Universidad Rey Juan Carlos

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERIA INFORMATICA

Curso Académico 2017/2018

Trabajo Fin de Grado

CONFIGURACIÓN OPTIMA DE LOS PARÁMETROS GEOMÉTRICOS DE UN ROBOT MÓVIL CON DIRECCIONAMIENTO DIFERENCIAL

Autor: David Vacas Miguel

Director/Tutor: Alberto Herrán González

Índice

Agradecimientos

Me gustaría agradecer a la universidad por el conocimiento recibido y en especial a mi tutor, Alberto Herrán, por su ayuda y apoyo durante el desarrollo del proyecto.

Resumen

En este TFG se ha desarrollado un algoritmo que calcula los parámetros óptimos de un robot con direccionamiento diferencial para obtener el mejor tiempo en un determinado circuito.

Se ha comenzado estudiando el funcionamiento del robot sobre el circuito puesto que esto proporciona la función objetivo del algoritmo. A continuación, se pasa al análisis del algoritmo utilizado, comenzando con una descripción de la metaheurística utiliza y sus diferentes partes: métodos constructivos, generación de vecindarios y selección de la siguiente solución. Inmediatamente después se realizó la implementación del algoritmo. Finalmente se implementaron diferentes tipos de instancias, constructivos, generadores de vecindarios selectores de la próxima solución y una búsqueda global.

Capítulo 1 Introducción

1.1. Motivación

Este trabajo nace motivado del problema que resulta tratar de buscar los parámetros óptimos de un robot con direccionamiento diferencial sobre un circuito determinado.

Este trabajo se basa en una aplicación anterior la cual simula la navegación autónoma de un robot con direccionamiento diferencial sobre un circuito. Este robot puede ser parametrizado de forma sencilla a través de una interfaz como se puede observar en la figura 1.

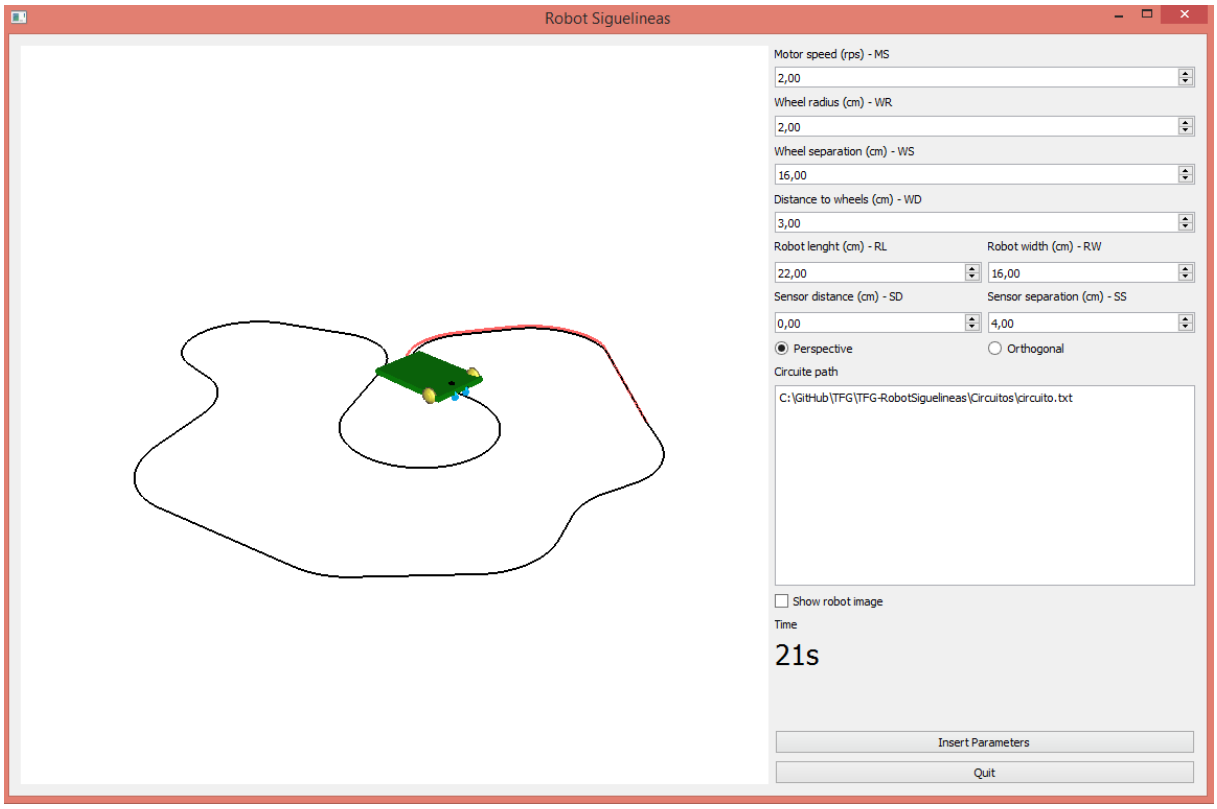


Figura 1. Aplicación de simulación de robot con direccionamiento diferencial.

Dado que se manejan diferentes parámetros geométricos de forma simultánea, puede llegar a ser complicado intentar tratar de ajustarlos de forma manual.

Debido a esto se ha decidido desarrollar un algoritmo que realice la búsqueda de los parámetros óptimos de forma automática.

1.2. Objetivos

Una vez vistos los motivos por los cuales se realiza este proyecto, los objetivos con los cuales subsanarlos, además de otros adicionales son:

- **Obtención de parámetros óptimos:** el objetivo principal del algoritmo se basa en obtener los parámetros óptimos del robot para un circuito determinado ya que la obtención de estos de forma manual es realmente costosa en esfuerzo y tiempo.

Por lo tanto, los requisitos que debe tener el algoritmo son:

- Conseguir los mejores parámetros para un determinado circuito.
- Incluir limites reales en los valores geométricos que se pueden obtener.
- Posibilidad de utilizarse en distintos circuitos.

1.3. Estado del arte

Para la resolución de problemas de optimización existen diferentes tipos de metaheurísticas[3]. Unos de los posibles tipos son las trayectoriales y las poblacionales. El aspecto diferenciador entre estas metaheurísticas es el numero de soluciones que se utilizan en el proceso de optimización. Las trayectoriales se basan utilizan una solución durante el proceso de búsqueda, es decir, la solución describe una trayectoria desde la solución inicial hasta encontrar la solución final. Por contrario, las metaheurísticas poblacionales hacen uso de un conjunto de soluciones que se optimizan de forma simultánea durante la búsqueda.

Las metaheurísticas trayectoriales más relevantes son:

- **Búsqueda local:** se basa en a partir de una solución inicial, la búsqueda de una mejor solución en un vecindario. Esto se repite de forma iterativa. La búsqueda local es la base de muchas metaheurísticas más complejas, en las cuales se modifican la forma de generar la solución inicial, la generación de vecindarios, la elección del vecino óptimo, la condición de finalización, etc.
- **Algoritmos voraces:** se basa en la elección optima en cada paso local, con la intención de conseguir así la solución general óptima. En caso de que la primera elección optima no sea factible, se descarta el elemento.

- **Algoritmo temple simulado:** es un tipo de búsqueda global capaz de escapar de mínimos locales admitiendo en ciertas situaciones peores soluciones a la actual durante la búsqueda. [3]
- **Búsqueda Tabú:** al igual que el anterior, es un algoritmo capaz de escapar de mínimos locales debido a que recopila información durante la exploración, la cual se utiliza para restringir las elecciones de vecinos en los vecindarios. Moverse hacia otra solución ocurre siempre después de analizar lo ocurrido anteriormente. [4]
- **VNS (Variable Neighborhood Search):** existen muchas variaciones de esta metaheurística, la más realiza una búsqueda local hasta que se encuentra con un optimo local, momento en el cual cambia su búsqueda de vecindarios para tratar de escapar de ese optimo local**.** [5]

En cuanto a las metaheurísticas poblacionales, algunas de las más relevante son:

- **Algoritmos genéticos:** se inspiran en la evolución biológica y su base genético-molecular. Se trata de una metaheurística de evolución que, a lo largo de las diferentes iteraciones, mantienen un conjunto de posibles soluciones del problema, cuyos valores evolucionan hacia las mejores soluciones mediante un proceso combinado de selección de individuos y operadores genéticos. Es decir, se basa en la recombinación de soluciones. [6]
- **Scatter search:** al igual que en los algoritmos genéticos, esta metaheurística se basa en la combinación de las soluciones para crear nuevas. La diferencia principal respecto a los genéticos reside en la población a considerar. Los algoritmos genéticos trabajan sobre grandes poblaciones escogidas aleatoriamente, sin embargo, scatter search trabaja sobre grupos pequeños escogidos estratégicamente. [7]
- **Ant colony optimization:** este algoritmo se agrupa dentro de los algoritmos de inteligencia de enjambres. La idea proviene, como el nombre indica, de la forma de comunicar caminos óptimos entre hormigas de una colonia. Cada "hormiga" genera incrementalmente una solución del problema. Cuando completa una solución, la evalúa y modifica el valor del camino. Esta información dirige a las futuras "hormigas". [8]

1.4. Estructura de la memoria

A continuación, se describe brevemente la estructura del resto del documento:

En el Capítulo 2, ***Descripción del problema***, se deduce la formulación matemática del problema sobre un robot con direccionamiento diferencial, es decir se deducen las fórmulas necesarias para la optimización del algoritmo. A continuación, se explica la navegación autónoma que realiza el robot. Por último, se describe como se lleva a cabo la simulación del sistema.

