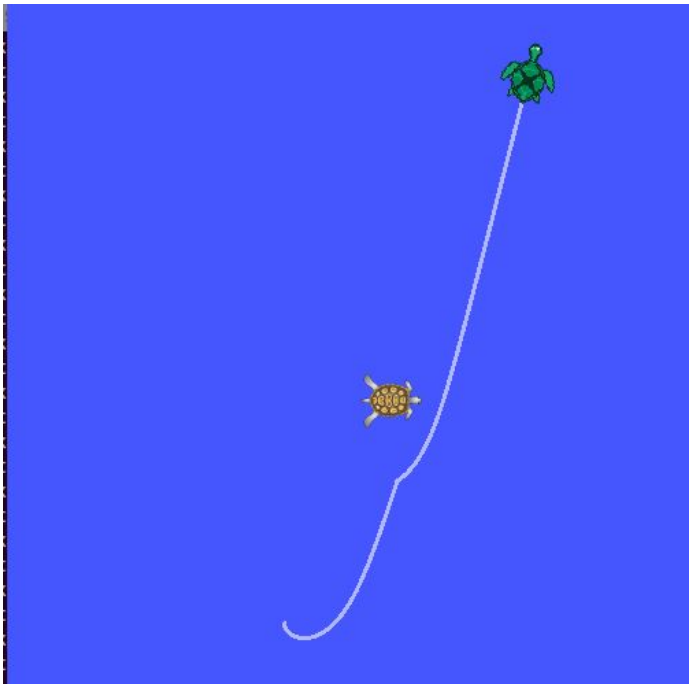
Trayectoria dirigida a colisión



Cambio de orientación de la tortuga



Ruta final con evasión



Análogo por el lado derecho

En las imágenes se percibe que se consiguió el objetivo de virar por abajo para la evasión, asumiendo que la tortuga que se esquiba se dirije hacia arriba, dadas las posiciones de las metas. Las pruebas fueron realizadas usando principalmente el lugar de arranque y destino establecidos en la tabla del grupo. Las posiciones de las tortugas estáticas varían pero eran colocadas en puntos que eran necesariamente parte de las trayectorias de la tortuga 5, la que nos fue asignada. Notamos que en ángulos donde las restas daban valores muy cercanos a cero la tortuga se ciclaba girando sobre su propio eje. La mayoría de estos errores se solucionaron agregando cláusulas de if que forzaban a ángulos que no cumplían con los criterios necesarios para funcionar, a ser valores por defecto funcionales.

Sin embargo, es importante reconocer que no se abarcaron todos los posibles escenarios de fracaso del algoritmo y que el código requiere ser más robusto con cláusulas que contemplen estos fallos y los resuelvan.

El código final puede ser evaluado aquí: https://github.com/monrubi/PM-Practica-1/blob/master/Proyecto/ros_p5.py Los resultados no son suficientemente concluyentes, pero el algoritmo resulta más eficiente y funcional que el previo, se esperan resultados positivos durante la simulación.

Conclusiones

Las conclusiones que se presentan a continuación consideran lo observado durante las pruebas y notas tomadas durante la simulación con todos los equipos. El resultado final es una implementación de poco código, en comparación con los resultados de otros equipos. Ello no implica que sea necesariamente más eficiente, pero las variables con las que se juegan y que se deben mantener bajo control son menos, ello vuelve al código rápidamente adaptable, permite notar los errores de forma más rápida y requiere un menor nivel de procesamiento. La lenta velocidad de avance permitió, efectivamente, chocar menos, hacía una lectura más fina de las distancias por lo que se conseguía llegar al punto meta sin mucho margen de diferencia. El lento avance puso a la tortuga, como se había previsto, en un mayor riesgo de colisión por parte de otras, sin embargo, los algoritmos implementados por los equipos cercanos a nuestra posición inicial eran suficientemente eficientes (o rápidamente ajustables), por lo que si se hubieran construido carros en el laboratorio, el nuestro no habría sufrido mayor percance. A pesar de que se trabajó considerando a las tortugas estáticas como el peor escenario (ya que ahí no hay un trabajo conjunto de evasión sino un obstáculo fijo), esta escena no fue frecuente. Una lección importante, a pesar de que no aporta nada el código ni el aprendizaje de la automatización, es que la forma en la que Windows maneja sus archivos de texto es diferente a la de Linux. El haber guardado el código en un archivo de drive y haberlo subido a Github desde el lado windows de la máquina, casi nos cuesta el total de puntos de nuestro proyecto y el de otro equipo. Es importante como ingenieros conocer este tipo de deficiencias, ya que sería mala práctica de la profesión depender de un único sistema operativo y no poder dominar los 3 más importantes del mercado. En especial Linux, dadas todas las herramientas que ofrece al ser open source y la única plataforma en la que se puede instalar ROS, el cual resulta súmamente útil como centro de conecciones entre escuchas y generadores de información. Tiene la capacidad de generar robots con un alto grado de complejidad y manipulación de información, la cual puede ser alimentada en tiempo real de forma inalámbrica .

Las pruebas realizadas previamente nos dejaron un arreglo de valores que se esperaba fueran los óptimos, pero ellas fueron realizadas con un código que desconocía la lógica de los otros equipos. Esta limitante implicaba que nuestra tortuga se enfrentaría a comportamientos no previsibles y esa fue la realidad observada durante la simulación: se generaron errores con los que cada equipo no se había enfrentado antes. Ahora, se pueden implementar mejoras al código o la heurística para mejorar el proyecto.

REFERENCES

[1] Bently, J.L. (1982). *Writing Efficient Programs*. Prentice-Hall. Recuperado el 20 de mayo de 2020 de https://archive.org/details/writingefficient00bent/

[2] JayEss. (2019). *Moving in a Straight Line*. ROS.org. Recuperado el 8 de mayo de 2020 de http://wiki.ros.org/turtlesim/Tutorials/Moving%20in%20a%20Straight%20Line/.

[3] Rodríguez, E.A. (2018). *Algoritmo de fuerza bruta*. CINVESTAV Tamaulipas. Recuperado el 20 de mayo de 2020 de https://www.tamps.cinvestav.mx/~ertello/algorithms/sesion06.pdf/

[4] Willow Garage. (2005) . *ROS*. Willow Garage. Recuperado el 20 de mayo de 2020 de http://www.willowgarage.com/pages/software/ros-platform/

[5] CleberCoutoFilho. (2016). *Tutorials Using Turtlesim*. ROS.org. Recuperado el 8 de mayo de 2020 de http://wiki.ros.org/turtlesim/Tutorials/

[6] Beuchot, M. (1999). *Heurísitca y Hermenéutica*. Aprender A Aprender (UNAM). Recuperado el 20 de mayo de 2020 de https://books.google.com.mx/books?id=QZ9Ics0JoWIC&pg=PP1&redir_esc=y#v=onepage&q&f=false/