UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Análisis de algoritmos heurísticos y su aplicación en un sistema robótico para Pick \& Place

TRABAJO FIN DE GRADO

Septiembre - 2021

AUTOR: Adrián Guijarro Terol

DIRECTOR/ES: Miguel Onofre Martínez Rach y Carlos Pérez Vidal

Contents

List of Figures

List of Tables

Resumen

El problema de *pick & place*, es un tipo específico de problemas de decisión que consiste en que un robot coloque un numero de piezas dado en su lugar correspondiente, diferenciando tipos de piezas. Este conjunto mencionado contiene problemas de búsqueda y de optimización. El principal objetivo de este trabajo es buscar una solución eficiente, es decir, que requiera pocos recursos computacionales, y además que sea lo más cercana posible a la solución óptima en términos de una función de coste. Esta solución se buscará utilizando diferentes algoritmos y problemas de optimización combinatorios, lo más cercana posible a la solución óptima de un sistema robótico.

1 Introducción, motivación y objetivos

En este capítulo se presenta una introducción del proyecto, la motivación que ha llevado a su planteamiento y también los objetivos que se plantearon en un primer momento.

1.1 Introducción

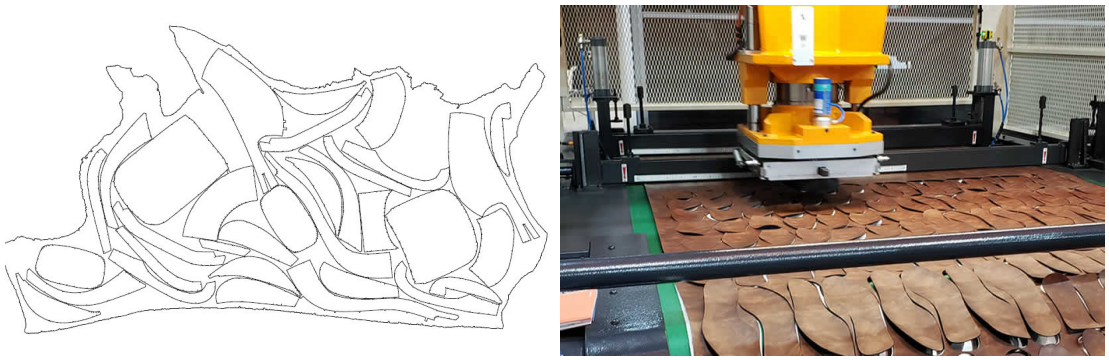
La mayoría de procesos industriales están, o son susceptibles de ser automatizados. Mediante esta automatización se mejoran los rendimientos en cuanto a la economía y costes. El robot utilizado en la aplicación de este proyecto es un robot industrial colaborativo. La robótica colaborativa es aquella que es capaz de trabajar en un entorno interaccionando con los operarios, facilitando el trabajo de estos, si necesidad de seguridad. Los robots colaborativos suelen ser ligeros, sencillos de manejar y fácilmente programables.[1]

Por otra parte los robots colaborativos, son fácilmente adaptables a distintos ámbitos dentro de una industria, tanto con requerimientos mínimos de seguridad, como en los costes al invertir en este tipo de robótica.[1]

En la tarea que se realiza en este proyecto, de *pick & place*, con el fin de mejorar los procesos de automatización se desarrollan diferentes algoritmos. Podemos realizar procedimientos deterministas para su resolución o procedimientos de optimización más complejos, por ejemplo, algoritmos genéticos.

Es buenos resultados teniendo en cuenta los requisitos de tiempo que necesitan los procesos a medida que se aumenta el número de variables del problema si utilizamos los métodos de búsqueda exhaustiva. Cuando esto ocurre, los métodos de búsqueda exhaustiva pueden ser sustituidos por métodos de optimización o métodos heurísticos que estos no siempre nos proporcionan la solución óptima, pero si una suficientemente cercana a la óptima.

Para llegar al proceso de *pick & place* antes se plasman en una lámina de cuero grande los distintos patrones de forma del las piezas de cuero del calzado, para posteriormente cortar estas piezas. En la siguiente imagen se muestran distintos patrones y como se recortan de la lámina grande de cuero.



(a) Patrones de las piezas (b) Recorte de las piezas

Figure 1: Preparación de las piezas

Una vez los patrones hayan sido cortados, se colocarán sobre distintas planchas donde más adelante pasarán a la zona donde esté el brazo robótico que hará el proceso de *pick & place*.

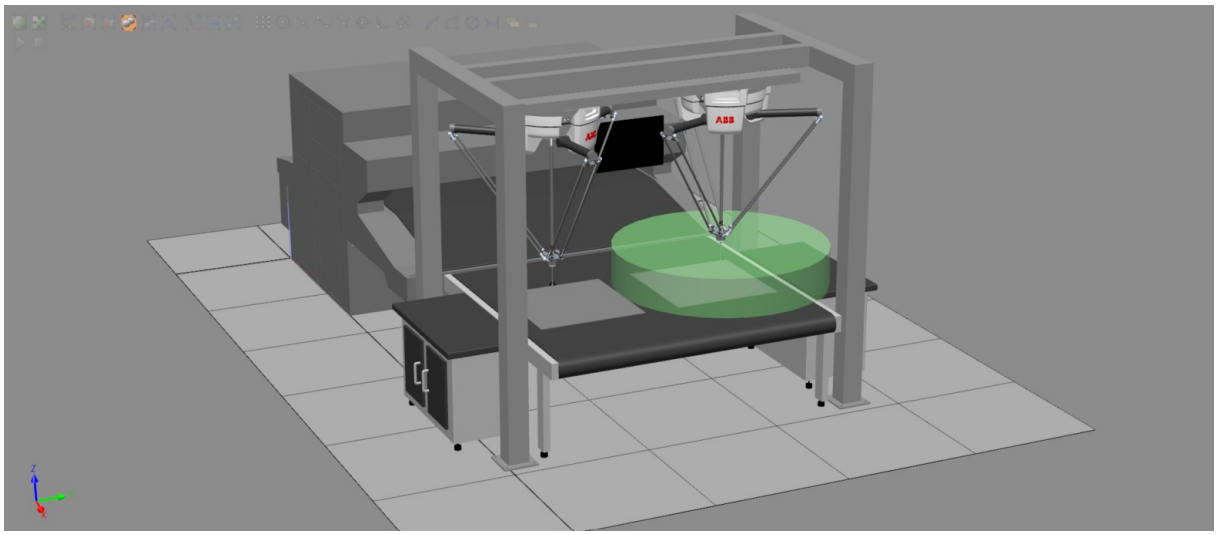


Figure 2: Estación brazo robótico

Finalmente, comienza el proceso de *pick & place*, que para un número de piezas (N) que están sobre la plancha de dimensiones de 1x1 metros consiste en tener que recoger esas N piezas y dejarlas en sus respectivos destinos, diferenciados por el tipo de cada pieza(por el patrón que han sido cortadas). Se han propuesto dos opciones a la hora de aplicar los algoritmos: La primera opción es que con antelación se conozcan las posiciones de las piezas en las planchas para poder calcular la secuencia de selección, si los tiempos computacionales fuesen excesivos, esta opción limita la adaptabilidad y carga del trabajo. Por otro lado, la segunda opción es calcular en tiempo real la secuencia de recogida de las piezas por lo que habría que tener en cuenta el tiempo que hay de preparación de las planchas.

1.2 Motivación

El proyecto está motivado por intentar mejorar el tiempo de recogida de las secuencias de *pick & place* de una instalación actual de corte de piezas de cuero para la industria del calzado, en concreto para la etapa de *pick & place*. La solución actual plantea la selección de las piezas en un orden determinado y fijo para todas las planchas.

A nivel personal la motivación del proyecto es aprender y ganar experiencia en el mundo de la programación y algoritmia, perfeccionar los conocimientos adquiridos durante el Grado. También fue otra motivación el poder iniciarme en el mundo de la robótica, que a conviene empezar a conocer ya que en el futuro será muy utilizada en cualquier industria.

1.3 Objetivos

Una vez introducido el proyecto, es importante fijar su objetivo. Desde el primer momento, el objetivo fue aprender sobre un tema que en el Grado no se profundiza tanto.

Por otro lado, el objetivo principal del proyecto es plantear una solución que mejore los tiempos de las secuencias de *pick & place* actuales utilizando un brazo robot y con tiempos computacionales adecuados. Otro objetivo es plantear una solución para resolver el mismo problema pero utilizando dos brazos robots, con las mismas consideraciones que con un solo brazo. Implementar los algoritmos de optimización y heurísticos en *python* y comparar y analizar los resultados. Utilizar un simulador para implementar los algoritmos y ver su viabilidad en un sistema real y finalmente plantear las conclusiones del proyecto.

1.4 Estructura del TFG

En el presente Trabajo de Fin de Grado está distribuido en seis capítulos más la bibliografía, se dará una breve explicación de cada uno a continuación:

Capitulo 1: Introducción, motivación y objetivos. Se expone una introducción al proyecto, la motivación para realizarlo y los objetivos que se persiguen.