En el Capítulo 3, ***Descripción algorítmica***, se comienza realizando una introducción a la metaheurística usada y se pasa a describir su aplicación: métodos constructivos utilizados y generadores de vecindarios. Para finalizar se explica el algoritmo de búsqueda local básica y posibles metaheurísticas con las que escapar de los óptimos locales.

En el Capítulo 4, ***Implementación***, se comienza informando sobre la metodología utilizada en el proyecto, lenguaje de programación, sistema de control de versiones, etc. A continuación, se pasa a explicar el diseño de la aplicación mediante un diagrama de clases. Por último, se describen los detalles algorítmicos de bajo nivel.

En el Capítulo 5, ***Resultados***, se comparan los diferentes resultados en tiempo de ejecución y solución final, obtenidos sobre las diferentes instancias y búsquedas locales mediante gráficas y tablas.

En el Capítulo 6, ***Conclusiones***, en este apartado se exponen los resultados personales de realizar este trabajo y las posibles mejoras que se pueden realizar al mismo.

Capítulo 2 Descripción del problema

2.1. Direccionamiento diferencial

Los robots móviles son robots que, gracias a los progresos entre la inteligencia artificial y los robots físicos, son capaces de moverse a través de cualquier entorno físico. Estos normalmente son controlados por software y usan sensores y otros equipos para identificar la zona de su alrededor.

Los robots móviles se pueden diferenciar en dos tipos, autónomos y no autónomos. Los robots móviles autónomos pueden explorar el entorno sin ninguna directriz externa a él, al contrario que los no autónomos, que necesitan algún tipo de sistema de guía para moverse.

Los vehículos con ruedas son un tipo de robot móvil los cuales proporcionan una solución simple y eficiente para conseguir movilidad sobre terrenos duros y libres de obstáculos, con los que se permite conseguir velocidades más o menos altas.

Su limitación más importante es el deslizamiento en la impulsión, además dependiendo del tipo de terreno, puede aparecer deslizamiento y vibraciones en el mismo.

Otro problema que tienen este tipo de vehículos se halla en que no es posible modificar la estabilidad para adaptarse al terreno, excepto en configuraciones muy especiales, lo que limita los terrenos sobre los que es aceptable el vehículo.

Los vehículos con ruedas emplean distintos tipos de locomoción que les da unas características y propiedades diferentes entre ellos en cuanto a eficiencia energética, dimensiones, maniobrabilidad y carga útil. El robot simulado en este programa tiene una configuración con direccionamiento diferencial.

En este sistema las ruedas se sitúan de forma paralela y no realizan giros. El direccionamiento diferencial viene dado por la diferencia de velocidades de las ruedas laterales. La tracción se consigue con esas mismas ruedas. Adicionalmente, existen una o más ruedas para el soporte. En la figura 2 se muestra una imagen de dicho esquema. [1]

Figura 2: Robot con direccionamiento diferencial.

Para poder realizar la simulación del movimiento del robot se debe conocer primeramente el estado del sistema. El estado del robot viene dado por su posición y orientación. Puesto que el robot se va a mover, se necesita saber el estado del mismo en los diferentes instantes de tiempo, por lo tanto, se debe definir un modelo de cambios de estado, es decir, unas ecuaciones que a partir del estado actual y unas entradas permiten calcular el estado del sistema en el siguiente instante de tiempo. En la ecuación (1) se puede observar la ecuación que define el sistema, siendo $\bar{s}$ el estado y $\bar{r}$ las entradas.

$$\bar{s}(t+\Delta t) = f(\bar{s}(t), \bar{r}(t)) \tag{1}$$

Para el cálculo de un robot móvil con direccionamiento diferencial $\bar{r}$ son las velocidades de las ruedas izquierda y derecha $(w_i(t), w_d(t))$, mientras que el estado $\bar{s}$ viene dado por la posición *(x(t), z(t))* y orientación $\varphi(t)$ del robot, tal y como se muestra en la ecuación (2).

$$\begin{aligned} \bar{s}(t) &= (x(t), z(t), \varphi(t)) \\ \bar{r}(t) &= (w_i(t), w_d(t)) \end{aligned} \tag{2}$$

En la figura 10 se pueden observar tanto el estado del sistema como sus entradas sobre el robot en cuestión.

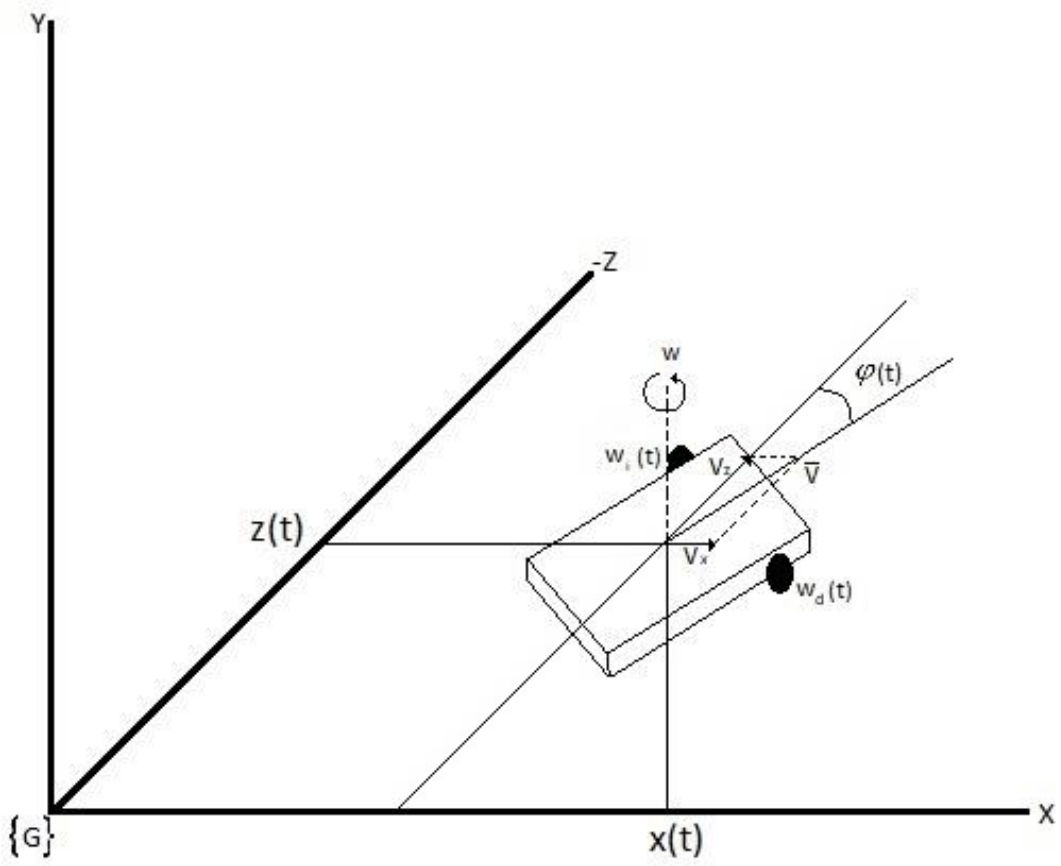


Figura 10. Estado y entradas de un robot con direccionamiento diferencial.

A partir de la figura 10, si el vehículo tiene una velocidad de desplazamiento v y de rotación w, se obtiene los componentes mostrados en la ecuación (3):

$$\begin{aligned} v_x &= v \cdot \sin(\varphi) \\ v_z &= v \cdot \cos(\varphi) \\ v_\varphi &= w \end{aligned} \tag{3}$$

Por otro lado, la derivada de una función puede aproximarse por el cociente incremental mostrado en la ecuación (4). Esto se conoce como derivada discreta hacia adelante, pero también puede aproximarse con el cociente incremental hacia atrás, la aproximación centrada, u otras aproximaciones más complicadas:

$$v_x = \frac{dx}{dt} = \frac{x(t+1) - x(t)}{dt} \tag{4}$$

El resultado es que, con una ecuación de este tipo, la nueva coordenada x $\{t+\Delta t\}$ se puede calcular a partir de la anterior (en *t*) mediante la ecuación:

$$x(t+\Delta t) = x(t) + v_x \cdot \Delta t \tag{5}$$

Aplicando el mismo resultado al resto de coordenadas del modelo cinemático directo, se obtiene la ecuación de cambio de estado (6):

$$\begin{aligned} x(t+\Delta t) &= x(t) + v_x \cdot \Delta t \\ z(t+\Delta t) &= z(t) + v_z \cdot \Delta t \\ \varphi(t+\Delta t) &= \varphi(t) + v_\varphi \cdot \Delta t \end{aligned} \tag{6}$$

Con el objetivo de buscar una ecuación similar a la mostrada en las ecuaciones (2), debemos relacionar las velocidades (v_x, v_z, v_φ) con las entradas reales del sistema $(\mathrm{w}_i, \mathrm{w}_d)$.

