



**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

Tutor: Alberto Herrán González

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 con los que recorrer un circuito en el menor tiempo posible.

Se ha comenzado estudiando el funcionamiento del robot sobre el circuito, definiendo así la función objetivo del algoritmo. A continuación, se pasa al análisis del algoritmo utilizado describiendo por separado cada una de sus partes: métodos constructivos, generación de vecindarios y selección de la siguiente solución. Inmediatamente después se ha llevado a cabo la implementación del algoritmo. Finalmente se han generado diferentes tipos de instancias, y se han probado los algoritmos desarrollados sobre las mismas.

Finalmente, las mejores soluciones obtenidas se han probado sobre el robot mostrando la mejora en el funcionamiento del mismo respecto a la solución base proporcionada.

Índice

CAPÍTULO 1 INTRODUCCIÓN	9
1.1. MOTIVACIÓN	9
1.2. OBJETIVOS.....	10
1.3. ESTADO DEL ARTE	10
1.4. ESTRUCTURA DE LA MEMORIA	12
CAPÍTULO 2 DESCRIPCIÓN DEL PROBLEMA	13
2.1. DIRECCIONAMIENTO DIFERENCIAL	13
2.2. NAVEGACIÓN AUTÓNOMA	18
2.3. PROBLEMA DE OPTIMIZACIÓN.....	20
CAPÍTULO 3 DESCRIPCIÓN ALGORÍTMICA.....	23
3.1. BÚSQUEDA LOCAL	23
3.2. MÉTODOS CONSTRUCTIVOS	24
3.3. GENERACIÓN DE VECINDARIOS.....	25
3.4. ESTRATEGIAS DE EXPLORACIÓN	25
3.5. BÚSQUEDAS GLOBALES.....	26
CAPÍTULO 4 IMPLEMENTACIÓN	31
4.1. METODOLOGÍA.....	31
4.2. DISEÑO	32
CAPÍTULO 5 RESULTADOS	35
5.1. DESCRIPCIÓN DE LAS INSTANCIAS	35
5.2. MÉTODOS CONSTRUCTIVOS	37
5.3. BÚSQUEDAS LOCALES.....	38
5.4. CONVERGENCIA DEL ALGORITMO.....	40
5.5. RESULTADOS FINALES.....	41
CONCLUSIONES.....	44

Capítulo 1 Introducción

1.1. Motivación

Este trabajo nace motivado del problema que resulta al 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 la mostrada en la figura 1.

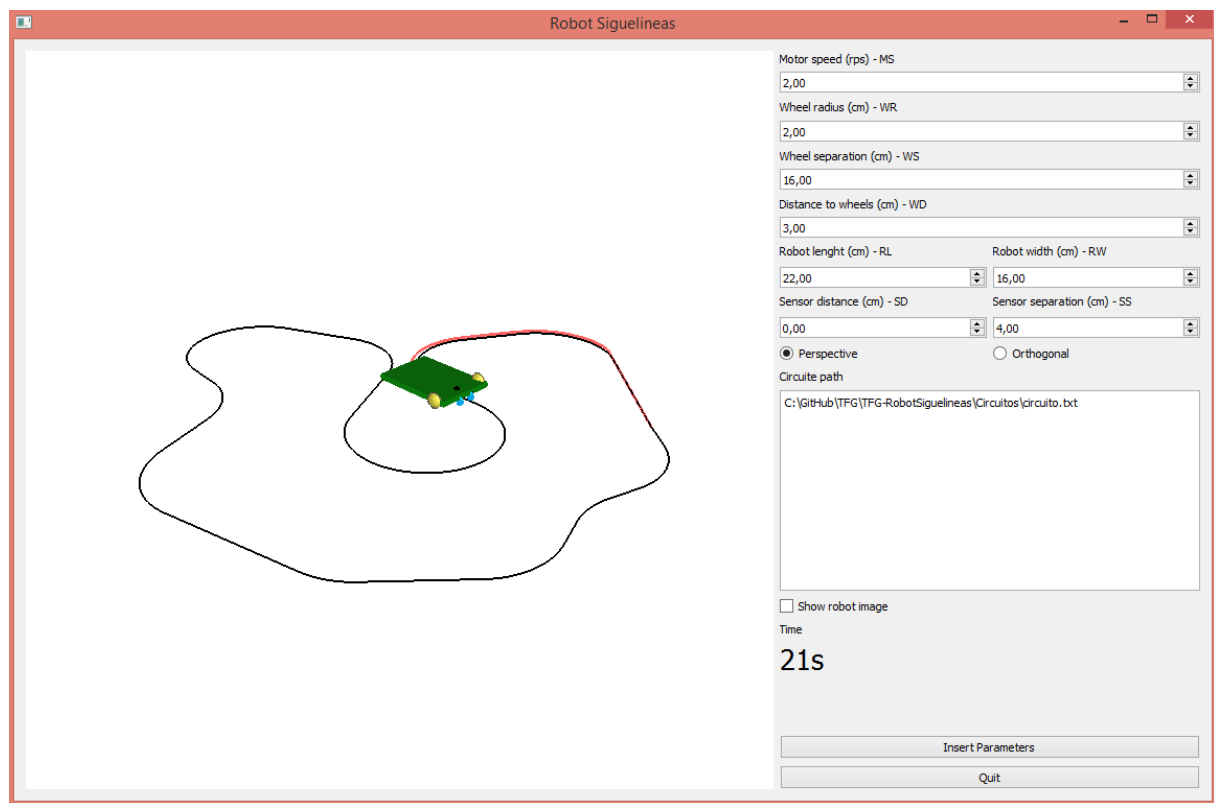


Figura 1. Aplicación de simulación de robot con direccionamiento diferencial.

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 sobre un robot real.

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

- Conseguir los mejores parámetros para un determinado circuito.
- Incluir límites 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]. Estos se pueden clasificar en dos categorías, las trayectoriales y las poblacionales. El aspecto diferenciador entre estas metaheurísticas es el número de soluciones que se utilizan en el proceso de optimización. Las trayectoriales se basan en utilizar 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 (LS, de Local Search):** 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 mejor vecino, la condición de finalización, etc.
- **Algoritmos voraces (Greedy algorithm en inglés):** se basa en la elección óptima en cada paso local, con la intención de conseguir así la solución

general óptima. En caso de que la primera elección óptima no sea factible, se descarta el elemento.

- **Algoritmo temple simulado (SA, de Simulated Annealing):** es un tipo de búsqueda global capaz de escapar de mínimos locales admitiendo en ciertas situaciones peores soluciones a la actual durante la búsqueda [4].
- **Búsqueda Tabú (TS, de Tabu Search):** 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. Seleccionar otra solución ocurre siempre después de analizar lo ocurrido anteriormente [5].
- **Búsqueda por entornos variables (VNS, de Variable Neighborhood Search):** existen muchas variaciones de esta metaheurística, la más básica realiza una búsqueda local hasta que se encuentra con un óptimo local, momento en el cual cambia de vecindario para tratar de escapar de ese óptimo local [6].

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

- **Algoritmos genéticos (GA, de Genetic Algorithms):** 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, mantiene 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 [7].
- **Búsqueda dispersa (SS, de 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 [8].
- **Algoritmo de la colonia de hormigas (ACO, de 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" [9].

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 describe la formulación matemática del problema sobre un robot con direccionamiento diferencial, es decir se deducen las ecuaciones necesarias para evaluar el comportamiento del sistema. 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 utilizada y se pasa a describir su aplicación: métodos constructivos y generación 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 describe 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 obtenidos en cuanto a tiempo de ejecución, costes y soluciones finales, obtenidos sobre las diferentes instancias y por las diferentes versiones de las búsquedas locales desarrolladas.