Capítulo 2: Introducción a los algoritmos heurísticos. En este capítulo se explicarán los métodos y algoritmos utilizados de una forma general.

Capítulo 3: Solución problema un brazo. En este apartado se profundizará más en la aplicación de los métodos y algoritmos utilizados.

Capítulo 4: Solución problema dos brazos. Se exponen las principales diferencias que encontramos con la solución de 1 brazo.

Capítulo 5: Simulación. En este capítulo se explica la robótica y programación utilizada a la hora de simular los problemas abarcados en el proyecto.

Capítulo 6: Conclusión. Se exponen las conclusiones del proyecto, comparándolas con los objetivos marcados.

Bibliografía. Se indican las referencias utilizadas en este proyecto.

2 Introducción a los algoritmos utilizados

En este capítulo, se van a explicar de forma general los métodos de búsqueda exhaustiva y los algoritmos que se utilizan en el proyecto.

En el proyecto se utilizarán dos tipos de algoritmos: algoritmos deterministas y algoritmos heurísticos.

Los algoritmos deterministas son aquellos que conociendo las entradas a los mismos, siempre nos darán el mismo resultado, por muchas veces que los ejecutes.

Los algoritmos heurísticos son usados para resolver problemas que pueden modelarse como de optimización de funciones o de búsqueda que son intratables en tiempos razonables con otro tipo de algoritmos o métodos.

2.1 Búsqueda de fuerza bruta

Los algoritmos de fuerza bruta son capaces de encontrar la solución a cualquier problema por complicado que sea. Tienen un fundamento muy simple, comprobar todas las combinaciones posibles, recorrer todos los caminos, hasta encontrar la mejor solución. No importa que inicia por caminos malos o muy malos, al llegar al final, iniciará otro camino, hasta finalizar todos los posibles y quedarse con la mejor.

2.2 Algoritmo Raster Order

Los algoritmos de *raster order* son algoritmos deterministas. Realizan un barrido de forma progresiva de un área. Aunque es mas rápido, en un sentido más general, se puede comparar a como se desplaza nuestra mirada cuando se leen las líneas de texto. Por lo tanto, solo nos genera una secuencia.

Este algoritmo será el utilizado como referencia a la hora de compararlo con los algoritmos heurísticos que se van a utilizar.

2.3 Algoritmo Voraz

Como se puede ver en la referencia [2]. Los algoritmos voraces, como en el *raster order*, son algoritmos deterministas. Actúan de la siguiente manera: en cada iteración el algoritmo evalúa cada candidato a formar parte de la solución según su aporte a la función objetivo, de tal forma que si el elemento es seleccionado pasa a formar parte de la solución.

Estos algoritmos son bastante veloces en cuanto a su ejecución y con un diseño relativamente sencillo. Sin embargo pueden tener poca precisión y quedar lejos de la solución óptima.

Este tipo de algoritmo se utiliza por ejemplo en el problema de la mochila, que es un problema de optimización combinatoria de formulación sencilla, aunque tiene una resolución compleja. Este problema se basa en que tenemos que llenar una mochila con distintos objetos sin exceder su capacidad.

2.4 Algoritmo ILS

Los algoritmos ILS son algoritmos heurísticos. La idea principal del *Iterated Local Search* (ILS) es la de construir un camino aleatorio en S, el espacio de óptimos locales denotado por la salida de un algoritmo de búsqueda local. Se necesitaran cuatro procedimientos básicos para generar un algoritmo ILS: un procedimiento que genere una solución inicial,, un procedimiento de búsqueda local, un esquema de cómo perturbar una solución, y un *AccentanceCriterion*, que decide un criterio a partir de qué solución se continúa la búsqueda.

La eficacia de S depende de la definición de los cuatro procedimientos del ILS: la efectividad del algoritmo de búsqueda local es de gran importancia, ya que influye mucho en la calidad de la solución final del ILS. Las perturbaciones deben permitir al ILS escapar eficazmente de los óptimos locales, incluye el tipo de camino que vamos a utilizar para controlar la diversificación de la búsqueda. Como se ve en la referencia [3]

El algoritmo tiene la siguiente estructura:

Algorithm 1 Estructura Iterated Local Search

s0 = GenerarSolucionInicial
S = LocalSearch(s0)
repetir
S_0 = Perturbation(S)
$S' = LocalSearch(S_0)$
S = AcceptanceCriterion(S,S')
hasta que cumpla el criterio de terminación

2.5 Algoritmo Genético

Como se puede ver en la referencia [4]. Los algoritmos genéticos, como el ILS, también es un algoritmo heurístico. El algoritmo genético es una técnica de búsqueda utilizada para encontrar soluciones aproximadas a problemas de optimización combinatoria.

Los algoritmos genéticos son, más apropiadamente, una técnica de optimización basada en la evolución. El algoritmo incluye la idea de supervivencia del más apto. La idea es encontrar primero las soluciones y luego combinar la solución o soluciones mas aptas para crear una nueva generación de soluciones que debería ser mejor que la anterior. También se incluye un elemento de mutación aleatoria. El proceso del algoritmo genético consiste en lo siguiente:

Codificación: Se encuentra una solución adecuada para la solución del problema.

Evaluación: A continuación se selecciona la población inicial, normalmente de forma aleatoria. A continuación se calcula la aptitud de cada individuo y se seleccionan los mejores.

Cruce: La aptitud se utiliza la posibilidad de cruce del individuo. El cruce consiste en combinar dos individuos para crear nuevos individuos, si es posible.

Mutación: A continuación se produce la mutación. Se eligen al azar algunos individuos para ser mutados y luego se eligen al azar un punto de mutación.

Decodificación: Una vez hecho los pasos anteriores, se ha formado una nueva generación y el proceso se repite hasta que se alcanza algún criterio de parada. En este punto se decodifica el individuo que más se acerca al óptimo y se completa el proceso.

Este algoritmo se utiliza por ejemplos en el problema del viajero, un problema clásico de optimización combinatoria. El problema consiste en encontrar el recorrido más corto posible a través de un conjunto de n-vértices de manera que cada vértice sea visitado exactamente una vez.

2.6 Algoritmo DTSA

El algoritmo Tree-Seed (TSA) es un algoritmo de búsqueda iterativa inspirado en la naturaleza y basado en la población. El TSA se propone para resolver problemas de optimización contínua inspirándose en las relaciones entre arboles y sus semillas. En este proyecto se utiliza un TSA rediseñado utilizando operadores de trnasformación de intercambio, desplazamiento y simetría para resolver los problemas de optimización codificados por permutación y se denomina DTSA. Este algoritmo, para investigar su rendimiento, como en el genético también se utiliza en el problema del viajero. Este es el tercer algoritmo heurístico utilizado en este proyecto.[5]

3 Solución problema un brazo

En este capítulo, se estudiará el problema para el caso de un brazo. En primer lugar, se analizan los métodos de fuerza bruta, un algoritmo de *raster order* y otro voraz. Seguidamente, se tratará el problema mediante algoritmos combinatorios: algoritmo genético, ILS (*Iterative Local Search*) y DTSA (*Discrete Tree Seed Algorithm*).

En el problema con un brazo se buscará una solución óptima o lo más cercana posible a esta, donde la solución óptima será la que recorra una menor distancia, o lo que es lo mismo, tarde un menor tiempo en recoger todas las piezas. El número de posibles permutaciones, $p = [p(1), \cdots, p(N)]^T$ del vector $[1, 2, \cdots, N]^T$, es $N!$, donde N es el numero de piezas, por lo tanto $N!$ es el número de posibles caminos que tenemos para recoger las piezas. La distancia que recorremos en cada camino viene dada por la siguiente expresión:

$$d = \sum_{k=1}^{N} \left\| \vec{r}_{p(k)} - \vec{d}_{p(k-1)} \right\|_2 + \left\| \vec{r}_{p(k)} - \vec{d}_{p(k)} \right\|_2 \tag{3.1}$$

donde, $\vec{r}_i$ y $\vec{d}_i$ denotan la posición de origen y destino de la pieza i, respectivamente, además por simplicidad en la notación, la posición inicial del brazo se denota como $p(0)$, aunque no es un elemento de la permutación p.

3.1 Algoritmo 1: Fuerza bruta

En este algoritmo, primero se calculan todas las combinaciones posibles con el numero de piezas dado, es decir, $N!$ permutaciones. A continuación, se calcula para cada permutación $p = [p(1), \cdots, p(N)]^T$ la distancia total que recorre el brazo hasta dejar todas las piezas en su destino. Esto se realiza mediante la expresión 2.1

Algorithm 2 Fuerza bruta

pini = punto inicial del brazo
mejorcoste = distancia de la primera permutacion
mejorpermut = la primera permutacion

```
1: for (j = 0 in range((N!))) do
2:     distancia = distTotal(pini,j)        ▷ Calculo distancia de la permutacion j
3:     if distancia < mejorcoste then
4:         mejorcoste = distancia
5:         mejorpermut = permutacion j
```

El número de operaciones que el algoritmo realiza, el cual denotamos como $C(N)$, crece de manera exponencial, más específicamente, es $\Omega(\sqrt{N}e^{N\log(N)})$, donde la notación $\Omega(f(N))$ es equivalente a:

$$\lim_{N\to\infty} \frac{C(N)}{f(N)} > 0 \tag{3.2}$$

esto se obtiene por medio de la utilización de la aproximación de Stirling, i.e, $N! \approx \sqrt{2\pi N}\left(\frac{N}{e}\right)^N$.