Sean w_i y w_d las velocidades de giro de las ruedas izquierda y derecha, respectivamente. Si el radio de la rueda es WR, las velocidades lineales correspondientes son $v_i = w_i \cdot WR$ y $v_d = w_d \cdot WR$. Es este caso, la velocidad lineal y velocidad angular correspondientes en el modelo vienen dadas por:

$$\begin{aligned} v &= \frac{v_d + v_i}{2} = \frac{(\mathrm{w}_d + \mathrm{w}_i)\cdot \mathrm{WR}}{2} \\ w &= \frac{v_d - v_i}{WS} = \frac{(\mathrm{w}_d - \mathrm{w}_i)\cdot \mathrm{WR}}{WS} \end{aligned} \tag{7}$$

Sustituyendo estas expresiones en las obtenidas en la ecuación (3), se obtienen los componentes de la velocidad del robot en el sistema *{G}* a partir de la velocidad de giro de cada rueda:

$$\begin{aligned} v_x &= \frac{-(\mathrm{w}_d + \mathrm{w}_i)\cdot \mathrm{WR}}{2}\cdot \sin(\varphi) &&= -(\mathrm{w}_d + \mathrm{w}_i)\cdot \frac{\mathrm{WR}\cdot \sin(\varphi)}{2} \\ v_z &= \frac{(\mathrm{w}_d + \mathrm{w}_i)\cdot \mathrm{WR}}{2}\cdot \cos(\varphi) &&= (\mathrm{w}_d + \mathrm{w}_i)\cdot \frac{\mathrm{WR}\cdot \cos(\varphi)}{2} \\ v_\varphi &= \frac{(\mathrm{w}_d - \mathrm{w}_i)\cdot \mathrm{WR}}{WS} &&= (\mathrm{w}_d - \mathrm{w}_i)\cdot \frac{\mathrm{WR}}{WS} \end{aligned} \tag{8}$$

Finalmente, utilizando el modelo discreto mostrado en la ecuación (6), se obtiene:

$$\begin{aligned} x(t+\Delta t) &= x(t) - (w_d + w_i) \cdot \frac{WR \cdot \sin(\varphi(t))}{2} \cdot \Delta t \\ z(t+\Delta t) &= z(t) + (w_d + w_i) \cdot \frac{WR \cdot \cos(\varphi(t))}{2} \cdot \Delta t \\ \varphi(t+\Delta t) &= \varphi(t) + (w_d - w_i) \cdot \frac{WR}{WS} \cdot \Delta t \end{aligned} \quad (9)$$

Para terminar, si utilizamos las variables s para representar el estado de los sensores (s=0 si esta sobre la línea y s=1 en caso contrario), se obtiene:

$$\begin{aligned} x(t+\Delta t) &= x(t) - w \cdot (s_d + s_i) \cdot \frac{WR \cdot \sin(\varphi(t))}{2} \cdot \Delta t \\ z(t+\Delta t) &= z(t) + w \cdot (s_d + s_i) \cdot \frac{WR \cdot \cos(\varphi(t))}{2} \cdot \Delta t \\ \varphi(t+\Delta t) &= \varphi(t) + w \cdot (s_d - s_i) \cdot \frac{WR}{WS} \cdot \Delta t \end{aligned} \quad (10)$$

Y definiendo las constantes $k_1 = \frac{w \cdot \Delta t}{2}$ y $k_2 = w \cdot \Delta t$ resulta:

$$\begin{aligned} x(t+\Delta t) &= x(t) - k_1 \cdot (s_d + s_i) \cdot WR \cdot \sin(\varphi(t)) \\ z(t+\Delta t) &= z(t) + k_1 \cdot (s_d + s_i) \cdot WR \cdot \cos(\varphi(t)) \\ \varphi(t+\Delta t) &= \varphi(t) + k_2 \cdot (s_d - s_i) \cdot \frac{WR}{WS} \end{aligned} \quad (11)$$

Estas ecuaciones son las que nos sirven para poder calcular el estado del robot en el instante siguiente, es decir, la posición y la orientación en $(t+\Delta t)$. Las variables necesarias para poder calcular este son: el estado anterior ya sea su posición $x(t), z(t)$ u orientación $\varphi(t)$, el radio de las ruedas WR y separación entre las mismas WS, así como la velocidad de giro de los motores w y el paso de la simulación Δt. El cálculo de las variables s se explica a continuación.

2.2. Navegación autónoma

Se coloca el robot sobre un fondo blanco con una línea negra que representa el circuito, como se muestra en la figura 4, y este deberá recorrer el circuito sin salirse del mismo. El robot sigue líneas que se ha implementado realiza su movimiento de manera autónoma, esto se puede realizar gracias a dos sensores que son implantados en la parte delantera del robot los cuales son responsables de la detección de la línea del circuito. En función de lo que estos sensores recojan (están sobre el circuito o no) el robot realiza cambios en la velocidad de sus ruedas resultando en un movimiento recto, rotatorio hacia la izquierda o rotatorio hacia la derecha.

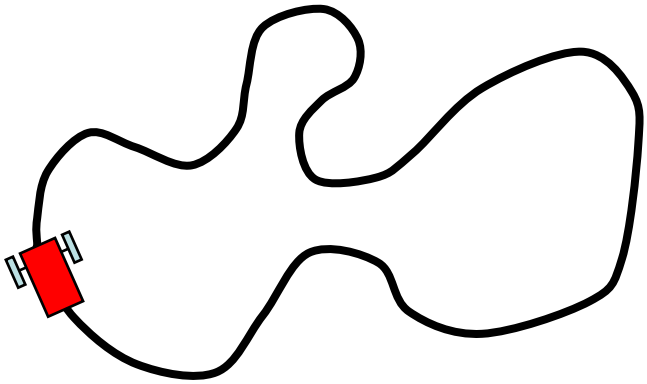


Figura 4. Navegación autónoma del robot sigue líneas.

Los sensores que se usan en este tipo de robots son sensores CNY70, los cuales se muestran en la Figura 5.

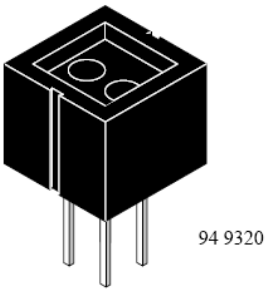


Figura 5. Sensor CNY70.

Estos son sensores ópticos reflexivos de corto alcance basados en un diodo de emisión de luz infrarroja y un receptor formado por un fototransistor que ambos apuntan en la misma dirección. La figura 6 muestra de forma simplificada su estructura interna.

Cuando el sensor se haya sobre una línea negra la luz emitida por el fotodiodo es absorbida por la misma y el fototransistor la señal correspondiente al circuito de control. Sin embargo, cuando se haya sobre fondo blanco la luz es reflejada y por lo tanto el fototransistor envía la señal contraria a la enviada al estar sobre negro.

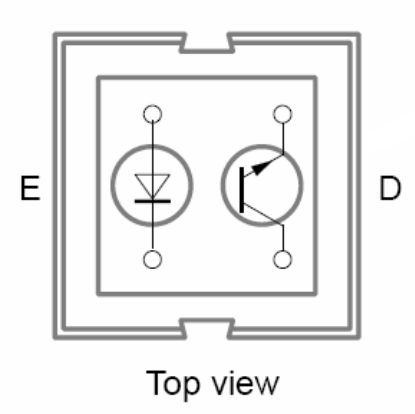


Figura 6. Estructura simplificada del sensor CNY70.

Para implementar dicho comportamiento en el simulador, se ha seguido la siguiente lógica:

`$s_i$ -> sensor izquierdo. | $s_d$ -> sensor derecho.`

Algoritmo 1: Algoritmo sensor

1. **if** *IsOnCircuite(s_i)*
2. *$Wi \leftarrow 0$*
3. **if** *IsOnCircuite(s_d)*
4. *$W_d \leftarrow 0$*

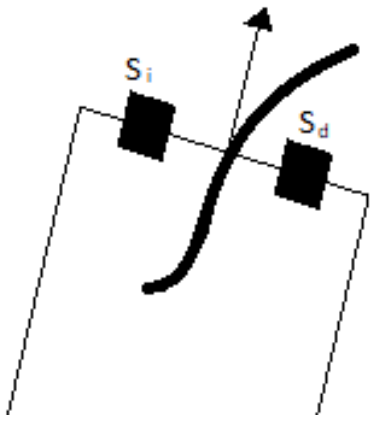


Figura 7. Posición sensores sobre robot.

Algoritmo 2: IsOnCircuite(pos)

```
1. for (i=0;  i<circuite.length; i++)
2.     distance = raiz(cuadrado(circuite(i).x - pos.x)) + cuadrado(circuite(i).z - pos.z)
3.     if distance<threshold
4.         return true
5. endfor
6. return false
```

Esta en una de las posibles implementaciones, otras opciones son: la rueda en vez de frenar realiza el giro hacia atrás $s_i = -1 \| s_d = -1$ o disminuye la velocidad en vez de frenar completamente $s_i = s_i - \Delta \| s_d = s_d - \Delta$ con $\Delta \in (0,1)$.

2.3. Formulación matemática (del problema de optimización)

La función objetivo de este algoritmo se trata de la simulación del robot realizando el circuito. Para poder implementarla se ha utilizado lo visto en este capítulo, es decir, se necesitan las ecuaciones (9) del apartado *2.1 Direccionamiento diferencial* para el calculo del movimiento y posición del robot en cada instante, y la simulación de los sensores vista en el apartado *2.2 Navegación autónoma* que determinan cuando va a realizarse un giro.

Los parámetros variables que se utilizan en la simulación son: la separación entre las ruedas, el radio de las ruedas, la distancia entre los ejes (distancia desde el centro de los sensores hasta el punto de rotación del robot) y la separación entre los sensores.