Finalmente se muestran las **Conclusiones**, donde presentan tanto las conclusiones como las posibles mejoras y ampliaciones futuras.

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 aquí considerado 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 permita 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.

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

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

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

En la figura 3 se pueden observar tanto el estado del sistema como sus entradas sobre el robot en cuestión.

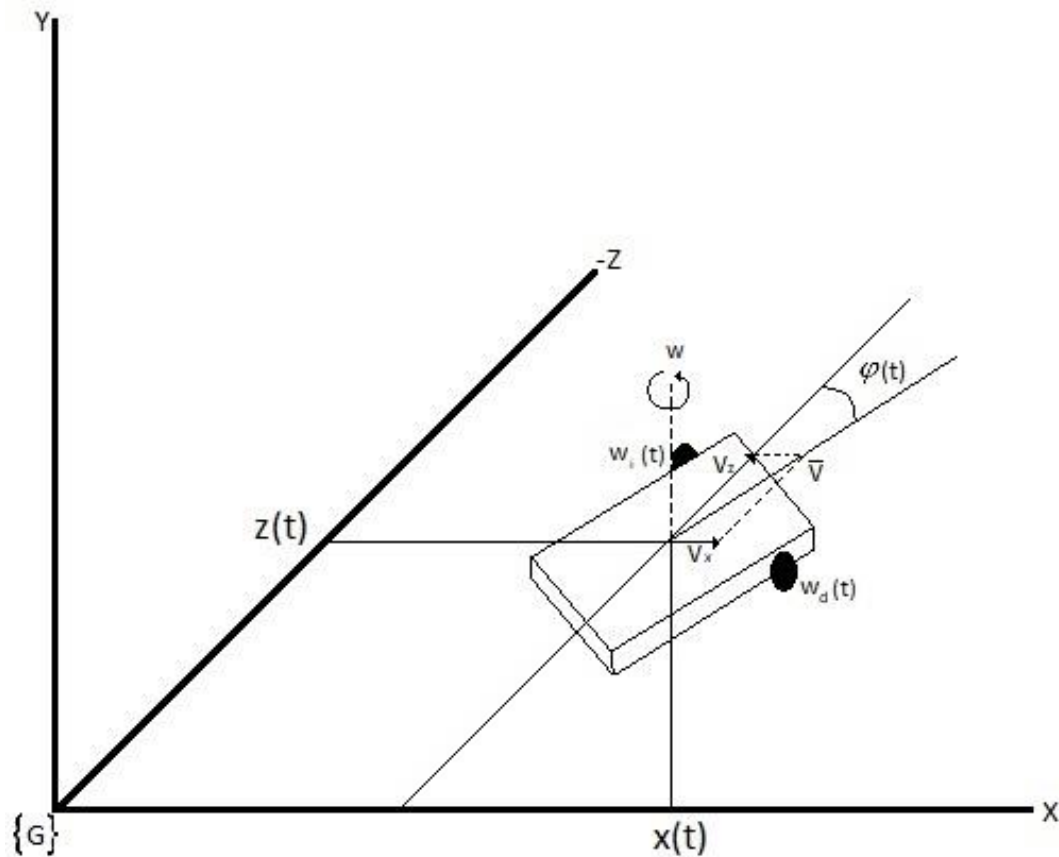


Figura 3. Estado y entradas de un robot con direccionamiento diferencial.

A partir de la figura 3, 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 $x(t)$ mediante la ecuación:

$$x(t + \Delta t) = x(t) + v_x \cdot \Delta t \quad (5)$$

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

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

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 (w_i, 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 $v_d = w_d \cdot WR$. En este caso, la velocidad lineal y velocidad angular correspondientes en el modelo vienen dadas por:

$$\begin{aligned} v &= \frac{v_d + v_i}{2} = \frac{(v_d + v_i) \cdot WR}{2} \\ w &= \frac{v_d - v_i}{WS} = \frac{(w_d - w_i) \cdot WR}{WS} \end{aligned} \quad (7)$$

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:

$$\begin{aligned} v_x &= \frac{-(w_d + w_i) \cdot WR}{2} \cdot \sin(\varphi) = -(w_d + w_i) \cdot \frac{WR \cdot \sin(\varphi)}{2} \\ v_z &= \frac{(w_d + w_i) \cdot WR}{2} \cdot \cos(\varphi) = (w_d + w_i) \cdot \frac{WR \cdot \cos(\varphi)}{2} \\ v_\varphi &= \frac{(w_d - w_i) \cdot WR}{WS} = (w_d - w_i) \cdot \frac{WR}{WS} \end{aligned} \quad (8)$$

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

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

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:

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

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

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

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 cada instante de tiempo. Las variables necesarias para poder realizar dicho calculo 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

La tarea que debe realizar el robot consiste en recorrer un circuito sin salirse del mismo. Este circuito se marca con una línea negra sobre un fondo de color blanco, como se muestra en la figura 4. El robot sigue líneas que se ha implementado realiza su movimiento de manera autónoma, esto se puede realizar gracias a dos sensores colocados 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.

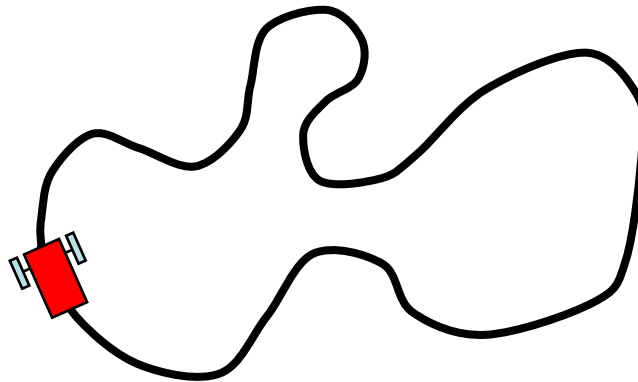


Figura 4. Navegación autónoma del robot sigue líneas.

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

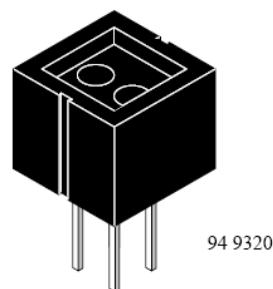


Figura 5. Sensor CNY70.

Se trata de 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 envía la señal correspondiente al circuito de control. Sin embargo, cuando se encuentra sobre fondo blanco la luz es reflejada y por lo tanto el fototransistor envía la señal contraria a la enviada en caso de estar sobre negro.

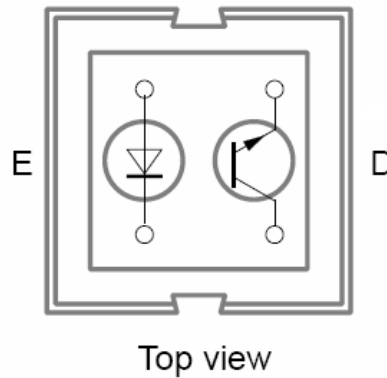


Figura 6. Estructura simplificada del sensor CNY70.

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. $W_i \leftarrow 0$
 3. **if** $IsOnCircuite(s_d)$
 4. $W_d \leftarrow 0$
-

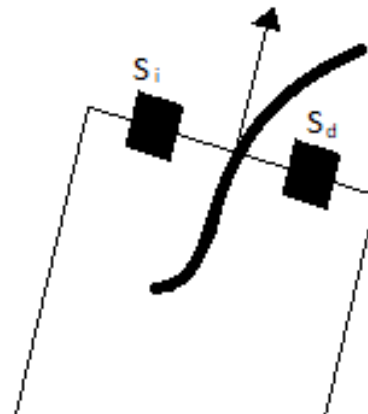


Figura 7. Posición sensores sobre robot.