3.2 Algoritmo 2: Raster order

En el *raster order*, no se calculan todas las combinaciones posibles, sino que se halla una sola combinación. En el orden que se van encontrando las piezas según sus posiciones, de izquierda a derecha y de arriba a abajo, por lo que el coste computacional disminuye en una gran cantidad. Para esta única combinación se calculará la distancia que recorre para tener una referencia con que comparar más adelante los algoritmos de optimización.

Algorithm 3 Raster order

perm = lista de las permutaciones posibles
pini = punto inicial del brazo
piezasOrdenadas = inicializo un vector vacío

```
for (i = 0 in range(len(N))) do
    Añado las piezas con su tiempo al vector piezas Ordenadas
    piezasOrdenadas.append(i, tiempoTodest(pini, piezas[i,1:3]))
Ordeno las piezas
piezasOrdenadas.sort(key = lambda x:x[1])
```

El coste de este algoritmo es cuadrático, es decir su complejidad es es $O(N^2)$, porque el algoritmo se basa en dos bucles *for* anidados para calcular el valor de la permutación.

3.3 Algoritmo 3: Voraz

En este algoritmo voraz, como en el anterior, solamente genero una combinación, el brazo recoge las piezas que más cerca estén de su destino. Se utilizará para tener otra referencia en el caso de un brazo para poder comparar mas adelante el funcionamiento de los algoritmos de optimización utilizados. Se generan listas de piezas de cada tipo y se calcula a que distancia están de su destino, las mas pequeñas se almacenan en otra lista, llamada candidatos, y de los mejores candidatos se calcula al distancia y la permutación correspondiente.

Algorithm 4 Voraz

```
pini = punto inicial del brazo
piezasA,piezasB,piezasC = inicializo un vector vacío
candidatos = vector vacio
pactual = inicializo al punto inicial que empieza el brazo
mejorcombinacion = vector vacio
pesototal = 0
menorpeso = valor alto para poder comparar con el primer peso calculado

for (i = 0 in range(len(piezas))) do
    tpieza = tipoPiezas(i)
    if (tpieza == A) then
        piezasA.append(i,distTodest(pieza))
    if (tpieza == B) then
        piezasB.append(i,distTodest(pieza))
    if (tpieza == C) then
        piezasC.append(i,distTodest(pieza))

Ordeno las piezas de menor peso a mayor
piezasA.sort(key = lambda x:x[1])
piezasB.sort(key = lambda x:x[1])
piezasC.sort(key = lambda x:x[1])

Añado el primero de cada tipo a candidatos
candidatos.append(piezasA[0][0])
candidatos.append(piezasB[0][0])
candidatos.append(piezasC[0][0])

while (len(candidatos)! = 0) do
    for (i = 0 in range(len(candidatos))) do
        p = peso(candidatos[i],pactual)
        if (p < menorpeso) then
            menorpeso = p
            mejorcandidato = candidatos[i]
pesototal += menorpeso
pactual = Actualizacion del punto actual
mejorcombinacion.append(mejorcandidato)
candidatos.remove(mejorcandidato)
nuevocandidato(mejorcandidato)
menorpeso = actualizacion a un valor muy alto
```

En este algoritmo, con respecto a los otros dos hay una función nueva, nuevocandidato(candidato), esta función añade un candidato nuevo, del mismo tipo de pieza que se ha eliminado anteriormente, siempre y cuando queden de ese tipo de piezas.

El coste de este algoritmo es cuadrático, al igual que en el *raster order*, por lo que su complejidad es O(N^2), porque el algoritmo se basa en dos bucles *for* anidados para calcular el valor de la permutación.

3.4 Algoritmo 4: Iterated Local Search (ILS)

Como se muestra en la referencia [1]. La idea principal del ILS es la de buscar mínimos locales de un algoritmo de búsqueda, mediante una selección aleatoria. Este algoritmo se basa en cuatro pasos principales: Generar una solución inicial, ya sea una permutación aleatoria o un resultado del *raster order* o el Voraz, una función de búsqueda local, que será la encargada de encontrar mínimos locales el valor de la distancia, una función *perturbation*, encargada de perturbar una solución y un *acceptance criterion*, que decide si la búsqueda del mínimo local continúa.

El algoritmo es el siguiente:

Algorithm 5 Iterated Local Search

```
1: s0 = SolucionInicial
2: s1 = LocalSearch(s0)
3: b = false
4: while (not b) do
5:     sprima = Perturbation(s1[1])
6:     s1prima = LocalSearch(sprima)
7:     b = AcceptanceCriterion(s1,s1prima)
8:     if (b == True) then
9:         s1 = s1prima
```

3.4.1 Local Search

Esta función, recibe como parámetro de entrada una permutación, que sea solución, se generan vecinos de esta permutación, los vecinos son todas las combinaciones posibles entre dos posiciones, este proceso se repite hasta que se encuentra un valor menor al mínimo local anterior encontrado. El numero de vecinos que se obtiene es:

$$N_v = \frac{N^2}{2} - \frac{N}{2} \tag{3.3}$$

donde N_v determina el número de vecinos.

3.4.2 Perturbation

Recibe como parámetro de entrada una solución, y, aleatoriamente se eligen entre 3 y 6 índices de la permutación, estos índices se barajan y se cambian en la permutación, para generar otra.

3.4.3 Acceptance Criterion

Esta función recibe como parámetros de entrada las dos soluciones obtenidas por el *local search*, si el valor de la segunda entrada es menor que el de la segunda, devuelve *True* y el bucle acaba.

3.5 Algoritmo 5: Genético

En primer lugar, en el algoritmo genético se define una población con un número inicial par, denotado por P, de individuos, calculamos el peso de cada individuo y me quedo con los $P/2$ individuos con el peso más bajo. En segundo lugar, se generan $P/2$ individuos para generar de nuevo la población inicial, a diferencia de otros algoritmos genéticos, en este no se cruzan los individuos, ya que por la definición del individuo no se pueden cruzar, por lo que se realiza una mutación. Esta mutación se realiza escogiendo dos índices de la permutación o individuo de forma aleatoria y se intercambian esas posiciones. Mediante un número de generaciones se comparan los mejores individuos y me quedo con el mejor.

Algorithm 6 Genético

```
M = 50                                                  ▷ Población inicial
permutacion = [ ]
poblacion = [ ]
for (j = 0 in range(len(M))) do
    b = True
    while (b) do
        permutacion = permutaciones(len(piezas))        ▷ Genera una permutación aleatoria
        b = comprobacionpermuts(permutacion, poblacion)
        if (b == False) then
            poblacion.append(permutacion)
(mp,mi,s) = seleccion(población)                        ▷ Selección de los mejores
generaciones = 100
newpob = [ ]                                            ▷ Vector para guardar la nueva población
for (n in range(generaciones)) do
    newpob = mutacion(s,M/2)
```

3.5.1 Permutaciones

Esta función genera las permutaciones de forma aleatoria, recibiendo como parámetro la cantidad de piezas que tenemos.

3.5.2 Comprobación Permuts

Recibe como parámetros una premoción y la lista de permutaciones creadas, devuelve *True* si la permutación esta en al lista y *False* si no esta contenida en la lista. Se utiliza para que en la lista de permutaciones no se repita ninguna anterior.

3.5.3 Mutación

Esta función utiliza otra llamada *swap pos*, la cual realiza un cambio entre dos posiciones aleatorias de una permutación. Mutación tiene como parámetros de entrada la selección de los mejores individuos y el numero de seleccionados, y devuelve la nueva población.

3.6 Algoritmo 6: Discrete Tree Seed Algorithm (DTSA)

Como se muestra en la referencia [2]. En este algoritmo, en la fase de inicialización, los arboles se crean de forma aleatoria, se generarán semillas mediante operadores de transformación, que se describirán en el siguiente apartado. Una vez tenga el mejor árbol, se le realizará el algoritmo 2-opt, se generarán todos los vecinos posibles con cambios de dos índices, el numero será el mismo que obtiene en el ILS, con la expresión 3.

3.6.1 Operadores de transformación

Se realizarán tres ordenadores de transformación: *Swap, Shift y Symmetri*. Swap: Dos posiciones aleatorias entre 0 y N-1, donde N es el numero de piezas. Estas dos posiciones se intercambian para crear una nueva semilla Shift: Dos posiciones aleatorias adyacentes elegidos de forma aleatoria entre 0 y N-1, donde N es el numero de piezas. Estas dos posiciones se desplazan hacia la izquierda. Symmetri: Se determinan dos bloqueas aleatorios de dos posiciones cada una. Cada bloque de posiciones se invierte, y después de intercambian los dos bloques.