Esta función da como resultado un valor que es el utilizado para medir la calidad de cada solución. Dicho valor es el tiempo real que el robot tarda en realizar el circuito dado de principio a fin.

A continuación, se expone la función objetivo matemáticamente. De forma general se escribe como:

`x: vector de decisión`

f(x): función objetivo

g(x): restricciones

$$\begin{aligned} \min_{x} \quad & f(x) \\ s.t. \quad & g(x) \leq 0 \\ & x \in \mathbb{R} \end{aligned}$$

En particular para este problema las variables anteriores se definen como:

X = [WD, WR, d_{ejes}, $d_{sensores}$]

f(x): como se ha dicho anteriormente, esta función mide el tiempo en segundos que tarda el robot en dar una vuelta al circuito.

g(x) ≥ 0

11,1 ≤ WD ≤ 21,1

0,9 ≤ WR ≤ 3,1

0 ≤ d_{ejes} ≤ 6,1

1,9 ≤ $d_{sensores}$ ≤ 4,1

Capítulo 3 Descripción algorítmica

3.1. Búsqueda local

La búsqueda local es un tipo de metaheurística trayectorial que se basa en la búsqueda de una solución que mejore la actual en un vecindario alrededor de la solución actual. Este vecindario se genera utilizando una operación básica denominada "movimiento", que se aplica a los diferentes elementos de una solución y con esto se generan los diferentes vecinos del vecindario.

Esta metaheurística se puede dividir en 3 pasos principales: construcción de la solución inicial, generación del vecindario y elección de una solución del vecindario.

El pseudocódigo asociado a esta metaheurística es el siguiente:

```
x: solución inicial | x*: mejor solución actual | N: vecindario

constructor: Genera la primera solución.

neighborhood: Crea el vecindario.

select: Escoge la solución del vecindario.

fobj: Función objetivo que devuelve el valor que indica la calidad de
la solución.
```

Algoritmo 3: Algoritmo de búsqueda local

1. $x \leftarrow Constructor()$
2. **do**
3. $x^* \leftarrow x$
4. $N \leftarrow Neighborhood(x^*)$
5. $x \leftarrow Select(N, x^*)$
6. **while** $fobj(x) < fobj(x^*)$
7. **return** x

3.2. Métodos constructivos

Dado que se trata de un algoritmo que se basa en variables reales, el conjunto de valores iniciales tiene unos límites físicos, además puesto que se realiza una sola

ejecución de la aplicación no se puede tener en cuenta valores de ejecuciones anteriores para la generación de esta primera solución.

Por esto el método constructivo que se ha utilizado se basa en la generación de valores aleatorios con limites superiores e inferiores distintos para cada una de las variables. Con esto cada vez que empiece el algoritmo se comienza desde una solución totalmente aleatoria y distinta.

Puesto que se existen soluciones no factibles, es decir, robots que no pueden realizar el circuito, el método constructivo no para de sacar posibles robots hasta que uno de ellos sea factible y a partir de ese se continua con la ejecución del algoritmo.

A continuación, se muestra el pseudocódigo que utiliza el constructivo del algoritmo:

Algoritmo 3: Constructor

```
1. do
2.     robot ← GenerarRobot()
3.     robot.tiempo ← FuncionObjetivo()
4. while NoFactible(robot)
5. return robot
```

Algoritmo 4: GenerarRobot

```
1. param1 ← Random(minParam1, maxParam1)
2. param2 ← Random(minParam2, maxParam2)
3. param3 ← Random(minParam3, maxParam3)
4. param4 ← Random(minParam4, maxParam4)
5. return new Robot(param1, param2, param3, param4)
```

3.3. Generación de vecindarios

La generación de vecinos se basa en la perturbación de la solución actual para generar un conjunto de soluciones cercanas a la misma con el objetivo de que alguna mejore a la actual, en la figura 8 se puede observar los vecindarios creados a partir de unas soluciones.

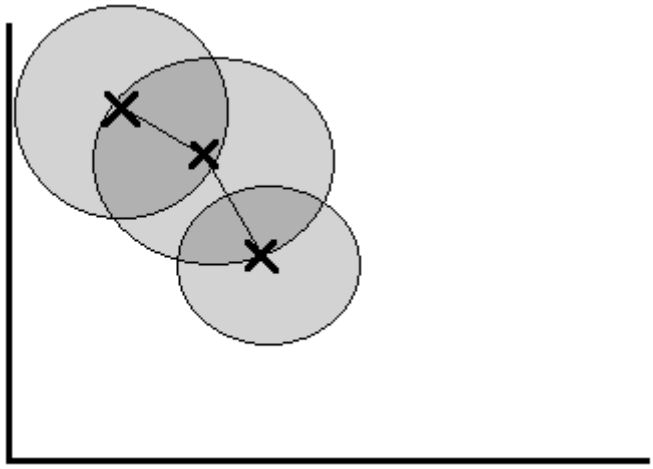


Figura 8. Vecindarios generados a partir de las soluciones.

Existen diferentes heurísticas para la generación de vecinos. Como se ha dicho anteriormente, debido a que se trata de valores reales, se ha optado por la modificación de variables para la generación del vecindario y el cambio de la magnitud de modificación de estas.

La opción elegida para realizar la búsqueda de vecinos se basa en que cada vecino se genera por las posibles combinaciones de adición y sustracción de una magnitud en tres valores distintos. Dicha magnitud toma los valores: 0.05, 0.1, 0.15 y 0.2.

Esta búsqueda genera una vecindad de 24 vecinos.

3.4. Búsquedas locales

Como se ha explicado en la introducción de este capítulo, el algoritmo de búsqueda local básica se basa en explorar el entorno de una solución con el fin de encontrar otra mejor. Para esto se realizan cambios sobre los diferentes elementos de una solución lo que genera más soluciones diferentes de la anterior, que se conocen como vecinos. De entre estos vecinos se elige uno que se convierte en la solución y se vuelve a iterar sobre este.

Todos los vecinos del vecindario se pasan por la función objetivo. Esta función es la encargada de informar sobre que vecino es mejor.

Existen diferentes formas de escoger el vecino que se convierte en la siguiente solución, algunas de ellas son:

- ***First***, se escoge el primer vecino que mejore a la solución actual. El pseudocódigo asociado a esta búsqueda es el siguiente:

```
actualRobot -> robot a partir del cual se generan vecinos

vecinos -> lista con los vecinos actuales
```

Algoritmo 5: First

```
1. foreach Robot vecino in vecinos
2.     vecino.tiempo ← FuncionObjetivo(vecino)
3.     if actualRobot.tiempo > vecino.tiempo
4.         return vecino
5.
```

- ***Best***, se escoge al mejor vecino. La implementación en pseudocódigo es la siguiente:

```
actualRobot -> robot a partir del cual se generan vecinos

vecinos -> lista con los vecinos actuales
```

Algoritmo 6: Best

```
1. mejorVecino ← actualRobot
2. foreach Robot vecino in vecinos
3.     vecino.tiempo ← FuncionObjetivo(vecino)
4.     if mejorVecino.tiempo > vecino.tiempo
5.         mejorVecino ← vecino
6. endforeach
7. return mejorVecino
```

3.5. Búsquedas globales

El principal problema de las búsquedas locales se haya en que se pueden quedar fácilmente atrapadas en un óptimo local. Un óptimo local es una solución que no puede ser mejorada en el vecindario que ella genera y que, además, no es la

solución óptima del problema. Las metaheurísticas que tratan de evitar este problema se las llama búsquedas globales.

Para solucionar el problema de los óptimos locales se puede utilizar:

- ***Multiarranque***, para tratar de evitar los óptimos locales, esta metaheurística reinicia la búsqueda desde una nueva solución construida con cualquiera de los métodos de construcción disponibles cuando se encuentra con un óptimo local. [9]

Algoritmo 7: Multiarranque

1. $y \leftarrow BusquedaLocal()$
2. $z \leftarrow BusquedaLocal()$
3. **return** $y > z ? y : z$

- ***ILS (Iterated Local Search)***, esta metaheurística cuando se encuentra con un óptimo local reinicia la búsqueda desde la solución actual o una perturbación de esta.

Algoritmo 8: Iterated Local Search

1. $x \leftarrow Constructor()$
2. $BusquedaSoluciones()$
3. $x \leftarrow PerturbarSolucion(x^*)$
4. $BusquedaSoluciones()$
5. **return** x^*

Algoritmo 9: BusquedaSoluciones

1. **do**
2. $x^* \leftarrow x$
3. $N \leftarrow Neighborhood(x^*)$
4. $x \leftarrow Select(N, x^*)$
5. **while** $fobj(x) < fobj(x^*)$

- ***VNS (Variable Neighborhood Search)***, como se ha explicado en el apartado *1.3 Estado del arte,* hay varios tipos de VNS, la gran parte de ellos se basa en tener varias formas de crear vecindarios que poder usarse y un "shake" cuando el algoritmo queda atrapado en un óptimo local.