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

2.3. Problema de optimización

De todos los parámetros geométricos del robot, los parámetros variables que se pretenden optimizar son: la separación entre las ruedas (WS), el radio de las ruedas (WR), la distancia entre los ejes (distancia desde el centro de los sensores hasta el punto de rotación del robot, definido como AD) y la separación entre los sensores (SS). Estos parámetros se pueden observar sobre un robot en la figura 8.

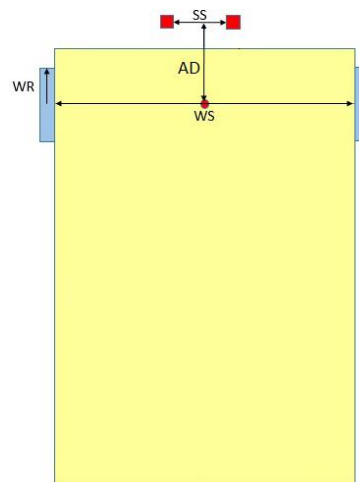


Figura 8. Parámetros geométricos a optimizar.

El problema aquí descrito puede plantearse como un problema de optimización, como el mostrado en la ecuación (12).

$$\begin{aligned} \min_x \quad & f(x) \\ \text{s.t.} \quad & g(x) \leq 0 \\ & x \in \mathbb{R} \end{aligned} \tag{12}$$

En particular para este problema las variables de decisión se definen como se muestra en la ecuación 13:

$$x = [WD, WR, AD, SS] \tag{13}$$

En nuestro caso, la función objetivo $f(x)$ viene dada por el tiempo utilizado por el robot para completar una vuelta del circuito. Dicho tiempo se calcula a partir de una simulación del robot recorriendo el circuito. Para esta simulación es necesario utilizar las ecuaciones (9) y los algoritmos de navegación autónoma para determinar la posición, rotación y movimiento del robot.

Finalmente, la función $g(x)$ representa las restricciones del problema y, en nuestro caso, simplemente se utiliza para imponer ciertos límites en las variables contenidas en el vector de decisión, tal y como se muestra en la ecuación (14).

$$\begin{aligned} 10,8 & \leq WD \leq 21,1 \\ 0,9 & \leq WR \leq 3,2 \\ 0 & \leq AD \leq 6,2 \\ 1,9 & \leq SS \leq 6,2 \end{aligned} \tag{14}$$

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.

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

El pseudocódigo asociado a esta metaheurística es el siguiente:

El algoritmo comienza construyendo una solución de partida, x , con alguno de los métodos disponibles para tal fin (paso 1). A continuación, el algoritmo comienza la búsqueda de la solución óptima realizando en bucle los siguientes pasos. En primer lugar, actualiza el mejor valor hasta el momento, x^* . A continuación, se genera el vecindario, N , de la mejor solución actual (paso 4). El último paso reside en la selección de la siguiente solución dentro del vecindario (paso 5). Una vez obtenida la nueva solución se comprueba si mejora la mejor solución actual, x^* , en caso afirmativo se vuelve al paso 3, si no la mejora se devuelve x^* como mejor solución encontrada.

Algoritmo 3: Búsqueda local

1. $x \leftarrow \text{Constructivo}()$
 2. **do**
 3. $x^* \leftarrow x$
 4. $N \leftarrow \text{Neighborhood}(x^*)$
 5. $x \leftarrow \text{Select}(N, x^*)$
 6. **while** $f(x) < f(x^*)$
 7. **return** x
-

3.2. Métodos constructivos

El método constructivo que se ha utilizado se basa en la generación de unos valores aleatorios con límites 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 se ejecuta varias veces hasta obtener uno que sea factible.

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

Lo primero que realiza este algoritmo es la generación de un robot aleatorio cuyos parámetros geométricos estén dentro de los límites establecidos. Una vez generado el robot, se utiliza la función objetivo para determinar el tiempo que tarda en realizar el circuito. En caso de que este tiempo sea mayor que un umbral, significa que el robot no ha podido realizar el circuito y por lo tanto no es un robot factible, por lo que se tendría que repetir el proceso hasta encontrar un robot factible. En caso de que no sobrepase el umbral, el robot es factible y, por lo tanto, se devuelve como solución inicial.

Algoritmo 4: Constructivo

1. **do**
 2. $x \leftarrow \text{RobotRandom}()$
 3. $\text{tiempo} \leftarrow f(x)$
 4. **while** $\text{tiempo} > \text{umbral}$
 5. **return** robot
-

Algoritmo 5: RobotRandom()

1. $WD \leftarrow \text{Random}(11.1, 21.1)$
 2. $WR \leftarrow \text{Random}(0.9, 3.1)$
 3. $AD \leftarrow \text{Random}(0, 6.1)$
 4. $SS \leftarrow \text{Random}(1.9, 4.1)$
 5. **return** $\text{new Robot}(WD, WR, AD, SS)$
-

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 este trabajo se han desarrollado diferentes heurísticas para la generación de vecinos. Debido a que se trata de valores reales, se ha optado por la permutación de determinadas variables en diferentes grados.

La opción elegida para realizar la búsqueda de vecinos se basa en sumar una magnitud, Δx , a dos parámetros y restársela a un tercero. Aplicando esto a las diferentes combinaciones de tres parámetros resulta una vecindad de 24 vecinos.

La magnitud que se suma o resta a los parámetros puede tomar 4 valores distintos, dando como resultado 4 búsquedas locales diferentes que poder comparar.

3.4. Estrategias de exploración

Una vez tenemos el vecindario generado se debe escoger el vecino que se convertirá la siguiente solución. Existen diferentes formas de realizar tal selección, algunas de ellas son:

En la estrategia **first** se escoge el primer vecino que mejore a la solución actual. El pseudocódigo asociado a esta búsqueda es el siguiente:

Este algoritmo itera sobre cada uno de los vecinos. Por cada vecino mira si su tiempo mejora a la solución actual (paso 2), en caso afirmativo se devuelve el vecino como nueva solución y se finaliza el algoritmo.

Algoritmo 6: Select.First(N, x^*)

1. **foreach** x **in** N
 2. **if** $f(x) < f(x^*)$
 3. **return** x
 4. **endif**
 5. **endforeach**
-

En la estrategia **best**, se escoge al mejor vecino de la vecindad. La implementación en pseudocódigo es la siguiente:

Para comenzar se guarda la mejor solución actual, x^* , como mejor solución temporal, x' (paso 1). A continuación, se itera sobre cada uno de los vecinos, x , del vecindario, N . Para cada uno de ellos se comprueba si mejora la mejor solución temporal (paso 3), si es así se reemplaza la mejor solución temporal por el vecino actual, de no ser así se continúa iterando sobre cada vecino. Cuando se ha terminado de comprobar todos los vecinos se devuelve x' como mejor solución de la vecindad.

Algoritmo 7: Select.Best(N, x^*)

```
1.  $x' = x^*$ 
2. foreach  $x$  in  $N$ 
3.   if  $f(x') > f(x)$ 
4.      $x' \leftarrow x$ 
5. endforeach
6. return  $x'$ 
```

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 desde su vecindario y que, además, no es la solución óptima del problema. Las metaheurísticas que tratan de evitar este problema tratan de llevar a cabo por tanto una búsqueda global.

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

- **Búsqueda Multiarranque:** 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 [10].

