# Trabajo Práctico Nº 1
# Búsqueda y Optimización

**EJERCICIOS**

**Ejercicio 1**. Dado un punto en el espacio articular de un robot serie de 6 grados de libertad, encontrar el camino más corto para llegar hasta otro punto utilizando el algoritmo A*. Genere
aleatoriamente los puntos de inicio y fin, y genere también aleatoriamente obstáculos que el robot debe esquivar, siempre en el espacio articular.

En primer lugar se muestra un pequeño diagrama de clases del algoritmo A*.

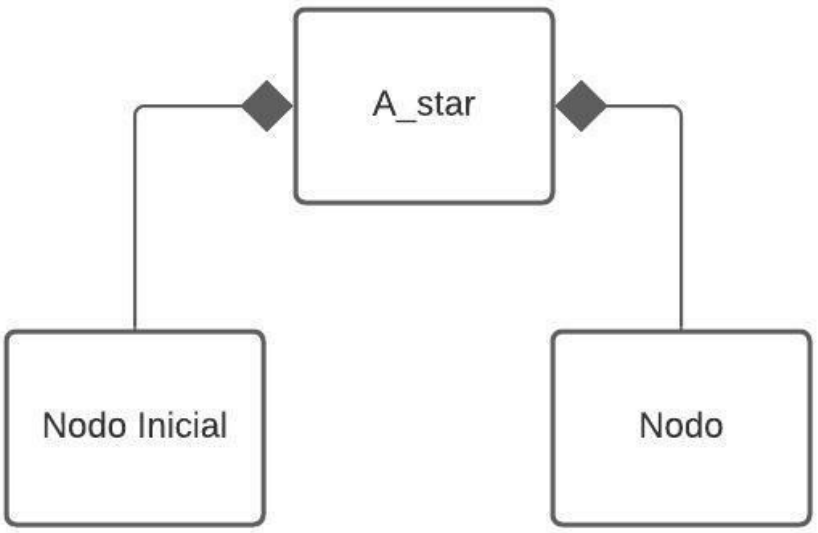



Figura 1.

Como se puede ver en el diagrama, A_star está compuesta por la clase nodo_inicial y nodos, por eso para comprender mejor cómo funciona la clase A_star empezaremos explicando que representan las clases nodos y nodo_inicial:

- nodos(ubicación, nodo anterior): esta clase representa todos los nodos del árbol de búsqueda que son creados por el método neighbors de la clase A_star, asi esta contiene en sus atributos toda la información necesaria para llevar a cabo la búsqueda:
    - ubicación espacial
    - ubicación en la lista de nodos
    - ubicación del padre en la lista de nodos
    - camino_recorrido
    - distancia al punto final
    - F: función de idoneidad de un nodo

- nodo_inicial(ubicación): esta clase tiene como objetivo únicamente representar el primer nodo del árbol. Si bien puede parecer innecesaria, esto no es así, ya que se decidió que para crear los objetos nodos estos tomaran el objeto padre como argumento.

A continuación explicaremos las principales características de A_star():

- A_star(dimensión, paso, obstáculos): como vemos para crear este objeto se necesitan: las dimensiones del espacio de búsqueda, el paso con el que se a discretizar este espacio y los obstáculos que se encuentran en este espacio

- neighbors(ubicación): este método genera los hijos de un nodo, para esto es necesario pasarle la ubicación de este. Como se puede observar este algoritmo trabaja de forma recursiva.

- isobstaculo(nodo_aux, ubicacion nodo_aux): este metodo se encarga de verificar que el nodo (nodo_aux) elegido para ser explorado no sea un obstáculo, y si este es uno, se le asigna un valor bajo de idoneidad para que no se vuelva a explorar.

- isinlistcerrada(nodo_aux,ubicacion nodo_aux): se encarga de verificar que el nodo (nodo_aux) elegido para ser explorado no haya sido explorado anteriormente, y si este ha sido explorado, se le asigna un valor bajo de idoneidad para que no se vuelva a explorar.

- together(nodo_aux, ubicacion nodo_aux): se encarga de verificar que si los nodos inicial y final son vecinos, el camino recorrido para llegar a estos no sea a través de obstáculos.

- end_point(nodo_aux, camino total): se encarga de verificar si se llegó al nodo buscado, si esto es así se devuelve el costo del camino o una lista con el mejor camino entre los puntos inicial y final.