Algoritmo 10: BVNS (Basic VNS)

1. $S^* \leftarrow Constructive()$
2. **while** $t < t_{max}$ **do**
3. $k \leftarrow 1$
4. **while** $k < k_{max}$ **do**
5. $S' \leftarrow Shake(S^*,k)$
6. $S'' \leftarrow LocalSearch(S')$
7. **if** $S'' < S^*$
8. $S^* \leftarrow S''$
9. $k \leftarrow 1$
10. **else**
11. $k \leftarrow k+1$
12. **endwhile**
13. $T \leftarrow CPU()$
14. **endwhile**
15. **return** S^*

- ***SA (Simulated Annealing)***, se basa en el tratamiento con calor de la metalurgia. Esta metaheurística acepta soluciones peores para intentar llegar al óptimo.

`c: valor de control de cada iteración`

`x*: mejor solución global |x': mejor solución parcial`

Algoritmo 11: Simulated Annealing

1. *x* ← *Constructor()*
2. *c* ← *Init(x)*
3. *L* ← *Length(c)*
4. *x** ← *x*
5. **do**
6. *x* ← *Cicle(c,L,x)*
7. **if** *fobj(x)<fobj(x*)*
8. *x** ← *x*
9. *c* ← *Cooling(c)*
10. *L* ← *Length(c)*
11. **while** *criterio_parada == false*
12. **return** *x**

Algoritmo 12: Cicle(c,L,x)

1. **for** *l* ← *1* **to** *L*
2. *N* ← *Neighborhood(x)*
3. *x'* ← *RandomSel(N)*
4. **if** *criterio_aceptacion(x,x*)==true*
5. *x* ← *x'*
6. **endfor**
7. **return** *x*

Capítulo 4 Implementación

4.1. Metodología

En cuanto a la tecnología utilizada para la realización de este proyecto se ha utilizado como lenguaje de programación **Java 8**, un lenguaje de programación de propósito general, concurrente y orientado a objetos. Se ha utilizado el IDE **NetBeans 8.2** puesto que tiene una buena integración con Java[4].

En lo referido a la metodología se han utilizado herramientas utilizadas habitualmente en proyectos en los que se aplica metodología ágil, principalmente Trello y GitHub.

Trello como tablero de tareas, este se divide en las columnas habituales (Product backlog, To Do, Doing, Done). A su vez cada tarea tiene asignada una dificultad representada mediante colores. Las tareas no tienen asignadas personas puesto que hay una sola persona encargada de este tablero. A pesar de no ser relevante para la organización de un equipo y la división de tareas, puesto que solo se trata de una persona, ha resultado muy útil para no perder la visión de proyecto. Todo esto se puede ver en la figura 9.

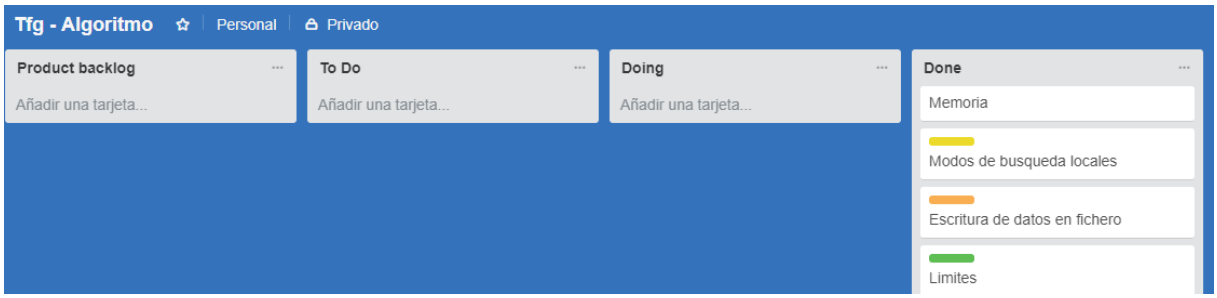


Figura 9. Trello utilizado en el desarrollo del proyecto.

Git como repositorio y control de versiones (utilizado concretamente GitHub). Sobre este repositorio se ha ido subiendo los diferentes incrementos de funcionalidad de la aplicación de forma periódica y gracias a él se ha podido realizar un control de versiones. Se puede observar el repositorio desde la aplicación de Windows en la figura 10.

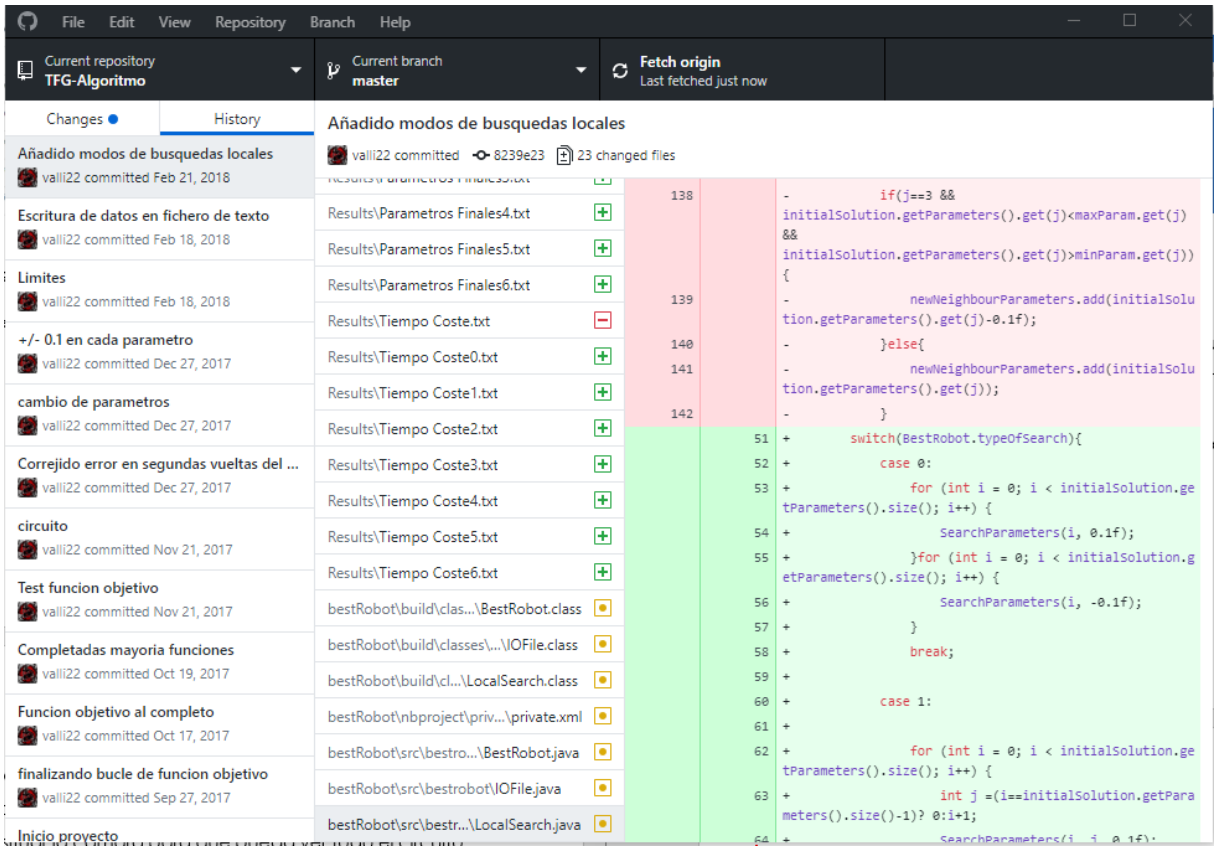


Figura 10. Repositorio en GitHub desde la aplicación de escritorio de Windows.

4.2. Diseño

El diseño de la aplicación se puede observar en el UML que se muestra en la figura 11.

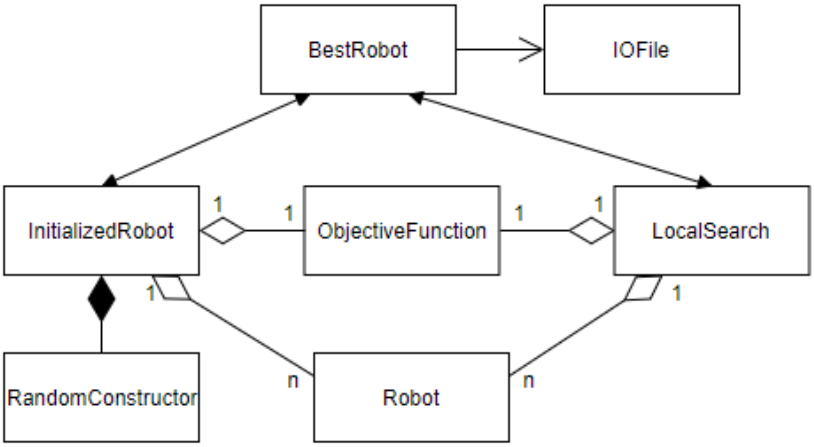


Figura 11. UML

A continuación, se explica brevemente el funcionamiento de las clases expuestas en el UML:

- **BestRobot**, esta clase es la clase principal donde se haya la lógica del algoritmo, y la que ira llamando a las distintas clases.
- **InitializedRobot**, es la clase constructora del algoritmo. Construye la primera solución desde la que el algoritmo parte.
- **RandomConstructor**, esta clase se usa en la clase InitializedRobot para determinar los valores de la primera solución. Esta clase genera los valores aleatorios que tiene la primera solución del algoritmo y además tiene los limites sobre los cuales se pueden mover los valores durante el algoritmo.
- **LocalSearch**, como se indica en el nombre, esta clase es la encargada de generar los vecindarios de las diferentes soluciones.
- **ObjectiveFunction**, como se indica en el nombre, esta clase se encarga de obtener el valor por el cual se comparan los vecinos para comprobar cuál es mejor.
- **Robot**, esta clase guarda la información necesaria sobre un robot, es decir guarda los valores que serán permutados a lo largo del algoritmo.
- **IOFile**, se trata de la clase que escribe en ficheros de texto los datos que se quieren guardar.