Este algoritmo comienza creando una solución, x (paso 1). A continuación, se ejecuta el algoritmo llamado MejoraLS(x), este algoritmo ejecuta los pasos 2 a 7 del algoritmo 3. Después se comprueba si la solución temporal, x' , obtenida por el algoritmo del paso 3 es mejor que la solución actual, x^*

(paso 3). En caso afirmativo se sustituye la mejor solución por la solución temporal. Por último, se entra en un bucle en el que se realizan los mismos pasos descritos hasta ahora: constructivo, MejoraLS, comprobación de si es mejor, asignación en caso afirmativo. Este algoritmo finaliza tras pasar un tiempo de ejecución máximo (paso 6).

Algoritmo 8: Multiarranque

```
1.  $x^* \leftarrow \text{Constructivo}()$ 
2.  $x' \leftarrow \text{MejoraLS}(x^*)$ 
3. if  $f(x') < f(x^*)$ 
4.    $x^* \leftarrow x'$ 
5.  $t \leftarrow \text{CPU}()$ 
6. while  $t < t_{\max}$  do
7.    $x' \leftarrow \text{Constructivo}()$ 
8.    $x'' \leftarrow \text{MejoraLS}(x')$ 
9.   if  $(x'') < f(x^*)$ 
10.     $x^* \leftarrow x''$ 
11.    $t \leftarrow \text{CPU}$ 
12. endwhile
```

- **Búsqueda local iterada (ILS, de Iterated Local Search):** Cuando se encuentra con un óptimo local reinicia la búsqueda desde una perturbación de la solución actual.

Este algoritmo realiza lo mismo que el algoritmo explicado anteriormente a excepción del paso 7, en el cual, en vez de reiniciarse la búsqueda como pasa en el algoritmo 8, se continua la búsqueda, pero desde una solución vecina de la actual o desde una perturbación de la misma.

Algoritmo 9: ILS

```
1.  $x^* \leftarrow \text{Constructivo}()$ 
2.  $x' \leftarrow \text{MejoraLS}(x^*)$ 
3. if  $f(x') < f(x^*)$ 
4.    $x^* \leftarrow x'$ 
5.  $t \leftarrow \text{CPU}()$ 
6. while  $t < t_{\max}$  do
7.    $x' \leftarrow \text{Perturbar}(x^*)$ 
8.    $x'' \leftarrow \text{MejoraLS}(x')$ 
9.   if  $f(x'') < f(x')$ 
10.     $x^* \leftarrow x''$ 
11.    $t \leftarrow \text{CPU}$ 
12. endwhile
```

- **Búsqueda por entornos variables (VNS, de 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.

El pseudocódigo mostrado a continuación es el de un algoritmo BVNS (Basic VNS). Este algoritmo comienza creando una solución (paso 1). A continuación, se realiza una perturbación de la mejor solución actual, x^* , en función de la iteración, k (paso 5). Cuanto mayor sea k mayor será la perturbación. Después de esto se realiza la mejora de la solución temporal (paso 6) y se compara el resultado con la mejor solución (paso 7), en caso de ser mejor se convierte en la mejor solución y se resetea el contador de la iteración, k ; en caso negativo, se incrementa la iteración. Se repite el proceso desde la perturbación (paso 5) hasta que se llega a un máximo de iteraciones. Si se ha llegado al máximo de iteraciones se reactiva la búsqueda de la solución con un valor de iteración de 1. El algoritmo finaliza pasado un límite de tiempo de ejecución.

Algoritmo 10: BVNS(t_{\max}, k_{\max})

```

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

```

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

El pseudocódigo mostrado a continuación muestra la implementación de un algoritmo de temple simulado. Tras los constructores y la inicialización de c , comienza el bucle en el que se trata de mejorar la solución. Se utiliza el algoritmo Cicle para intentar encontrar una solución mejor (paso 5), este algoritmo es explicado más adelante. En caso de que la solución proporcionada por Cicle, x' , mejore la mejor solución actual, x^* , se actualiza la mejor solución actual con la encontrada por Cicle (paso 7). A continuación, se actualiza c a partir de el mismo y α . Después de esto, se reinicia el bucle.

El algoritmo Cicle escoge L vecinos de la solución y les aplica un criterio de aceptación, en caso de que lo pasen la mejor solución temporal, x , pasa a ser el vecino que lo ha pasado. Este criterio de aceptación determina la probabilidad con la que se acepta el vecino a partir de la mejor solución

actual. Si el vecino es mejor que la mejor solución actual, la probabilidad de ser aceptado es de 1, en caso contrario la probabilidad se calcula como $e^{\frac{-\Delta f(x',x)}{c}}$ dónde $\Delta f(x',x) = f(x) - f(x')$.

Algoritmo 11: SA(α, C_0, L)

1. $x^* \leftarrow \text{Constructivo}()$
 2. $c \leftarrow C_0$
 3. $t \leftarrow \text{CPU}()$
 4. **while** $t < t_{\max}$ **do**
 5. $x' \leftarrow \text{Cicle}(x^*, c, L)$
 6. **if** $f(x') < f(x^*)$
 7. $x^* \leftarrow x'$
 8. $c = \alpha \cdot c$
 9. **endwhile**
 10. **return** x^*
-

Algoritmo 12: Cicle(x, c, L)

1. **for** $l \leftarrow 1$ **to** L
 2. $x' \leftarrow \text{Perturbar}(x)$
 3. **if** $\text{CriterioAceptacion}(x', x, c)$
 4. $x \leftarrow x'$
 5. **endfor**
 6. **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 [11].

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.

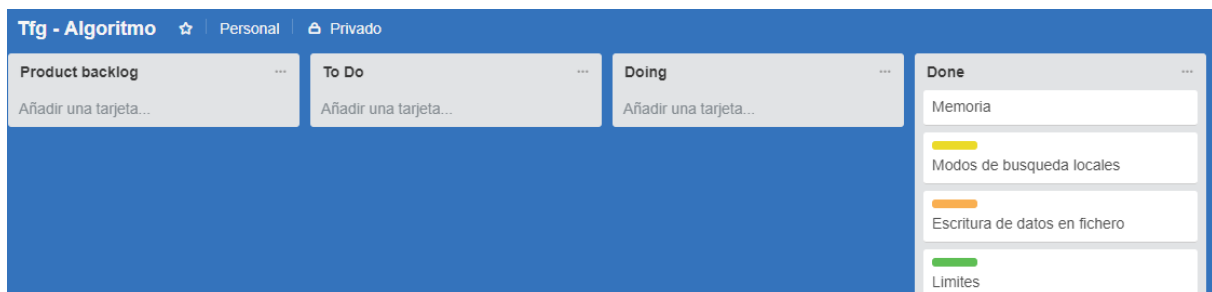


Figura 9. Trello utilizado en el desarrollo del proyecto.

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.