Algorithm 7 DTSA

```
Determinar el numero de evaluaciones maxEvaluaciones
Numero de piezas D
Determinar el numero de arboles iniciales N
Determino el parámetro de tendencia de búsqueda ST
Determino los arboles iniciales
Calculo el peso de cada uno de los arboles
Los ordeno y escojo el mejor de los arboles iniciales
Inicializo un contador
numero de semillas ns = 6
Inicializo un vector de semillas
while contador < maxEvaluaciones do
    for (i = 0 in range(len(arbolesOrdenados)) do
        repetirBucle = True                    ▷ Variable que detiene el siguiente while
        while (repetirBucle) do
            numArbol = arbol aleatorio entre 0 y len(arbolesOrdenados)-1
            if numArbol! = i then
                arbolAleatorio = arbolesOrdenados[numArbol]
                repetirBucle = False
        Determino un numero aleatorio (rand) entre 0 y 1 para comparar con ST
        if (rand < ST) then
            Creo una semilla con el operador swap usando el mejor arbol
            Creo una semilla con el operador shift usando el mejor arbol
            Creo una semilla con el operador symmetri usando el mejor arbol
            Creo una semilla con el operador swap usando arbolAleatorio
            Creo una semilla con el operador shift usando arbolAleatorio
            Creo una semilla con el operador symmetri usando arbolAleatorio
        if (rand > ST) then
            Creo una semilla con el operador swap usando arbolesOrdenados[i]
            Creo una semilla con el operador shift usando arbolesOrdenados[i]
            Creo una semilla con el operador symmetri usando arbolesOrdenados[i]
            Creo una semilla con el operador swap usando arbolAleatorio
            Creo una semilla con el operador shift usando arbolAleatorio
            Creo una semilla con el operador symmetri usando arbolAleatorio
        contador += ns
        Ordeno las semillas
        Me quedo con la mejor semilla
        if (mejorSemilla < mejorArbol) then
            Sustituyo el mejor arbol por la mejor semilla
Aplico el algoritmo 2-opt a la mejor semilla
```

3.7 Resultados

En esta sección se comentarán los resultados obtenidos en los algoritmos desarrollados y explicados en las secciones anteriores. Primero se presentaran los tiempos de recogida de piezas para distintos números de piezas, además, también se tendrá en cuenta el coste computacional de cada algoritmo.

3.7.1 Comparación de los algoritmos

El desarrollo de los algoritmos se realizará con una intervalo de entre 5 y 50 piezas para poder comprobar que sean viables.

El primer algoritmo descrito es el ILS, como se puede ver en la tabla el tiempo aumenta con el numero de piezas, teniendo como referencia el algoritmo de *raster order*. En el caso de las 50 piezas, mejora el tiempo respecto el *raster order* un 8.90%. También puede verse que la mayor mejora se produce en las 15 piezas, con un 14.30%.

Table 1: Algoritmo ILS (s)

Nº piezas	T.ordenada	T.mejor	T.peor	Mejor-Ordenada	Mejor-Peor	%(mejor-ordenada)
5	6.955	6.186	8.775	-0.769	-2.589	11.06%
6	9.026	7.989	10.947	-1.037	-2.958	11.49%
7	10.297	9.438	12.434	-0.859	-2.996	8.34%
8	11.958	10.631	14.21	-1.327	-.379	11.10%
9	13.07	11.683	15.439	-1.387	-3.756	10.61%
10	14.137	12.323	16.217	-1.814	-3.894	12.83%
11	16.192	14.307	18.716	-1.885	-4.409	11.64%
12	18.858	16.327	21.319	-2.531	-4.992	13.42%
15	24.088	20.643	26.835	-3.445	-6.192	14,30%
20	31.521	27.677	34.791	-3.844	-7.114	12.20%
25	41.285	35.906	44.930	-5.379	-9.024	13.03%
30	51.628	45.11	55.426	-6.518	-10.316	12.62%
35	61.734	54.776	65.9	-6.958	-11.124	11.27%
40	70.447	63.699	76.367	-6.748	-12.668	9.58%
45	73.381	66.247	81.565	-7.134	-15.318	9.72%
50	79.937	72.82	89.46	-7.117	-16.64	8.90%

Donde T.ordenda es el tiempo que el *raster order* tarda en colocar las piezas, T.mejor y T.peor son el mejor y peor tiempo del algoritmo, Mejor-Ordenada es la diferencia de tiempo entre la permutación ordenada y el mejor tiempo del algoritmos, Mejor-Peor es la diferencia de tiempo entre estos dos tiempos y % (mejor-ordenada) el porcentaje de mejora respecto la permutación ordenada.

En la siguiente tabla se muestra los resultados del algoritmo genético, con una cantidad de 50 generaciones. Se puede ver como el tiempo aumenta conforme aumenta el numero de piezas, como en el ILS. En el caso de las 50 piezas, el algoritmo genético mejora el

tiempo un 7.01% respecto al *raster order*, un 1.89% menos que en el ILS. También vemos que la mayor mejora es en el caso de 15 piezas con un 14.21%, 0.09% menos que en el ILS.

Table 2: Algoritmo Genético (s)

Nº piezas	T.ordenada	T.mejor	T.peor	Mejor-Ordenada	Mejor-Peor	%(mejor-ordenada)
5	6.955	6.057	8.876	-0.898	-2.819	12.91%
6	9.026	7.926	10.947	-1.1	-3.021	12.19%
7	10.297	9.325	12.549	-0.972	-3.224	9.44%
8	11.958	10.632	14.21	-1.326	-3.578	11.09%
9	13.07	11.683	15.265	-1.387	-3.582	10.61%
10	14.137	12.149	16.217	-1.988	-4.068	14.06%
11	16.192	13.987	18.476	-2.205	-4.489	13.62%
12	18.858	16.433	21.319	-2.425	-4.886	12.86%
15	24,088	20.664	26.679	-3.424	-6.015	14.21%
20	31.521	27.779	34.519	-3.742	-6.74	11.87%
25	41.285	36.048	44.824	-5.237	-8.776	12.68%
30	51.628	45.49	55.045	-6.138	-9.555	11.89%
35	61.734	55.277	65.48	-6.457	-10.203	10.46%
40	70.447	64.388	75.715	-6.059	-11.327	8.60%
45	73.381	67.555	80.933	-5.826	-13.378	7.94%
50	79.937	74.33	88.145	-5.607	-13.815	7.01%

Por último, el algoritmo DTSA, para hallar los datos de la siguiente tabla se realizan 3600 evaluaciones en el algoritmo. Al igual que en los dos casos anteriores, el tiempo aumenta con el número de piezas. En este caso, para las 50 piezas, mejora un 8.82% respecto a la referencia del *raster order*, mientras que la mayor mejora, nuevamente en las 15 piezas, con una mejora de 14.16%.

Table 3: Algoritmo DTSA (s)

Nº piezas	T.ordenada	T.mejor	Mejor-Ordenada	%(mejor-ordenada)
5	6.955	6.158	-0.797	11.46%
6	9.026	7.989	-1.037	11.49%
7	10.297	9.325	-0.972	9.44%
8	11.958	10.632	-1.362	11.09%
9	13.07	11.683	-1.387	10.61%
10	14.137	12.199	-1.938	13.71%
11	16.192	14.244	-1.948	12.03%
12	18.858	16.433	-2.425	12.86%
15	24.088	20.678	-3.41	14.16%
20	31.521	27.734	-3.787	12.01%
25	41.285	35.921	-5.364	12.99%
30	51.628	45.12	-6.508	12.61%
35	61.734	54.765	-6.969	11.29%
40	70.447	63.7	-6.747	9.58%
45	73.381	66.381	-7	9.54%
50	79.937	72.89	-7.047	8.82%

Comparando los tres algoritmos, nos dan resultados muy similares, aunque, tienen pequeñas diferencias, en el algoritmo ILS, para el caso de 50 piezas, tiene un tiempo un poco menor a los otros dos, como se muestra en la siguiente figura.