En cuanto a la ejecución del programa, se debe entrar al código para cambiar la forma en la que se ejecuta. De forma predefinida se ejecuta el programa 10 veces por cada combinación de tipos de búsquedas locales e instancias. Si se desea cambiar esto se puede cambiar el primer parámetro que se encuentra en la clase ***`BestRobot`***, este parámetro ***`typeOfExecution`***, indica el tipo de ejecución que se desea. Si se deja a 0 se realiza la ejecución predeterminada, si se pone a 1 se realiza una sola ejecución del programa y se puede modificar tanto el robot, como el tipo de búsqueda y la instancia (esto se cambia al principio del código de esta ejecución), si por otra parte se pone este parámetro en 2 se realiza lo mismo que en el caso anterior, sin embargo, se aplica ILS.

El método de obtener los resultados se basa en la escritura de los mismos en ficheros de textos, las variables ***resultsPath*** y ***resultsPathOptimo*** indican el path de las carpetas donde se guardarán los ficheros de los resultados. Este path se puede modificar cambiando el contenido de estas variables. En la carpeta ***resultsPath*** se guarda por cada iteración de la aplicación el tiempo de CPU y el tiempo del mejor robot en esa iteración. En cambio, en la carpeta indicada por ***resultsPathOptimo*** se guardan los parámetros del mejor robot obtenido tras finalizar el algoritmo.

Capítulo 5 Resultados

5.1. Descripción de las instancias

Las instancias son los parámetros que el usuario no puede modificar. El algoritmo se va a ejecutar sobre 9 instancias que resultan de la combinación de 3 circuitos distintos y 3 velocidades de ruedas diferentes. El primer circuito es un circuito estándar, tamaño medio y con algunas curvas, se puede observar en la figura 12.

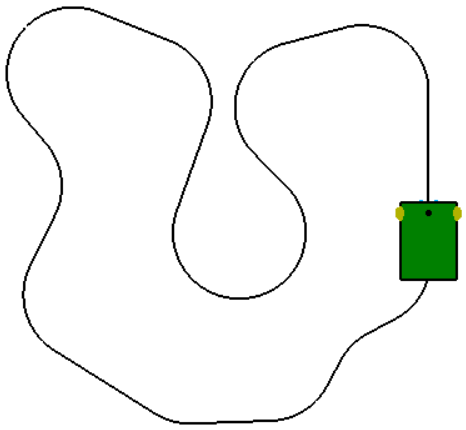


Figura 12. Circuito 1 estándar

El segundo de los circuitos se trata de un circuito de gran tamaño sin gran cantidad de curvas, este se puede observar en la figura 13.

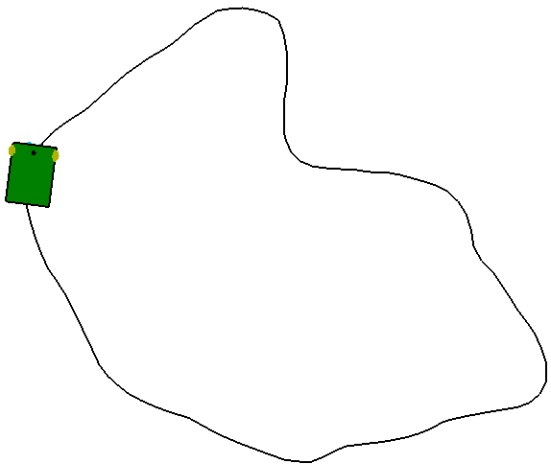


Figura 13. Circuito 2 grande sin curvas

El tercero de los circuitos se trata de un circuito de tamaño medio, pero con curvas muy cerradas que hace que varios de los posibles robots no puedan realizarlo, se puede observar este circuito en la figura 14.

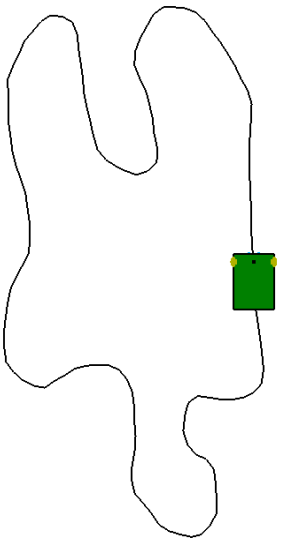


Figura 14. Circuito 3 tamaño medio con curvas cerradas

Las velocidades del robot que se utilizan son 2,3 y 4 rps.

CIRCUITO / VEL. RUEDAS	CIRCUITO 1	CIRCUITO 2	CIRCUITO 3
1RPS	Instancia 1	Instancia 4	Instancia 7
2RPS	Instancia 2	Instancia 5	Instancia 8
3RPS	Instancia 3	Instancia 6	Instancia 9

Tabla 1.

La ejecución del algoritmo se ha realizado sobre un ordenador con las siguientes especificaciones:

- Procesador: i7-6400 4GHz.
- RAM: 16 GB.
- SO: Windows 10 Home.

5.2. Métodos Constructivos

Los métodos constructivos crean la primera solución del algoritmo. El método constructivo implementado genera valores aleatorios dentro de un rango específico para cada valor, es decir, cada valor tiene unos límites adecuados para su geometría real.

A continuación, se muestra una tabla en la que es expone el resultado medio de la ejecución del constructivo 10 veces sobre cada instancia:

		Coste
Circuito 1	Instancia 1	33,87
	Instancia 2	23,72
	Instancia 3	26,13
Circuito 2	Instancia 4	28,53
	Instancia 5	53,17
	Instancia 6	20,44
Circuito 3	Instancia 7	41,56
	Instancia 8	30,63
	Instancia 9	21,65

Tabla 2. Valores del constructivo sobre cada instancia.

Como se puede observar, al aumentar la velocidad del robot disminuye el tiempo en el que realiza el circuito. Esto no pasa en el circuito 1, sin embargo, al ser un constructor aleatorio es posible que se generen irregularidades, y esta es una de ellas. Además, se puede observar que el circuito 2 de media es más rápido recorrerlo, sin embargo, el circuito 3 es el más lento.

5.3. Búsquedas locales

Se elige como se va a realizar la búsqueda de vecindarios. Para ello se ha usado la generación de vecinos vista en el punto *3.3 Búsquedas locales.* A partir de esta se generan 8 posibles búsquedas locales combinando el cambio del parámetro de

modificación de vecinos (0.05, 0.1, 0.15, 0.2) y los métodos de búsqueda locales **best** y **first**. Se puede observar dichas combinaciones en la tabla 3.

	BEST	**FIRST**
0.05	Búsqueda local 1	Búsqueda local 5
0.1	Búsqueda local 2	Búsqueda local 6
0.15	Búsqueda local 3	Búsqueda local 7
0.2	Búsqueda local 4	Búsqueda local 8

Tabla 3. Combinaciones de búsquedas locales

A continuación, se ha ejecutado el algoritmo sobre todas las búsquedas locales e instancias. Este proceso se ha repetido 10 veces y se ha sacado el tiempo medio de los tiempos medios de cada instancia. Se pueden observar los resultados en la tabla 4.

		Coste	**TiempoCPU**	**Desviación**	
Best	LS 1	15,56	143,01	0,91	0
	LS 2	15,39	73,81	0,74	0
	LS 3	14,98	64,18	0,33	1
	LS 4	**14,65**	54,84	0	7
First	LS 5	16,69	6,94	2,04	0
	LS 6	16,59	5,53	1,94	0
	LS 7	16,13	**4,41**	1,48	0
	LS 8	16,07	4,48	1,42	1

Tabla 4. Tabla de resultados.

Si nos fijamos en las búsquedas locales, se puede ver una clara diferencia de tiempos entre las búsquedas con **best** y **first**, donde las primeras alcanzan un valor mejor que las segundas. Además de esto también se puede observar como el aumento del valor de modificación de los vecinos implica una mejora en los tiempos de los robots (LS4<LS3<LS2<LS1 y LS8<LS7<LS6<LS5).

Además, en lo respectivo al tiempo que se tarda en encontrar el mejor valor, es mucho menor utilizando **first** que **best**, destacando como más rápida la búsqueda local 7. También se puede comprobar como al aumentar el valor de modificación de vecinos se encuentra el mejor valor de manera más rápida.

Una vez vistas las mejores búsquedas vamos a comparar de forma mas general los datos obtenidos.