- buscar_camino(punto inicial, punto final, camino total): se encarga de iterar para buscar el mejor camino entre el punto inicial y final. Mediante la función neighbors genera los hijos del nodo elegido como padre y luego busca el nodo con el mejor fitness para elegir el nuevo padre y si sucesivamente hasta que encuentra el punto final.

**Resolución del ejercicio:**

Para resolver el ejercicio se tuvo en cuenta las características físicas del robot IRB140, esto para conocer los límites que presenta nuestro espacio articular. Luego se procedió a elegir dos puntos de manera aleatoria para el inicio y fin de la trayectoria y los puntos considerados como obstáculos.
A la hora de implementar el algoritmo surgieron inconvenientes en cuanto al tiempo de ejecución, cuando se buscaba la trayectoria en el espacio articular de 6 dimensiones con un

paso unitario el algoritmo no llegaba a una solución en tiempos finitos, ya que la complejidad espacial y temporal de estos es factorial. Por lo que se decidió dividir el problema en dos: orientacion y trayectoria, así en un primer lugar se procedió a buscar el camino en el espacio articular que representa la orientación (las tres últimas coordenadas del espacio de 6 dimensiones), para esto se discretizó el espacio con un paso grande, y luego se volvió a realizar una búsqueda entre los puntos obtenidos por la búsqueda anterior pero con un paso más chico, lo mismo se realizó para la orientación.

El resultado obtenido fue el siguiente:

El mejor camino entre [[-116 -60 -188 -32 -96 -264],[20 -12 -124 -8 40 -184]] es:

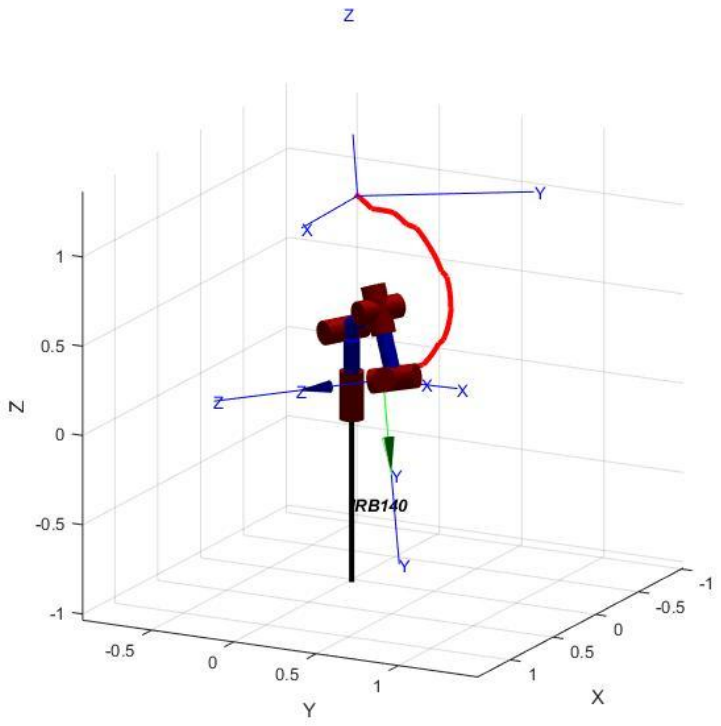



Figura 2.

En la figura 2 podemos ver el camino recorrido por el robot y las posiciones inicial y final que toma este representadas por los ejes coordenados.