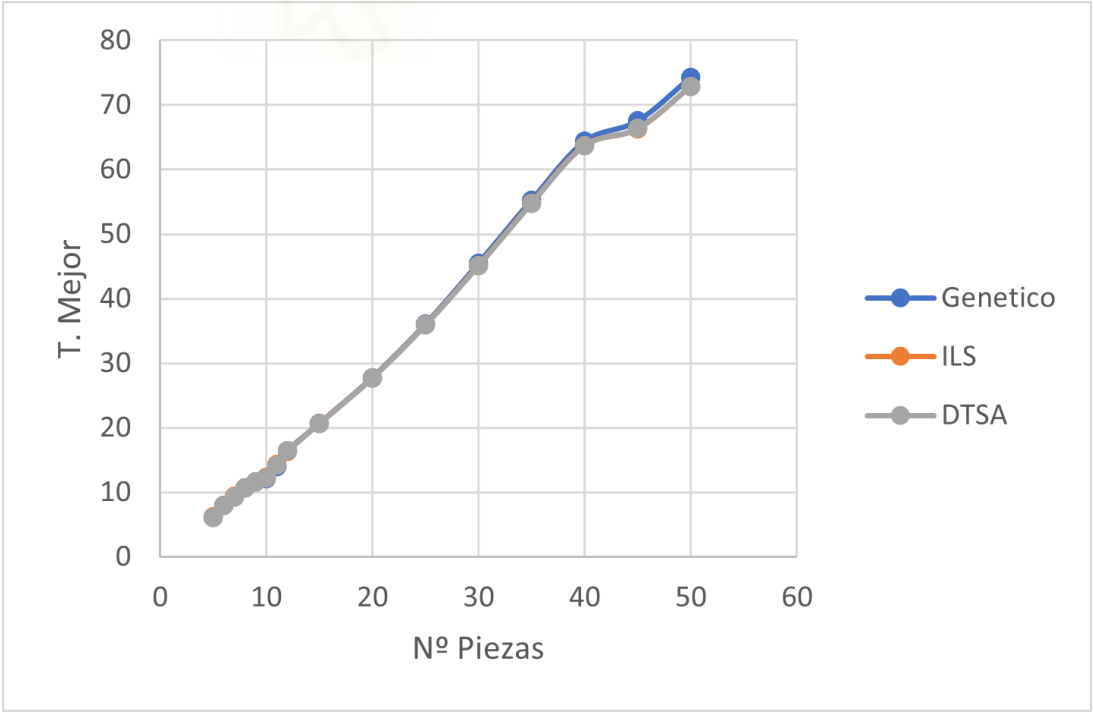


Figure 3: Comparación algoritmos

3.7.2 Coste computacional

El coste computacional también es algo importante a tener en cuenta, ya que un tiempo muy bajo de la secuencia se puede ver afectado por un tiempo muy alto computacional. En la siguiente tabla se muestra el coste computacional de los tres algoritmos, como se puede observar, esta característica es más importante en un numero de piezas elevado.

Table 4: Coste computacional (s)

Nº piezas	Genético	ILS	DTSA
5	35.057	0.596	16.802
6	43.503	0.689	20.199
7	47.583	1.458	23.784
8	57.437	1.686	26.895
9	64.928	1.479	30.237
10	72.333	2.053	33.741
11	78.592	4.399	36.965
12	87.593	6.68	40.416
15	109.672	14.695	50.421
20	139.944	53.295	67.223
25	180.747	105.452	85.396
30	215.176	198.046	100.867
35	249.622	309.249	117.434
40	282.279	464.074	134.395
45	321.073	784.01	150.9
50	357.749	1177.374	167.291

Como se puede apreciar en la tabla anterior, el algoritmo DTSA es el que antes converge para los casos de un número de piezas elevado, aunque tiene un tiempo computacional bastante más alto al tiempo que se tardan en recoger las propias piezas. En la siguiente tabla se han reducido el número de evaluaciones del DTSA para que el tiempo computacional sea mas viable aunque se sacrifique un poco el tiempo en el que se recogen las piezas.

Table 5: Ajuste evaluaciones DTSA (s)

Nº evaluaciones	T. recogida	T.computacional
1000	73.318	97.459
500	73.685	74.973
400	73.711	71.359
300	73.869	67.122
200	74.325	62.310
100	76.605	57.895

4 Solución problema dos brazos

En este capítulo se trata el mismo problema que explicado en el problema 3, con la diferencia que ahora se asume que hay dos brazos robóticos para al colocación de las N piezas.

La principal diferencia entre este escenario y el mostrado en el capítulo anterior, es que en este caso hay que tener en cuenta la colisión de los brazos, es decir, hay que añadir una restricción al problema de optimización, y por lo tanto no todas las permutaciones pertenecerán al conjunto factible. En concreto, diremos que los dos brazos colisionan si existe un instante t tal que la distancia entre ambos brazos es menor que el diámetro de estos.

Otra diferencia es la expresión con la que se calcula la distancia recorrida por los brazos, siendo en este caso calculada por las siguientes expresiones recursivas:

$$\begin{cases} \begin{aligned} D_{k+1} = D_k &+ b(k)\left(\left\|\vec{a}_k^{(1)} - \vec{r}_{p(k)}\right\|_2 + \left\|\vec{r}_{p(k)} - \vec{d}_{p(k)}\right\|_2\right) \\ &+ (1-b(k))\left(\left\|\vec{a}_k^{(2)} - \vec{r}_{p(k)}\right\|_2 + \left\|\vec{r}_{p(k)} - \vec{d}_{p(k)}\right\|_2\right) \end{aligned} \\ \vec{a}_{k+1}^{(1)} = (1-b(k))\,\vec{a}_k^{(1)} + b(k)\vec{d}_{p(k)} \\ \vec{a}_{k+1}^{(2)} = b(k)\vec{a}_k^{(2)} + (1-b(k))\,\vec{d}_{p(k)} \end{cases} \tag{4.1}$$

donde $b = [b(1), \cdots, b(N)]^T$, es un vector que denota que brazo va a mover que pieza, en específico $b(k) = 0\,(1)$ significa que el brazo $1\,(2)$ va a colocar la pieza $p(k)$. $\vec{a}_k^{(1)}$ y $\vec{a}_k^{(2)}$ representan la posición del brazo uno y dos después de colocar la pieza k respectivamente.

Otra cosa a tener en cuenta con los dos brazos, es que el tiempo que tardan en recoger las piezas, no es el tiempo que tardan los dos brazos, sino el brazo que mas tiempo tarda en recoger sus piezas.

4.1 Algoritmo 1: Fuerza bruta

En este algoritmo se calculan todas las combinaciones posibles con el numero de piezas que tengo ($N!$), para cada una de estas combinaciones, se generan todas las combinaciones binarias como número de piezas tengo. Las combinaciones binarias indican que piezas coge cada brazo. Por ejemplo, para 3 las combinaciones binarias serían como en la siguiente imagen:

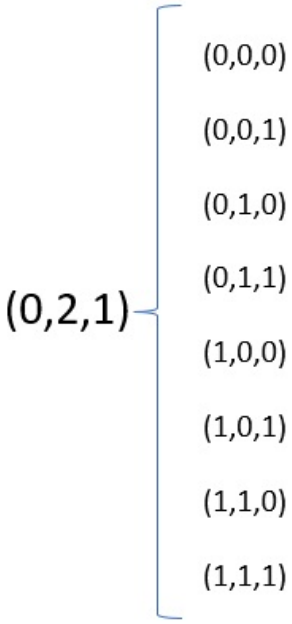


Figure 4: Combinaciones binarias para 3 piezas

Algorithm 8 Fuerza Bruta dos brazos

```
Inicializo un mejorCoste, mejorPermutacion y mejorCombinacion
for (i = 0 in range(len(perm))) do
    for (j = 0 in range(len(b))) do          ▷ b es el vector con combinación binaria
        Calculo la distancia de los dos brazos
        Calculo la secuencia que sigue el brazo 1 y brazo 2 (secuenciaTRayectoria)
        Calculo las trayectorias completas de los dos brazos (muestreoTramos)
        Calculo la diferencia entre todos los puntos de las dos trayectorias
        for (w = 0 in range(len(dif))) do
            Calculo de la colisión (colisionTrayectoria)
            if (colision == 0) then
                break
            if (colision == 1) then
                if (distancia2brazos < mejorCoste) then
                    Sustituimos los valores
```

El coste de este algoritmo es:

$$\Omega(2^N\sqrt{N}e^{Nlog(N)})$$

La diferencia con el de un brazo, es el factor 2^N por las combinaciones binarias de cada permutación.

4.1.1 secuenciaTrayectoria

Esta función recibe como parámetros de entrada una permutación, una combinación binaria y un indicador del brazo que queremos saber su trayectoria y nos devuelve un vector con las posiciones en las que el brazo ha parado a recoger o dejar las piezas, es decir, divide en tramos una trayectoria.

4.1.2 tiempoTramos

Esta función es la que calcula el tiempo que tarda cada tramo en realizarse, devuelve un vector de los tiempos de cada tramo y recibe como parámetro el vector que contiene la secuencia de puntos. En esta distancia se le añade una distancia virtual:

$$dist_V = v * tiempoBS \tag{4.2}$$

donde v determina la velocidad a la que se mueve el brazo y $tiempoBS$ es el tiempo el cual tarda en bajar y subir de nuevo para recoger la pieza.

4.1.3 muestreoTramos

Esta función realiza un muestreo de las coordenadas por los tramos que pasa cada brazo hallados en la función secuenciaTrayectoria. Este muestreo se realiza cada Δt a elegir. Se calcula mediante la siguiente expresión:

$$\begin{aligned} r = iTActu + ((tActu - tTramT + tTram[contT])/ \\ tTram[contT]) * (fTActu - iTActu) \end{aligned} \tag{4.3}$$

donde $iTActu$ determina el inicio del tramo en el que estoy, $tActu$ es el tiempo que tengo que ir actualizando conforme avanza el bucle, $tTramT$ es el tiempo acumulado de todos los tramos, $tTram$ es el tiempo que se tarda en recorrer el tramo actual en el que estamos, $contT$ es un contador para saber en que tramo estoy, y $fTActu$ es el final del tramo en el que estoy. (Explicación extra : si me encuentro en el final del tramo ((tActual-tiempoTramT+tiemTram[contT])/tiemTram[contT]) = 1 por lo que tendríamos r = finalTActu)

4.1.4 colisionTrayectoria

Esta función simplemente devuelve un 0 si el vector diferencia es menor que un valor escogido, es decir, si hay colisión o devuelve un 1 si es mayor.