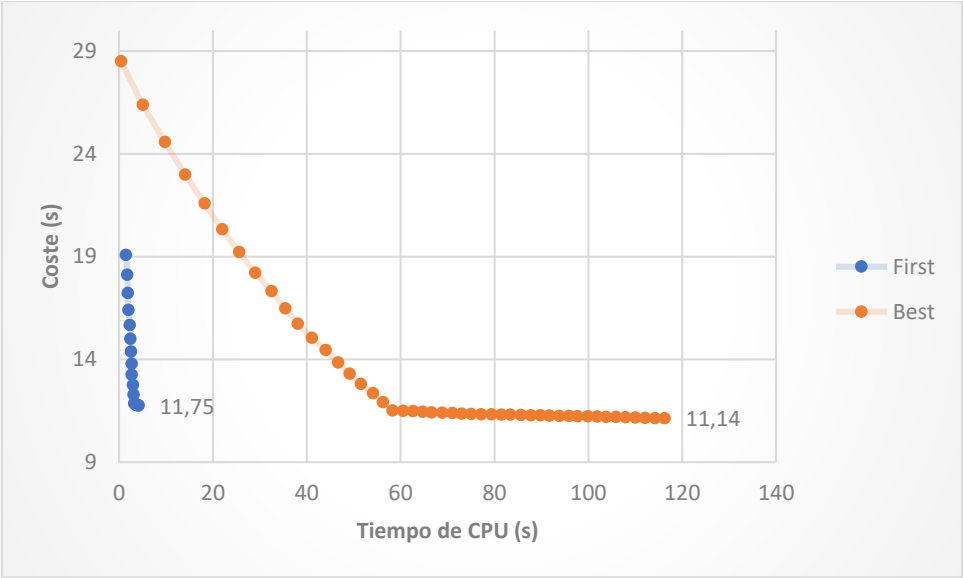
En cuanto a la relación calidad de la solución tiempo de ejecución, **best** obtiene resultados un **8%** mejores que **first**, sin embargo, tarda **25.83** veces más en obtenerlos.

En lo respectivo al valor de modificación de los vecinos, su aumento implica una mejora tanto del valor obtenido como del tiempo de ejecución. En cuanto a la mejora obtenida, por cada aumento del valor en 0.05 se mejora el tiempo del robot entre un **1%-2%**. El porcentaje de mejora del tiempo de ejecución, al aumentar el modificador de los vecinos varía entre los diferentes aumentos y si es **best** o **first**, esta mejora se muestra en la tabla 5.

	BEST	**FIRST**
0.05-0.1	93%	25%
0.1-0.15	15%	25%
0.15-0.2	17%	-2%

Tabla 5. Mejora del tiempo de ejecución del algoritmo al aumentar el valor de modificación de vecinos en best y first.

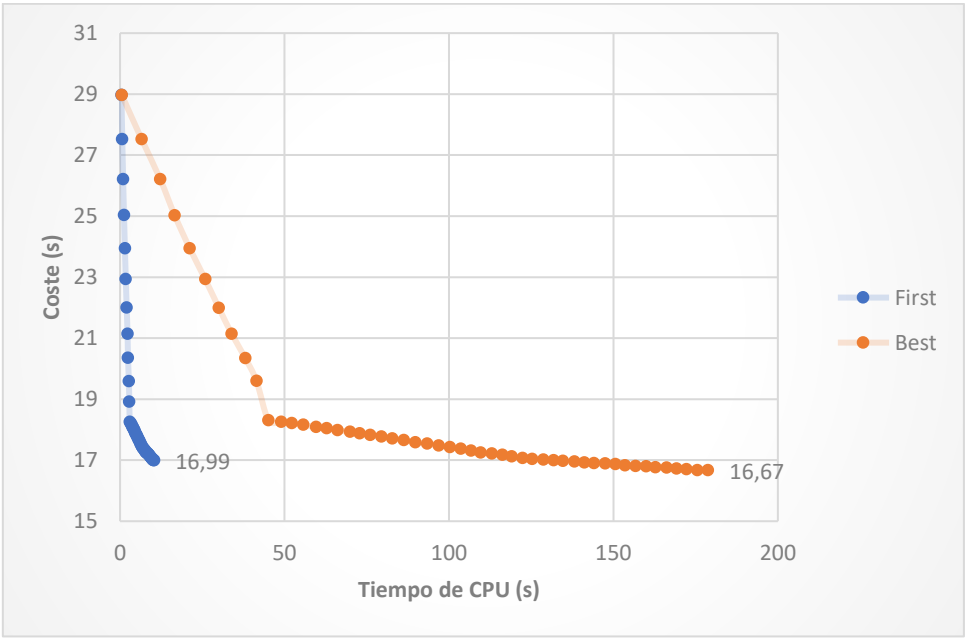
A continuación, se muestra una gráfica en la que se muestra un caso estándar (Búsqueda local 2 y 6 – Instancia 3), con **best** y **first**, y los valores obtenidos a lo largo del tiempo de ejecución.



Gráfica 1. Comparación best y first sobre misma instancia.

En esta gráfica se puede observar como mediante **first** se encuentra un valor óptimo muy rápido, sin embargo, el valor que consigue **best** es mejor. Se puede comprobar también como **best** parece que se estanca, pero continúa sacando mejores valores, sin embargo, **first** se estanca y no consigue mejorar su resultado.

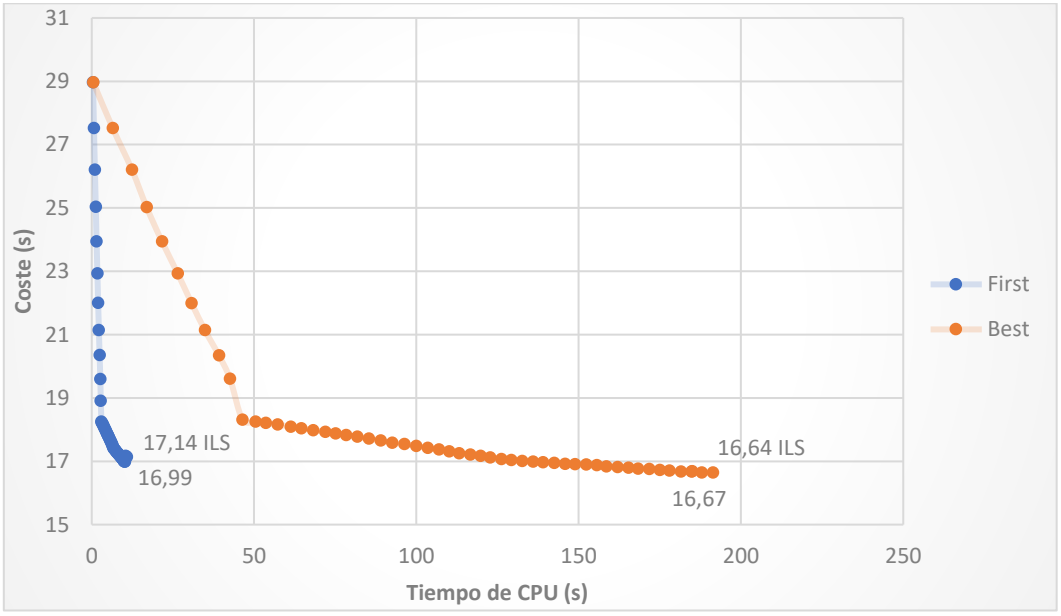
En la siguiente gráfica se ha modificado el constructor para que **best** y **first** empiecen desde el mismo robot inicial (Búsqueda local 2 y 6 – Instancia 1).



Gráfica 2. Comparación best (naranja) y first (azul) desde mismo robot inicial.

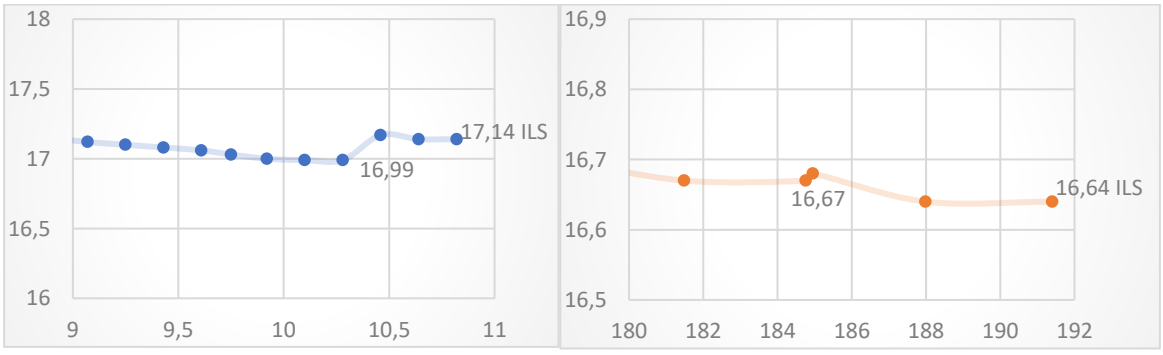
Gracias a esta gráfica se puede comprobar lo dicho anteriormente, puesto que, a pesar de partir del mismo robot, ocurren los mismo hechos que en las anteriores, es decir, **first** obtiene un valor un poco peor pero mucho más rápido que **best**.

Por último, se ha utilizado **ILS** sobre la ejecución de la gráfica 2.



Gráfica 3. Misma ejecución que gráfica 2 aplicando ILS.

Para poder observar mejor lo ocurrido se han ampliado la gráfica, en la gráfica 4 se puede observar esta misma ampliada.



a) Final first ampliado **b)** Final best ampliado

Gráfica 4. Final ampliado de la gráfica 3.

Tras utilizar ILS se puede observar como tras permutar el optimo local se puede obtener un resultado mejor, en este caso sucede en **best** como se puede observar en la gráfica 5 caso b). Sin embargo, también puede que tras utilizar ILS no mejore la solución como se puede ver en la gráfica 5 caso a) que muestra el resultado tras utilizarlo sobre **first**.