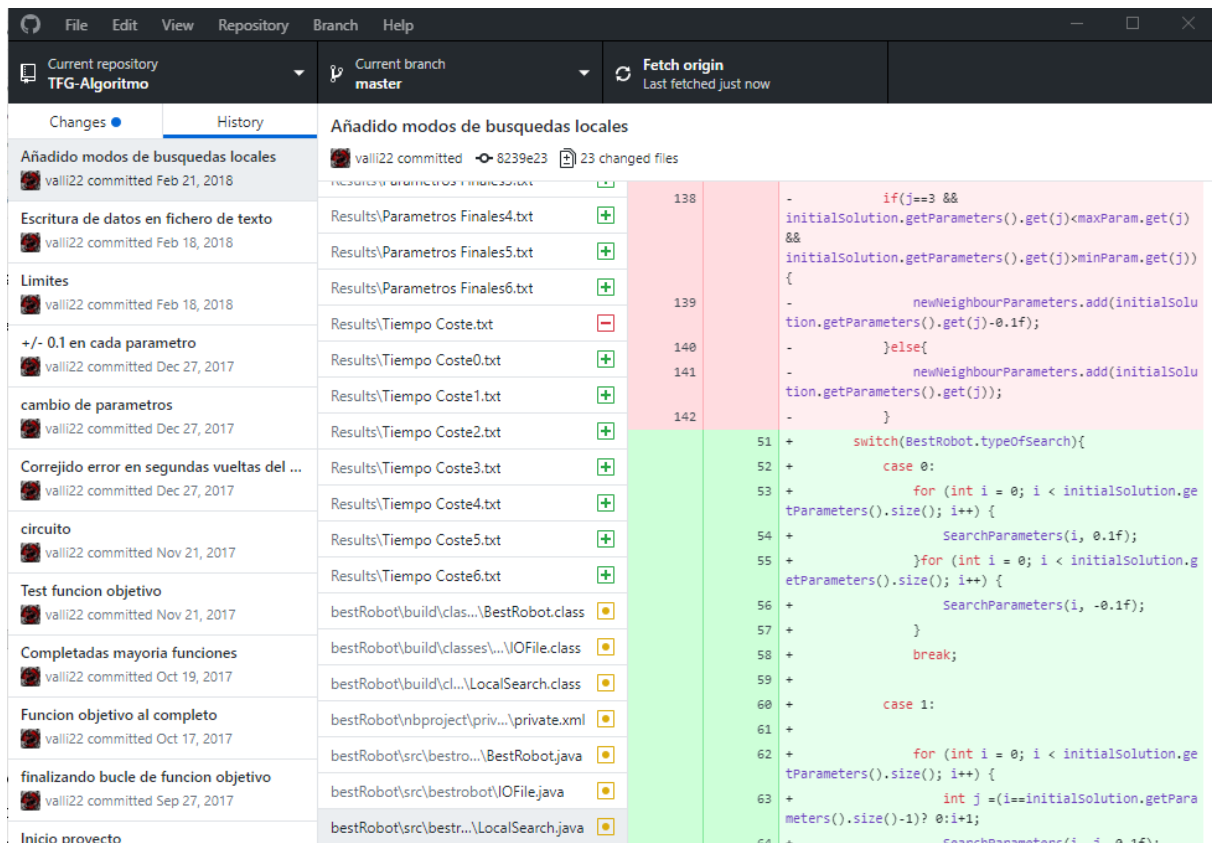


Figura 10. Repositorio en GitHub desde la aplicación de escritorio de Windows.

4.2. Diseño

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

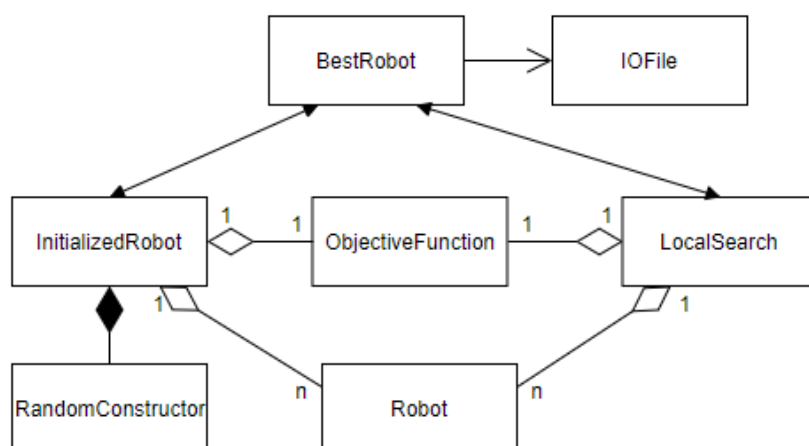
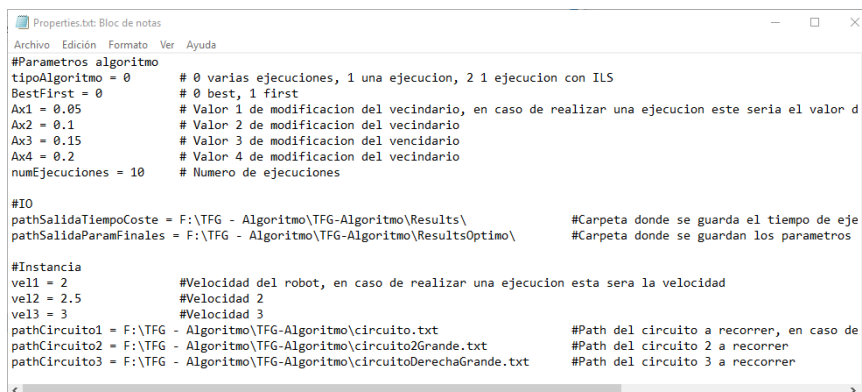


Figura 11. UML

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.

Para ejecutar el algoritmo primero se debe acceder al fichero de texto Properties y cambiar ahí lo que se desee hacer, este fichero se puede observar en la figura 12.



```

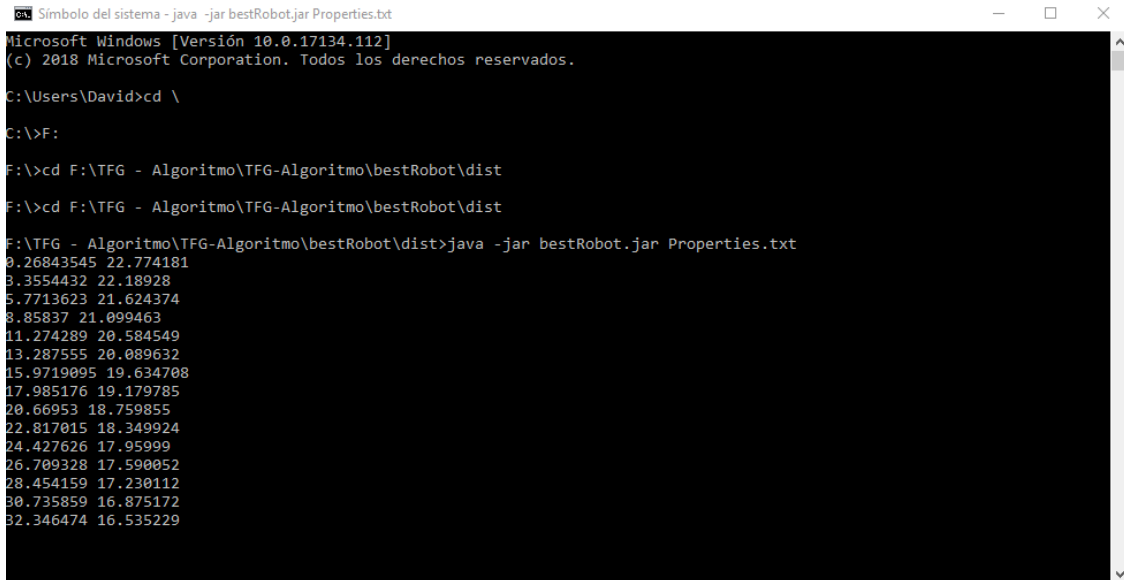
#Parametros algoritmo
tipoAlgoritmo = 0      # 0 varias ejecuciones, 1 una ejecucion, 2 1 ejecucion con ILS
BestFirst = 0          # 0 best, 1 first
Ax1 = 0.05             # Valor 1 de modificacion del vecindario, en caso de realizar una ejecucion este seria el valor d
Ax2 = 0.1              # Valor 2 de modificacion del vecindario
Ax3 = 0.15             # Valor 3 de modificacion del vecindario
Ax4 = 0.2              # Valor 4 de modificacion del vecindario
numEjecuciones = 10    # Numero de ejecuciones

#IO
pathSalidaTiempoCoste = F:\TFG - Algoritmo\TFG-Algoritmo\Results\          #Carpeta donde se guarda el tiempo de eje
pathSalidaParamFinales = F:\TFG - Algoritmo\TFG-Algoritmo\ResultsOptimo\  #Carpeta donde se guardan los parametros

#Instancia
vel1 = 2               #Velocidad del robot, en caso de realizar una ejecucion esta sera la velocidad
vel2 = 2.5             #Velocidad 2
vel3 = 3               #Velocidad 3
pathCircuito1 = F:\TFG - Algoritmo\TFG-Algoritmo\circuito.txt             #Path del circuito a recorrer, en caso de
pathCircuito2 = F:\TFG - Algoritmo\TFG-Algoritmo\circuito2Grande.txt       #Path del circuito 2 a recorrer
pathCircuito3 = F:\TFG - Algoritmo\TFG-Algoritmo\circuitoDerechaGrande.txt #Path del circuito 3 a recorrer
  