4.2 Algoritmo 2: ILS

El algoritmo de optimización ILS para el caso de dos brazos, el mayor cambio que encontramos en este escenario con el de un brazo es que tengo que asegurarme de que no ocurre colisión entre las trayectorias, además de que se cumpla el AccentanceCriterion.

Otra cosa a tener en cuenta, es que la primera búsqueda local que se realiza, también tiene que cumplir el requisito de las colisiones, para asegurarse de que una va a cumplir y tener una solución válida. También impongo un numero máximo de iteraciones en la segunda búsqueda local.

4.3 Algoritmo 3: Genético

La principal diferencia del genético con dos brazos, es que los seleccionados tienen que cumplir que no exista colisión, si esta condición no estuviese, cabría la posibilidad de que todos los seleccionados fuesen mejores, ya que si no se tiene en cuenta la colisión, se tardará menos en recoger todas las piezas.

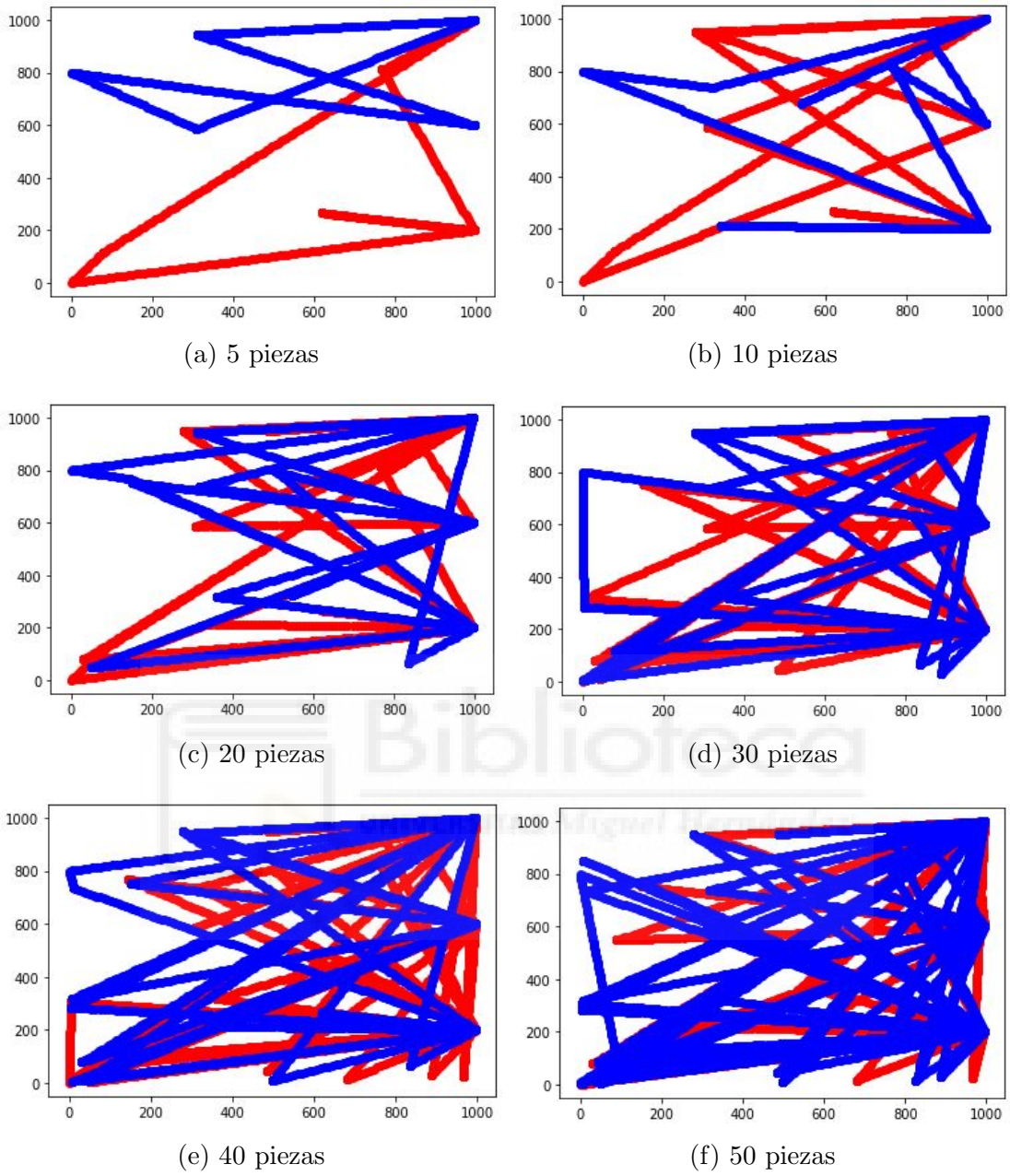
4.4 Algoritmo 4: DTSA

En cuando al DTSA, el cambio es, como en los dos anteriores, tener en cuenta en los individuos que no solo se generan por la distancia y la permutación, sino también por una combinación binaria, y por supuesto, que se cumpla la condición de no colisión.

4.5 Resultados

En este apartado, una vez analizados los resultados obtenidos con el caso de un brazo, vamos a comprarlos con los resultados que tenemos con dos brazos. Los tiempos obtenidos con los dos brazos son obviamente mas bajos, aunque no siempre puede ser así, ya que como se ha mencionado antes el funcionamiento de los dos brazos es más complejo que el de un brazo.

En este caso, no se ha utilizado ni el algoritmo *raster order* ni el voraz, ya que no se ha podido hallar una forma de recoger las piezas en la que no se genere una colisión, para entenderlo de una manera mas visual la siguiente figura muestra las trayectorias de los dos brazos una vez recogidas todas las piezas.



(a) 5 piezas (b) 10 piezas

(c) 20 piezas (d) 30 piezas

(e) 40 piezas (f) 50 piezas

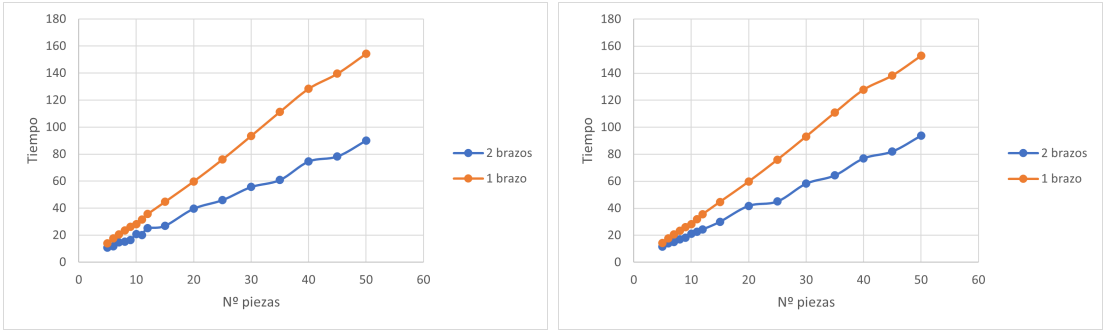
Figure 5: Trayectorias dos brazos

En la siguiente tabla se puede ver los tiempos de recogida de los dos brazos y el porcentaje de mejora con un brazo, para compararlo en las mismas condiciones, se ha añadido una penalización a los tiempos de un brazo, es decir, se ha tenido en cuenta el tiempo de subida y bajada del brazo. Se puede ver en la mayor parte de los casos, a medida que aumentamos las piezas, el tiempo también aumenta, hay casos en los que no ocurre lo mencionado por el factor aleatorio de los algoritmos. EL algoritmo que más porcentaje de mejora es el genético con un 41.64% para el caso de 50 piezas.

Table 6: Comparativa entre uno y dos brazos (s)

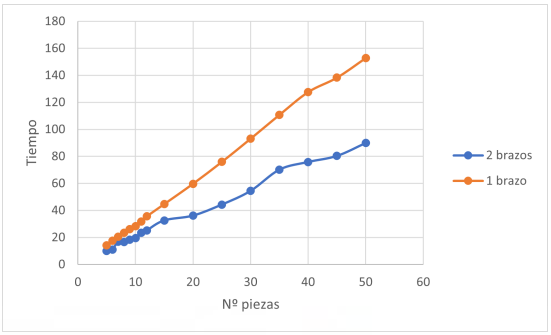
Nº piezas	Genetico	ILS	DTSA	% mejora Gen	% mejora ILS	% mejora ILS
5	-3.243	-2.709	-4.233	23.07%	19.10%	29.90%
6	-5.662	-3.634	-6.519	32.31%	20.66%	37.07%
7	-5.892	-5.597	-3.734	28.71%	27.12%	18.19%
8	-8.214	-6.438	-6.873	35.05%	27.48%	29.33%
9	-9.592	-7.973	-7.837	36.77%	30.57%	30.05%
10	-7.439	-7.247	-8.626	26.43%	25.59%	30.59%
11	-11.424	-9.321	-8.505	36.17%	29.21%	26.71%
12	-10.496	-11.22	-10.393	29.46%	31.58%	29.17%
15	-17.768	-14.723	-12.093	39.78%	32.98%	27.07%
20	-20.196	-17.929	-23.613	33.79%	30.04%	39.53%
25	-30.079	-30.915	-31.633	39.55%	40.73%	41.67%
30	-37.826	-34.882	-38.690	40.46%	37.46%	41.55%
35	-50.412	-46.406	-40.615	45.31%	41.89%	36.67%
40	-53.799	-50.714	-51.878	41.90%	39.71%	40.63%
45	-61.244	-56.114	-58.057	43.89%	40.59%	41.95%
50	-64.260	-58.939	-63.012	41.64%	38.57%	41.21%

En las siguientes figuras, también podemos observar la diferencia de los resultados entre un brazo y dos brazos, donde se puede ver que a medida que las piezas aumentan también crece la diferencia entre los tiempos de recogida.