5.4. Resultados finales

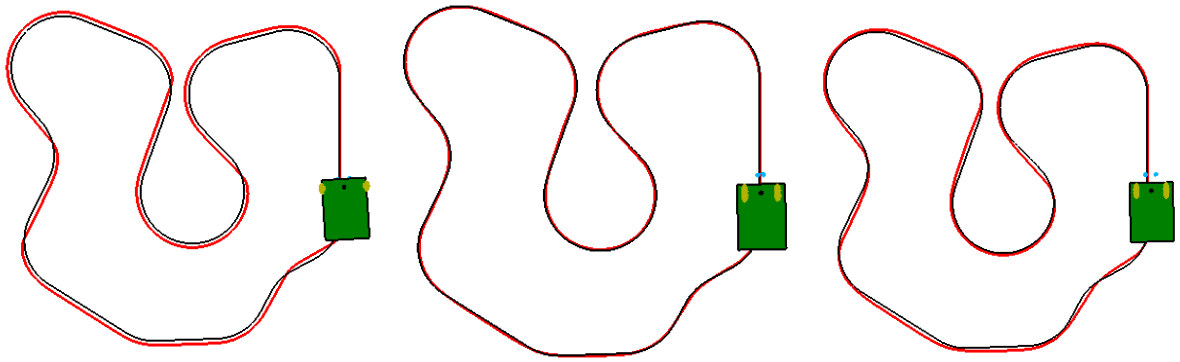
Una vez comprobados los mejores métodos, se comprueban los cambios realizados en los valores geométricos del robot. A continuación, se muestra una tabla con los parámetros iniciales y los finales utilizados en la gráfica 2.

	INICIALES	BEST	FIRST
SEPARACION ENTRE RUEDAS	16	10,99	10,99
RADIO RUEDA	2	3,09	3,09
DISTANCIA ENTRE EJES	3	6,09	6,09
SEPARACION ENTRE SENSORES	4	1,9	4

Tabla 6. Cambio en valores geométricos sobre búsqueda local 2 y 6 e instancia 1.

Como se puede observar, para la instancia 1 los mejores cambios son disminuir la separación entre ruedas y la separación entre sensores y aumentar el radio de las ruedas y la distancia entre ejes.

En la figura 15 se muestra la simulación del robot inicial y los robots obtenidos mediante **best** y **first**.



a) Primer robot b) Best c) First

Figura 15. Simulación de los robots de la tabla 6.

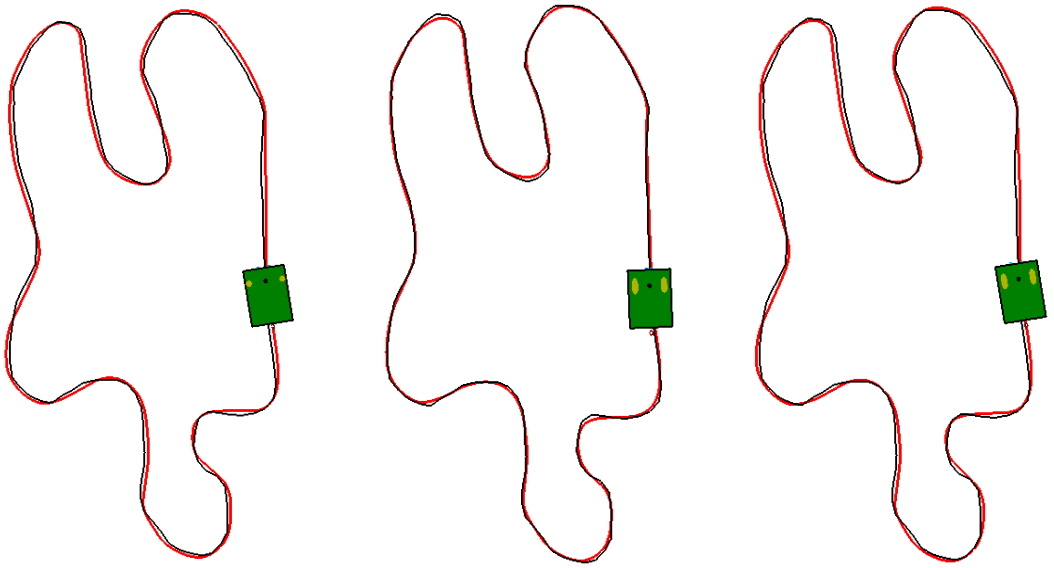
En la tabla 7 se muestra un robot inicial construido con el constructor del algoritmo y los valores obtenidos al ejecutar el mejor **best** (búsqueda local 4) y el mejor **first** (búsqueda local 8) sobre la instancia 8.

	INICIALES	**BEST**	**FIRST**
SEPARACION ENTRE RUEDAS	13,08	10,97	10,98
RADIO RUEDA	1,36	3,05	3,06
DISTANCIA ENTRE EJES	5,20	6,09	6,00
SEPARACION ENTRE SENSORES	4,13	1,93	4,13

Tabla 7. Cambio en valores geométricos sobre búsqueda local 4 y 8 e instancia 8.

Comparando los parámetros resultantes tanto de la tabla 6 como la 7, se puede deducir los parámetros óptimos del robot. Además, la diferencia que se da entre el robot optimo con **best** y **first** se haya en la separación entre sensores, la cual **best** disminuye y **first** no la cambia.

En la figura 16 se muestra la simulación de los robots mostrados en la tabla 7.



a) Primer robot b) Best c) First

Figura 16. Simulación de los robots de la tabla 7.

Por último, se muestra una tabla que indica el porcentaje de mejora del resultado en función de la búsqueda utilizada sobre cada una de las instancias.

		% Mejora
Best	LS 1	77
	LS 2	80
	LS 3	85
	LS 4	**88**
First	LS 5	67
	LS 6	69
	LS 7	72
	LS 8	74

Tabla 8. Porcentaje de mejora respecto a la primera solución.

En cuanto al tipo de búsqueda que mejor resultado proporciona es la búsqueda local 4 con un **88%** de mejora respecto al primer valor y la búsqueda que menos mejora es la 5 con un **67%** de mejora.

Al examinar los porcentajes de mejora en los respectivo a las instancias se observan grandes variaciones entre las mismas, desde un **53%** hasta un **127%**, lo que hace que se pueda confirmar que se han usado buenas instancias y se pueda verificar aun mas las pruebas realizadas.

Una vez vistos todos los datos se puede llegar a la conclusión de que si lo que se necesita es una mayor precisión en conseguir el óptimo, el mejor método es **best**, sin embargo, si se busca una relación de mejora-tiempo, **first** es un método mucho mejor puesto que encuentra una buena solución en un tiempo mucho menor.

Conclusiones

Gracias a este proyecto he indagado un poco mas en metaheurísticas, mas concretamente en las búsquedas locales y su gran variabilidad para obtener mejoras.

Lo mas complicado a la hora de realizar este proyecto ha sido la correcta implementación de la función objetivo, debido a que, si esta tenía algún fallo, todo el algoritmo dejaba de funcionar. Otra de las cosas que mas problemas han causado ha sido la decisión ha tomar sobre los robots infactibles, así como el método constructivo a utilizar.

El objetivo principal propuesto ha sido logrado con una mejora media del 76,5%.

En cuanto a los resultados obtenidos se puede concluir que **first** realiza un trabajo mas optimo puesto que la relación calidad tiempo es mucho mas grande que para **best**, sin embargo, si se desea un mejor resultado sin tener en cuenta el tiempo de ejecución, **best** funciona mejor.

Además de esto la implementación de ILS como prueba ha mostrado que implementar un algoritmo que trate de evitar los óptimos locales puede ser una buena manera de mejorar aun mas los resultados obtenidos.

Las posibles mejoras a realizar entran dentro de la mejora de este esté algoritmo, añadiendo búsquedas globales cuando te encuentras con un óptimo local o pensando diferentes métodos constructivos que mejoren el random.

Otra posible mejora es la utilización de otras metaheurísticas para estudiar la posible mejora de los tiempos de ejecución sobre estas.

Además, como posible mejora, se podría unir esta aplicación con el simulador en Qt y que automáticamente coja los datos del fichero creado por este algoritmo y ejecute la simulación.

Bibliografía

Robótica

1. Fernando Reyes Cortés (2018) *"Robótica. Control de robots manipuladores"*.
2. Alonzo Kelly (2013) *"Mobile Robotics: Mathematics, Models, and Methods"*.

Metaheurística

3. D. Henderson, S.H. Jacobson y W.A. Jonhson (2003) *"Handbook of metaheuristic"*
4. F. Glover y M. Laguna (1997) *"Tabu Seatch"*
5. P. Hansen y N. Mladenovìc (2003) *"Handbook of Metaheuristic"*
6. J. H. Holland (1975) *"Adaptation in Natural and Artificial Systems"*
7. F. Glover (1998) *"In Artificial Evolution, Lecture Notes in Computer Science"*
8. M. Dorigo (1992) *"Optimization, Learning and Natural Algorithms"*
9. C.G.E. Boender, A.H.G. Rinnooy, L. Stougie y G.T. Timmer (1982) *"Mathematical Programming"*

Java

10. Cay S. Horstmann (2013) *"Java SE 8 for the Really Imaptient"*
11. Documentación online: https://docs.oracle.com/javase/10/docs/api/overview-summary.html

Metodología ágil

12. Andrew Stellman y Jenifer Greene (2014) *"Learning Agile: Understanding Scrum, XP, Lean, and Kanban"*.
13. Tipos de metodologías resumidas: https://www.versionone.com/agile-101/agile-methodologies/