```

Figura 12. Properties.txt

Cada parámetro de este fichero esta comentado para saber que es lo que se está cambiando al modificarlo. Una vez se ha modificado el fichero como se desea se debe ejecutar por línea de comandos él .jar de la aplicación con el fichero Properties como argumento. La línea de comandos seria la siguiente: "java -jar bestRobot.jar Properties.txt". Esto se puede observar en la figura 13.



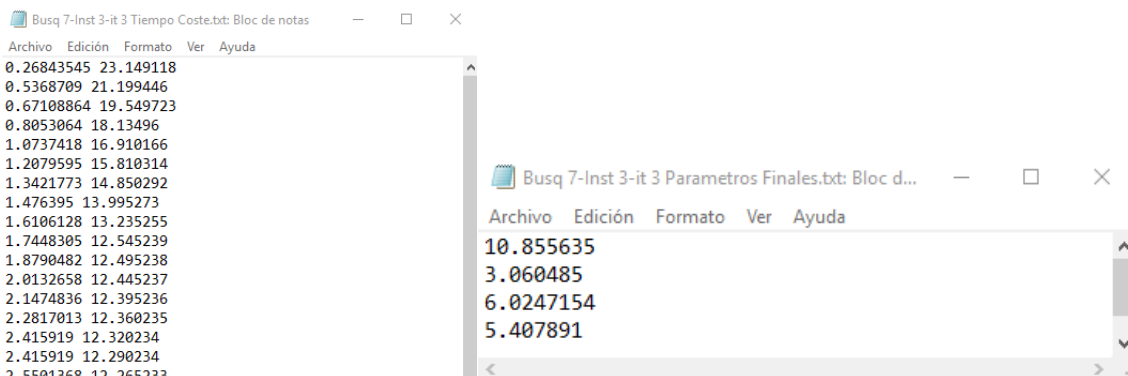
```

C:\Users\David>cd \
C:\>F:
F:\>cd F:\TFG - Algoritmo\TFG-Algoritmo\bestRobot\dist
F:\>cd F:\TFG - Algoritmo\TFG-Algoritmo\bestRobot\dist
F:\TFG - Algoritmo\TFG-Algoritmo\bestRobot\dist>java -jar bestRobot.jar Properties.txt
0.26843545 22.774181
3.3554432 22.18928
5.7713623 21.624374
8.85837 21.099463
11.274289 20.584549
13.287555 20.089632
15.9719095 19.634708
17.985176 19.179785
20.66953 18.759855
22.817015 18.349924
24.427626 17.95999
26.709328 17.590052
28.454159 17.230112
30.735859 16.875172
32.346474 16.535229

```

Figura 13. Ejecución del programa.

Una vez se ha acabado de ejecutar la aplicación, se podrá encontrar en la carpeta que se indico en Properties los resultados de la ejecución del algoritmo. En la figura 14 se puede ver los dos ficheros resultantes de la ejecución de la aplicación, en la figura 14.a) se puede observar el fichero en el que se encuentra el tiempo de ejecución y el coste de los diferentes robots y en el fichero 14.b) los parámetros del mejor robot obtenido.



```

Busq 7-Inst 3-it 3 Tiempo Coste.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
0.26843545 23.149118
0.5368709 21.199446
0.67108864 19.549723
0.8053064 18.13496
1.0737418 16.910166
1.2079595 15.810314
1.3421773 14.850292
1.476395 13.995273
1.6106128 13.235255
1.7448305 12.545239
1.8790482 12.495238
2.0132658 12.445237
2.1474836 12.395236
2.2817013 12.360235
2.415919 12.320234
2.415919 12.290234
2.5501368 12.265233

Busq 7-Inst 3-it 3 Parametros Finales.txt: Bloc d...
Archivo Edición Formato Ver Ayuda
10.855635
3.060485
6.0247154
5.407891

```

a) Tiempo-Coste

b) Parámetros finales

Figura 14. Ficheros de salida del 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 15.

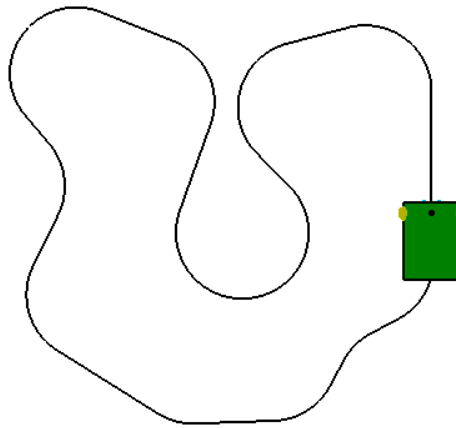


Figura 15. Circuito 1 estándar

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 16.

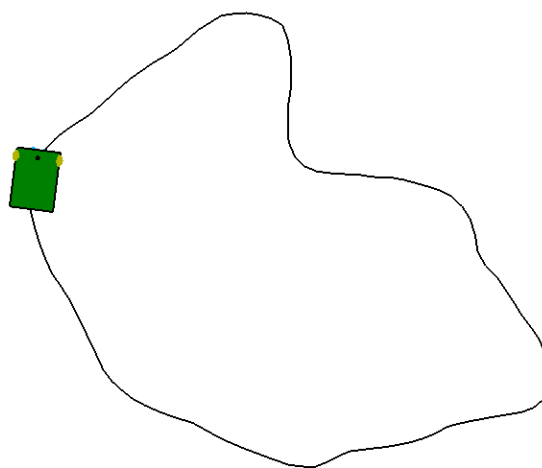


Figura 16. Circuito 2 grande sin curvas

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 17.

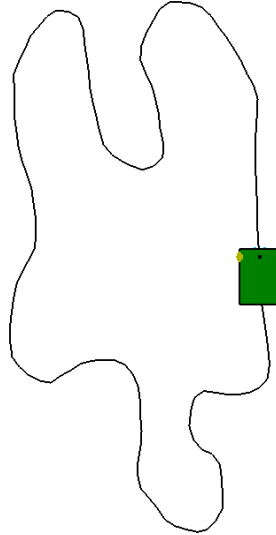


Figura 17. Circuito 3 tamaño medio con curvas cerradas

Las velocidades del robot que se utilizan son 2.0, 2.5 y 3.0 rps.