(a) Genético

(b) ILS



(c) DTSA

Figure 6: Comparación 2 brazos con 1 brazo

Los dos brazos mejoran mucho en cuanto a un solo brazo, aunque no tanto como se podría esperar, ya que la complejidad de los dos brazos aumenta significativamente con la complejidad de un brazo.Además un segundo brazo en los casos que tenemos pocas piezas no vale la pena ya que no se consiguen mejoras significativas. Algo importante a tener en cuenta en estas comparaciones es que para que estén en igualdad de condiciones se le ha añadido la distancia virtual.

4.6 Coste computacional

En cuanto al coste computacional, podemos ver como en el caso de un brazo, el mejor es el DSTA, con un tiempo de 167.291 segundos con mucha diferencia respecto al ILS que tiene un tiempo de 1177.374 segundo. En el caso de los dos brazos, el mejor sigue siendo el DTSA con un tiempo de 204.842 segundos, mientras que, en este caso el peor es el genético con un tiempo de 1138.848 segundos. Esto se debe a que en un brazo, por la naturaleza del algoritmo no tiene un límite de iteraciones para finalizar, simplemente finaliza cuando se encuentra un mínimo, en los dos brazos, ya que la complejidad aumenta mucho, es necesario un límite de operaciones, ya que en ocasiones no se llega a encontrar un mínimo, como se ha mencionado antes.

Table 7: Coste computacional (s)

	Un brazo				Dos brazos		
Nº piezas	Genético	ILS	DTSA		Genético	ILS	DTSA
5	35.057	0.596	16.802		115.968	58.301	22.419
6	43.503	0.689	20.199		138.591	68.334	26.644
7	47.583	1.458	23.784		160.828	80.362	30.771
8	57.437	1.686	26.895		184.959	91.446	34.749
9	64.928	1.479	30.237		198.563	101.369	38.236
10	72.333	2.053	33.741		220.47	109.624	41.56
11	78.592	4.399	36.965		257.745	122.478	46.384
12	87.593	6.68	40.416		283.647	131.936	50.439
15	109.672	14.695	50.421		354.197	165.57	62.889
20	139.944	53.295	67.223		463.11	212.15	82.941
25	180.747	105.452	85.396		584.675	273.984	104.375
30	215.176	198.046	100.867		703.143	322.266	124.909
35	249.622	309.249	117.434		825.312	375.319	147.18
40	282.279	464.074	134.395		939.45	429.532	169.213
45	321.073	784.01	150.9		1037.959	475.604	186.395
50	357.749	1177.374	167.291		1138.848	529.609	204.842

5 Simulación

En esta sección se hablará sobre la robótica y programación utilizada a la hora de simular los problemas que se han abarcado en los capítulos anteriores.

5.1 Introducción a la robótica

Como vemos en la referencia [1]. En el sector tecnológico la robótica es una de las soluciones mas importantes a la hora de la fabricación de productos, tanto para que sea más versátil como más económica.

Estos robots se pueden diferenciar entre distintos tipos, los más conocidos son:

Robots industriales de manipulación: Estos tiene una base fija anclada a una plataforma de trabajo, son robots articulados diseñados para el desplazamiento de un útil de trabajo por la plataforma. En la siguiente figura se muestra un brazo robot industrial.

Figure 7: Brazo robot industrial

Robots de servicio: Son dispositivos, que sustituyen a las personas en los procesos cotidianos, por ejemplo las tareas de limpieza de una casa. En la siguiente figura se muestra un robot de limpieza del hogar.

Figure 8: Robot de Limpieza del hogar

Robots médicos: La robótica, cada vez más presente en el ámbito médico, facilitan las tareas del personal sanitario. Para diversas operaciones, por ejemplo, se emplean dispositivos láser de gran precisión. En la siguiente difura se muestra el robot quirúrgico *Da Vinci*

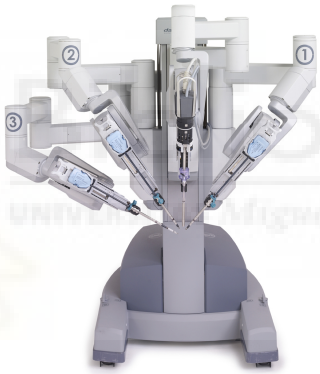


Figure 9: Robot quirúrgico

Nano robots: La idea de estos robots, es que sean capaces de buscar y destruir distintas enfermedades, como por ejemplo, tumores. En la siguiente figura se muestran los nano robots que se han conseguido diseñar hasta ahora.

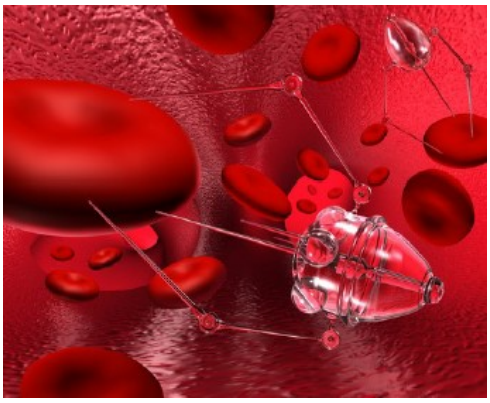


Figure 10: Nano robots

5.1.1 Robótica colaborativa

Tras una breve introducción a los tipos de robots, hay que enfocarse un poco más en el tipo de robótica que se ha utilizado en este proyecto, la robótica colaborativa. Esta robótica aparece con la necesidad de que los operarios puedan interaccionar con robots en el sector industrial, ya que es necesario optimizar los recursos disponibles.

Estos robots colaborativos están diseñados con una serie de características técnicas que garantizan la seguridad de un trabajador cuando entra en contacto con el robot. Estas características incluyen materiales ligeros, contornos redondeados y sensores en la base o en las articulaciones que miden y controlan la fuerza y la velocidad para que no excedan un umbral definido en caso de que haya contacto.

Estos robots, su finalidad no es la de reemplazar a los trabajadores, mas bien se utilizan para liberarlos de tareas monótonas y repetitivas y de esta forma permitirles centrarse en trabajos mas complejos, es decir se utilizan para mejorar la productividad. [5]

5.2 RobotStudio

El programa RobotStudio ABB es un programa que permite crear, programar y simular estaciones de robots industriales, diseñado y patentado por la empresa ABB, líder en este tipo de robot. Utiliza el lenguaje RAPID. Tiene un controlador virtual, una copia exacta del software real que emplean los robots en la producción. Esto permite programar un robot en un PC sin necesidad de parar la producción.

Para saber unos datos mas realistas y no tan ideales como los que nos dan los algoritmos, el robot simulado es un ABB IRB 1200. Tiene 6 grados de libertad (GDL), esta diseñado para las operaciones que se han tratado en el proyecto, pick and place.