CIRCUITO VEL. RUEDAS	CIRCUITO 1	CIRCUITO 2	CIRCUITO 3
	Instancia 1	Instancia 4	Instancia 7
2.0RPS			
2.5RPS	Instancia 2	Instancia 5	Instancia 8
3.0RPS	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 variable del vector de decisión, es decir, cada una de ellas tiene unos límites adecuados para su geometría real.

A continuación, se muestra una tabla en la que se expone el resultado medio de la ejecución del constructivo 10 veces sobre cada instancia, en donde la columna 1 especifica la instancia, las columnas 2 y 3 sus características y, finalmente, la columna 4 el coste de la solución que viene dado por el tiempo que tarda el robot en completar una vuelta al circuito medido en segundos.

Instancia	Circuito	Velocidad (rps)	Coste (s)
I1	C1	2.0	33.87
I2	C1	2.5	23.72
I3	C1	3.0	26.13
I4	C2	2.0	28.53
I5	C2	2.5	53.17
I6	C2	3.0	20.44
I7	C3	2.0	41.56
I8	C3	2.5	30.63
I9	C3	3.0	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

En este experimento se va a utilizar los métodos de selección **best** y **first**, así como 4 valores, {0.05, 0.1, 0.15, 0.2}, para el parámetro de modificación de vecinos, Δx , visto en el punto 3.3. La combinación de estos genera 8 posibles búsquedas locales. Se puede observar dichas combinaciones en la tabla 3.

Búsqueda local	Estrategia	Δx
LS1	Best	0.05
LS2	Best	0.10
LS3	Best	0.15
LS4	Best	0.20
LS5	First	0.05
LS6	First	0.10
LS7	First	0.15
LS8	First	0.20

Tabla 3. Combinaciones de búsquedas locales

A continuación, se ha ejecutado el algoritmo utilizando todas las búsquedas locales e instancias. Para dotar de mayor robustez a los resultados obtenidos, cada experimento se ha ejecutado 10 veces, promediándose además los resultados para todas las instancias. Se pueden observar los resultados en la tabla 4, donde la columna 1 indica la búsqueda local utilizada, la columna 2 el coste de la solución que viene dado por el tiempo que tarda un robot en dar una vuelta al circuito medido en segundos, la tercera columna es el tiempo, en segundos, que el algoritmo ha tardado en encontrar la solución óptima, la columna 4 hace referencia a la diferencia entre el mejor coste obtenido y el dado por el algoritmo correspondiente y la última columna es el número instancias en el que el algoritmo ha dado el mejor resultado.

Búsqueda local	Coste (s)	TiempoCPU (s)	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	0.00	7
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 de ejecución entre las búsquedas con **best** y **first**, siendo bastante menor las segundas. Además de esto, también se puede observar como el aumento de Δx implica una mejora en el coste de las soluciones obtenidas, siendo la mejor búsqueda local **LS4**.

También se puede comprobar como al aumentar Δx se encuentra el mejor valor de manera más rápida.

Una vez analizados los resultados de las búsquedas locales, vamos a comparar de forma más 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 un **25.83%** más en obtenerlos. Estos porcentajes se han obtenido de promediar el coste y el tiempo de CPU de **first** y **best** y con esos valores obtener el porcentaje.

En lo relativo a Δx , su aumento implica una mejora tanto del valor obtenido como del tiempo de ejecución. En cuanto a la mejora obtenida, por cada aumento de Δx 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. Los datos obtenidos en la tabla resultan de obtener el promedio de cada valor de Δx y realizar

el porcentaje con cada una de sus etapas, es decir el porcentaje de $\Delta x = 0.05$ con respecto a $\Delta x = 0.1$, etc.

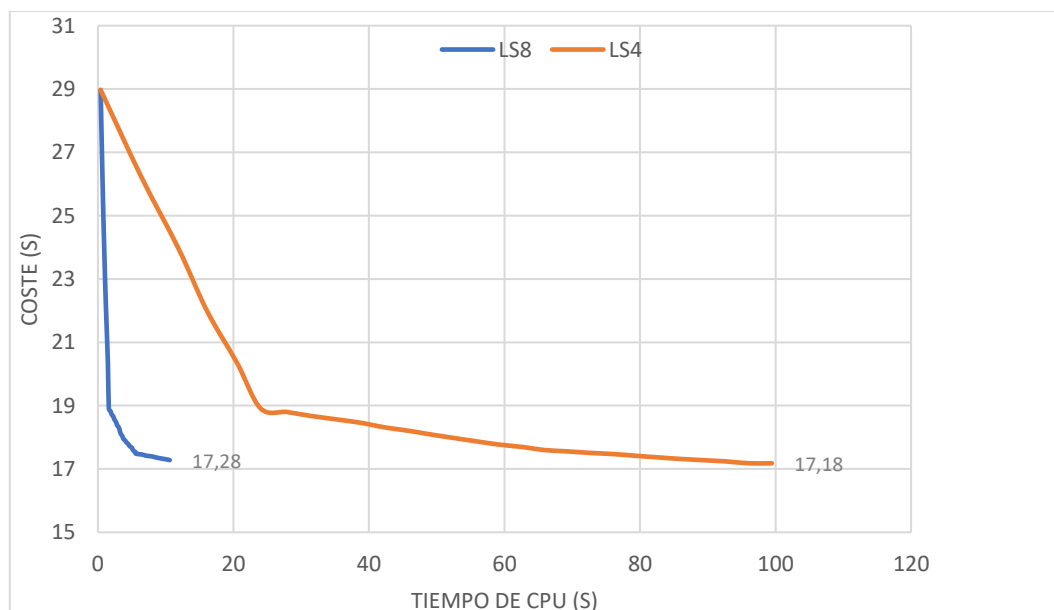
Δx	BEST	FIRST
0.05-0.1	93%	25%
0.1-0.15	15%	25%
0.15-0.2	17%	-2%

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

5.4. Convergencia del algoritmo

En este apartado se analiza la convergencia de las diferentes versiones del algoritmo desarrollado.

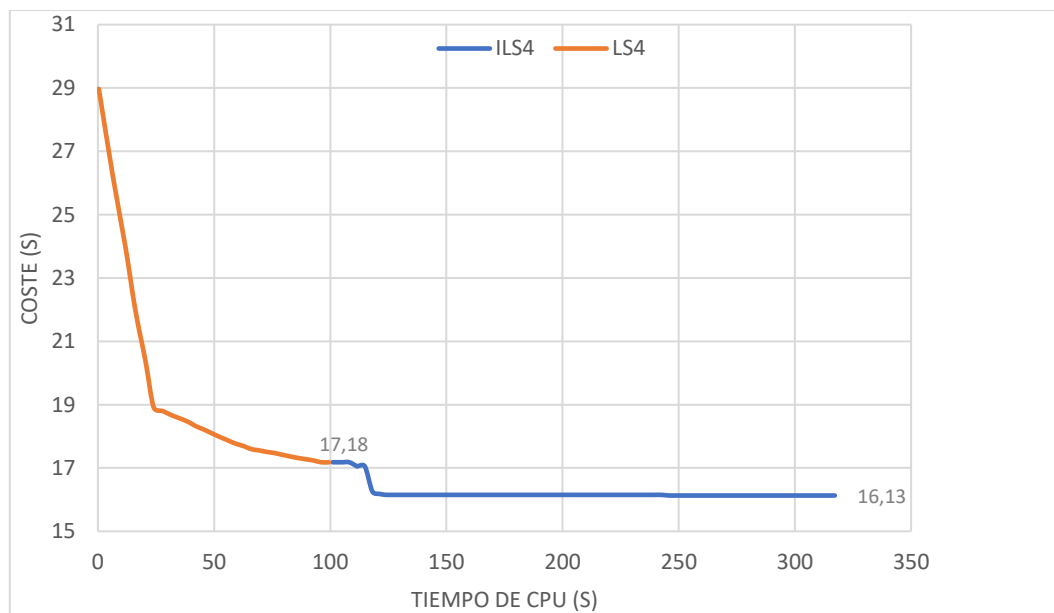
En la siguiente gráfica se ha modificado el constructor para que **best** y **first** empiecen desde el mismo robot inicial y se ha utilizado el mejor algoritmo que utiliza **best** LS4 y el mejor algoritmo de **first** LS8, esto se ha ejecutado sobre la instancia 1.



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

En esta gráfica se puede confirmar lo dicho anteriormente, **first** obtiene un valor un poco peor pero mucho más rápido que **best**.

Por último, se ha probado a utilizar **ILS** para intentar escapar de los óptimos locales. La ejecución con **ILS** se ha realizado sobre la mejor búsqueda local encontrada: LS4.



Gráfica 3. Misma ejecución que gráfica 2 aplicando ILS.

Tras utilizar **ILS** se puede comprobar cómo se puede obtener un mejor valor a cambio de tiempo de ejecución. En esta gráfica se puede comprobar como gracias a **ILS** se escapa del óptimo local, sin embargo, parece atascarse en otro tras haber optimizado el coste en gran medida. Si el algoritmo se deja más tiempo cabe la posibilidad de que **ILS** consiga escapar de este óptimo local y acceder a un mejor valor.

5.5. Resultados finales

Una vez comprobados los mejores métodos, se comprueban los cambios realizados en los valores geométricos del robot con la mejor búsqueda local, LS4.

En la tabla 6 se muestran los costes de un robot construido y optimizado a mano por un humano y los costes obtenidos al ejecutar LS4 para cada una de las instancias.

Instancia	Humano	LS4
I1	28.97	16.63
I2	23.17	13.57
I3	19.31	11.13
I4	26.21	16.03
I5	20.98	12.74
I6	17.49	11.22
I7	34.13	20.64
I8	27.31	16.42
I9	22.77	13.42
Media	25.91	14.64

Tabla 6. Costes de robots optimizados por humanos y por LS4.

En la siguiente tabla se muestran los valores geométricos de los robots optimizados tanto por una persona humana como por el mejor algoritmo obtenido LS4. En cuanto a los robots optimizados por personas, para el circuito 1 y 2 se ha optimizado con el mismo robot, sin embargo, el robot optimizado para el circuito 3 es distinto debido a que el robot que se utiliza en el robot 1 y 2 es infactible en este circuito.

Parámetros geométricos	Humano Circuito 1 y 2	Humano Circuito 3	LS4 I1	LS4 I2	LS4 I3	LS4 I4	LS4 I5	LS4 I6	LS4 I7	LS4 I8	LS4 I9
WS	16	13	10.99	10.84	10.94	10.91	11.19	13.95	10.92	10.91	10.80
WR	2	2	3.18	3.11	3.10	3.09	3.11	3.01	3.11	3.10	3.16
AD	3	5	6.13	6.01	6.10	4.86	4.62	4.61	6.03	6.17	6.04
SS	4	4	1.95	1.92	1.92	6.16	5.84	5.83	1.94	1.88	1.82

Tabla 7. Valores geométricos optimizados.

Se puede observar en esta tabla como WR tiende a 3 en cualquier instancia, sin embargo, el resto de los parámetros varía en función de la instancia.

En la figura 18 se muestra el comportamiento de varios de los robots de la tabla 7 simulados en la herramienta previa.

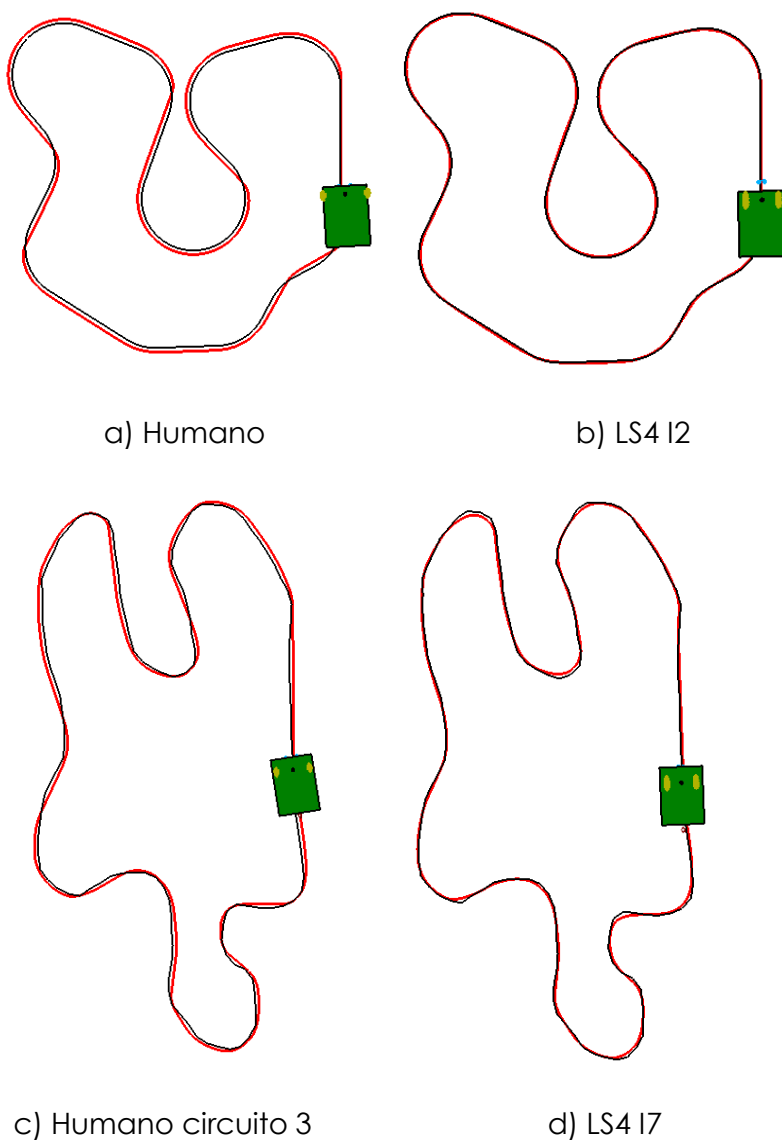


Figura 18. Simulación de varios robots de la tabla 7.

Conclusiones

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

Lo más 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 más problemas ha causado ha sido la decisión a 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 más óptimo puesto que la relación calidad tiempo es mucho más 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 aún más los resultados obtenidos.

Las posibles mejoras a realizar entran dentro de la mejora de este este 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. Abraham Duarte Muñoz, Juan José Pantrigo Fernández y Micael Gallego Carrillo (2007) *"Metaheurísticas"*
4. D. Henderson, S.H. Jacobson y W.A. Jonhson (2003) *"Handbook of metaheuristic"*
5. F. Glover y M. Laguna (1997) *"Tabu Seatch"*
6. P. Hansen y N. Mladenović (2003) *"Handbook of Metaheuristic"*
7. J. H. Holland (1975) *"Adaptation in Natural and Artificial Systems"*
8. F. Glover (1998) *"In Artificial Evolution, Lecture Notes in Computer Science"*
9. M. Dorigo (1992) *"Optimization, Learning and Natural Algorithms"*
10. C.G.E. Boender, A.H.G. Rinnooy, L. Stougie y G.T. Timmer (1982) *"Mathematical Programming"*

Java

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

Metodología ágil

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