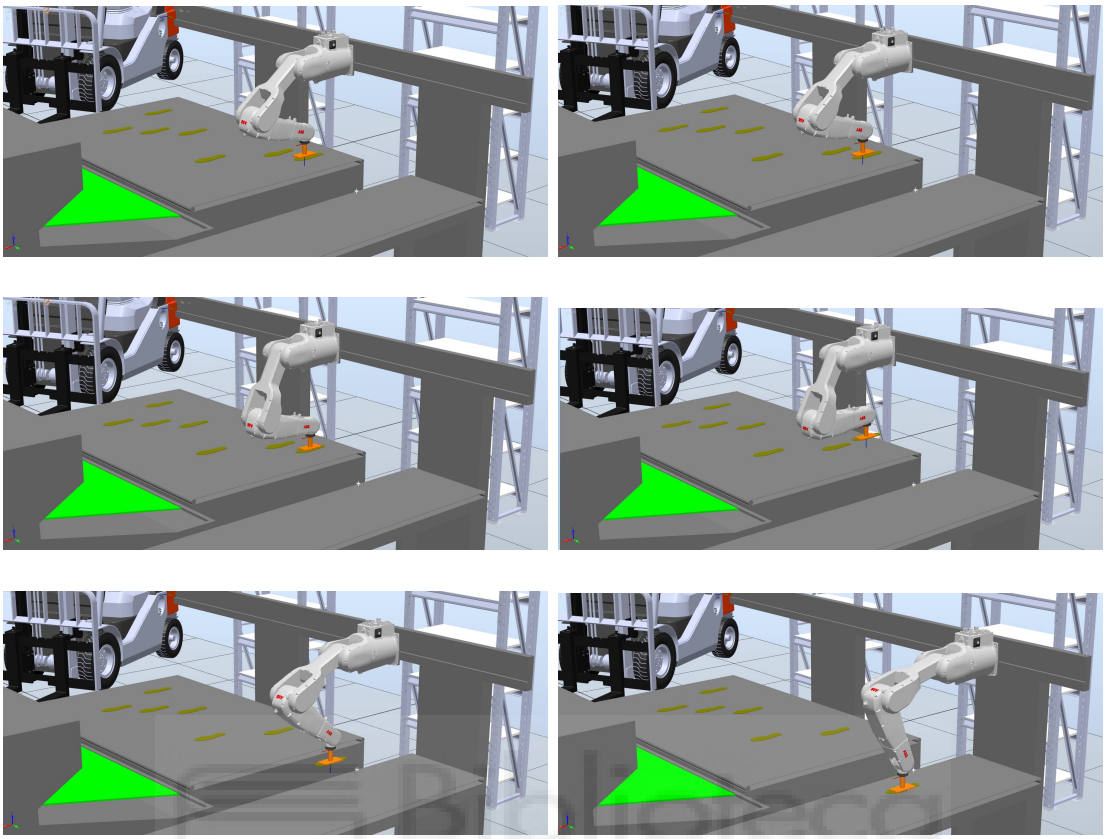


Figure 11: Simulador RobotStudio

5.3 Rapid

Se utiliza el lenguaje RAPID comunicandolo via socket entre python y RobotStudio para poder mandar las secuencias óptimas obtenidas en los algoritmos. RAPID es un lenguaje de programación textual de alto nivel desarrollado por la empresa ABB. Una aplicación de RAPID consta de un programa y una serie de módulos del sistema. Se utiliza para programar estaciones virtuales y las reales. Es una programación secuencial que controlan el robot mediante distintas instrucciones, y por bloques, es decir, se tiene una función main en la cual llamamos a otras funciones dentro de esta.

El programa utilizado en esta simulación es el siguiente:

```
PROC main()

    Read;
    Generate;
    Secuence;

ENDPROC
```

Esta función main, consta de tres funciones que se ejecutan de forma secuencial: Read, Generate y Secuence.

La función **Read()** abrirá tres archivos para leerlos, el de las posiciones de las piezas, el de las posiciones destino donde tienen que colocarse las piezas y el orden de recogida generado por el algoritmo. Tanto para los ficheros de posiciones y el del orden están configurados para que pueda haber el número que se quiera de piezas en la simulación, y para ello se usa un bucle for. Para hacerlo de manera mas directa en el fichero del orden de la simulación, el primer número del fichero es el numero de piezas que hay. De esta manera esta será la condición de parada del bucle.

```
PROC Read()

    Open pickdirectorio, pickfile \Read;
    Open placedirectorio, placefile \Read;
    Open simdirectorio, simfile \Read;

    num_pieces := ReadNum(simfile/Delim:=" ");

    FOR i FROM 1 TO num_pieces DO
        SPiece{i}:=ReadStr(pickfile\Delim:=" ");
        Piece_X{i}:=ReadNum(pickfile\Delim:=" ");
        Piece_Y{i}:=ReadNum(pickfile\Delim:=" ");
    ENDFOR

    SPieceA:=ReadStr(placefile\Delim:=" ");
    PieceA{1}:=ReadNum(placefile\Delim:=" ");
    PieceA{2}:=ReadNum(placefile\Delim:=" ");

    SPieceB:=ReadStr(placefile\Delim:=" ");
    PieceB{1}:=ReadNum(placefile\Delim:=" ");
    PieceB{2}:=ReadNum(placefile\Delim:=" ");

    SPieceC:=ReadStr(placefile\Delim:=" ");
    PieceC{1}:=ReadNum(placefile\Delim:=" ");
    PieceC{2}:=ReadNum(placefile\Delim:=" ");

    FOR i FROM 1 TO num_pieces DO
        order{i}:=ReadNum(simfile\Delim:=" ");
    ENDFOR

    close pickfile;
    close placefile;
    close simfile;
```

```
ENDPROC
```

En el caso de las funciones **Generate()** y **Secuence()**. En la función **Generate()** se crea un array para almacenar las posiciones de las piezas para, posteriormente enviarlas al robot, es decir, el lenguaje para que el robot sepa como moverse a ese punto.

```
PROC Generate()

    FOR j FROM 1 TO num_pieces DO
        PickPiece{j}:=[[altura,Piece_Y{j}, Piece_X{j}],[0,0.707106781,0,0.707106781]
        ,[0,-2,3,0 ],[Pieces_Y{j}, 9E+09,9E+09,9E+09,9E+09,9E+09]];
    ENDFOR

    PlaceA:=[[altura+5,PieceA{2},PieceA{1}],[0,0.707106781,0,0.707106781]
    ,[0,-2,3,0 ],[PiecesA{j}, 9E+09,9E+09,9E+09,9E+09,9E+09]];
    PlaceB:=[[altura+5,PieceB{2},PieceB{1}],[0,0.707106781,0,0.707106781]
    ,[0,-2,3,0 ],[PiecesB{j}, 9E+09,9E+09,9E+09,9E+09,9E+09]];
    PlaceC:=[[altura+5,PieceC{2},PieceC{1}],[0,0.707106781,0,0.707106781]
    ,[0,-2,3,0 ],[PiecesC{j}, 9E+09,9E+09,9E+09,9E+09,9E+09]];

ENDPROC
```

En la función **Secuence()**, con otro bucle se realiza la secuencia y se mandan los comandos al robot para su movimiento y realización de la tarea. En el caso de los movimientos del robot se realizará con el comando *MoveJ*, que mueve el robot hacia un punto usando coordenadas articulares o *MoveL*, que mueve el robot hacia un punto usando la línea recta. La función contiene los siguientes métodos:

```
PROC Place()

    IF Spiece{position}="A" THEN
        Place_A;
    ELSEIF SPiece{position}="B" THEN
        Place_B;
    ELSEIF SPece{position}="C" THEN
        Place_C;
    ENDIF

ENDPROC

PROC Place_A()

    MoveJ offs(PlaceA,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;
```

```
    MoveL PlaceA, v5000,z0,Zapas\WObj:=wobj0;
    WaitTime 0.5;
    MoveL offs(PlaceA,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;

ENDPROC

PROC Place_B()

    MoveJ offs(PlaceB,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;
    MoveL PlaceB, v5000,z0,Zapas\WObj:=wobj0;
    WaitTime 0.5;
    MoveL offs(PlaceB,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;

ENDPROC

PROC Place_C()

    MoveJ offs(PlaceC,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;
    MoveL PlaceA, v5000,z0,Zapas\WObj:=wobj0;
    WaitTime 0.5;
    MoveL offs(PlaceC,-100,0,0),v1000,z0,Zapas\WObj:=wobj0;

ENDPROC
```


5.4 Resultados

En la siguiente tabla podemos ver los resultados de la simulación:

Table 8: Resultados de la simulación (s)

Nº piezas	Raster Order	DTSA	% mejora
10	52.632	52.788	2.9%
25	131.352	131.28	0.5%
35	158.028	153.168	3.1%
50	258.78	224.486	2.4%

Como se puede apreciar, la los tiempos aumentan mucho y la mejora disminuye con respecto a los casos teóricos que se han obtenido con los algoritmos.

6 Conclusión

Como hemos visto en los resultados los algoritmos heurísticos cumplen con el objetivo propuesto al principio del proyecto, mejoran las secuencias del algoritmo usado como referencia, el *raster order*. Considero que son resultados interesantes, ya que aún sacrificando un poco de tiempo de recogida de las piezas se ha conseguido que el DTSA tenga un menor tiempo en la mejor secuencia que el *raster order* y un tiempo computacional menor que el tiempo en el que se tarda en recoger.

Por otra parte, en el caso los dos brazos, como era de esperar, mejora bastante el caso de un solo brazo y podría llegar a ser una solución viable a la hora de aplicarla al *pick & place*. Aunque no mejora como se podría llegar a esperar en un principio.

Aunque en la parte de los algoritmos se hayan tenido unas conclusiones positivas, a la hora de simular en un entorno real las mejoras no son tan buenas, ya que las mejoras disminuyen.

References

[1] Juan José Pérez Hernández. Programación y desarrollo de una estación robótica en robot studio. *Trabajo de Fin de Máster*, 2020.

[2] Luis Eduardo Muñoz Guerrero , Guillermo Solarte Martinez. Algoritmos voraces.

[3] Thomas Stützle , Matthis den Besten and Marco Dorigo. Design of iterated local search algorithms.

[4] Sano Saxena Varshika Dwivedi, Taruma Chauhan and Princie Agrawal. Travelling salesman problem using genetic algorithm.

[5] Sedat Korkmaz , Ahmet Cevahir Cinar and Mustafa Servet Kiran. A discrete tree-seed algorithm for solving symmetric traveling salesman problem. *Engineering Science and Technology, an International Journal*, 23:879–890, 2020.