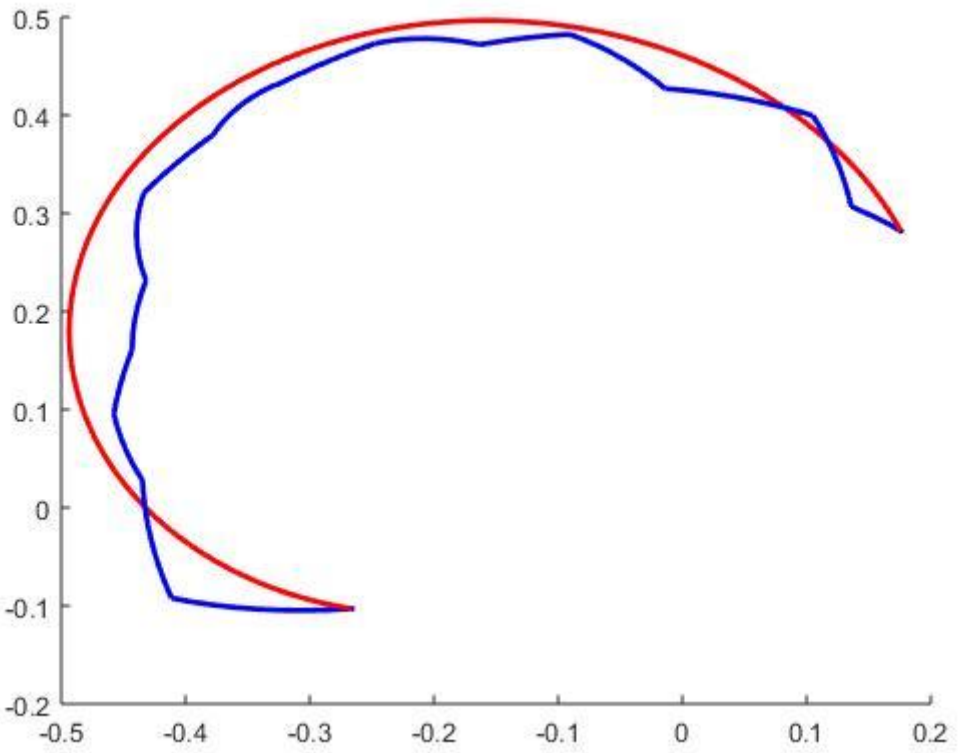



Figura 3.

En la figura 3 podemos ver el camino obtenido por el algoritmo (curva azul) y el obtenido por funciones de un toolbox de matlab. Vemos que si bien el camino es más irregular, debido a las discretizaciones adoptadas, este sigue un buen camino ya que la curva azul es producto de una interpolación en línea recta entre los dos puntos consignas.

**Ejercicio 2.** Dado un almacén con un layout similar al siguiente, calcular el camino más corto (y la distancia) entre 2 posiciones del almacén, dadas las coordenadas de estas posiciones, utilizando el algoritmo A*.

Para llevar a cabo este ejercicio se utilizó el algoritmo A* realizado en el primer punto. Además se creó una clase Almacen y mapa. Donde Almacén contiene toda la información referida a este y mapa es un motor gráfico de este.

A continuación explicaremos la clase Almacen:

- Almacen(col=None,row=None,estantesX=None,estantesY=None,pasillos=None,plot =False): al crear este objeto se puede definir las columnas (col) y filas (row) de estanterías, la cantidad y distribución de estantes por estantería y por último el ancho del pasillo. Así el constructor de este objeto crea las siguientes variables:
    - matriz_deposito: contiene todos los elementos del deposito
    - estanterias: contiene todos los elementos que de las estanterías

    - obstaculos: contiene las posiciones de las estanterias. Variable necesaria para implementar el A*

- getPosicion(_ID): este método permite obtener la posición de un producto en el almacén. Asi, le pasamos el ID del producto y este método nos devuelve las coordenadas del layout donde se encuentra, las cuales son necesarias para llevar a cabo la búsqueda por A*, ya que este es independiente del layout del problema.

Y ahora la clase Mapa:

- mapa(): En esta clase se programó con pygame una serie de funciones y objetos para la visualización del mapa del almacén.
    - Primero se crea una matriz (del tipo lista), que contiene tantos elementos como posiciones hay en el almacén, y en cada posición tiene guardado un número entero que representa el color con el que se va a mostrar. Las estanterías llevan un color distinto de los caminos y además según se haya visitado ya o no la estantería también cambia de colores.
    - Hay un bucle principal que se llama mostrarMapa() que corre en un hilo y se encarga de leer constantemente la matriz antes mencionada y mostrar cada celda con su color correspondiente. En esta función también se accede al objeto LayoutAlmacen para ver qué id de estanterías hay y qué productos hay en cada estantería.
    - Luego por último hay muchas funciones las cuales reciben caminos (secuencia de celdas visitadas) y van mostrando animaciones de cómo se va desde un punto de inicio hasta un punto final recorriendo paso por paso y cuando entramos a una estantería a buscar un producto.

**Resolución del ejercicio:**

Para resolver el ejercicio se creó un layout de características similares a las dadas en el ejemplo mediante la clase Almacen. Luego se eligen dos productos de manera aleatoria y mediante el método getProducto se obtienen sus coordenadas en el layout. Después, se hace uso de la clase A_star y se procede a buscar el camino y, por último, se grafica el camino mediante la clase Mapa (Figura 4.).

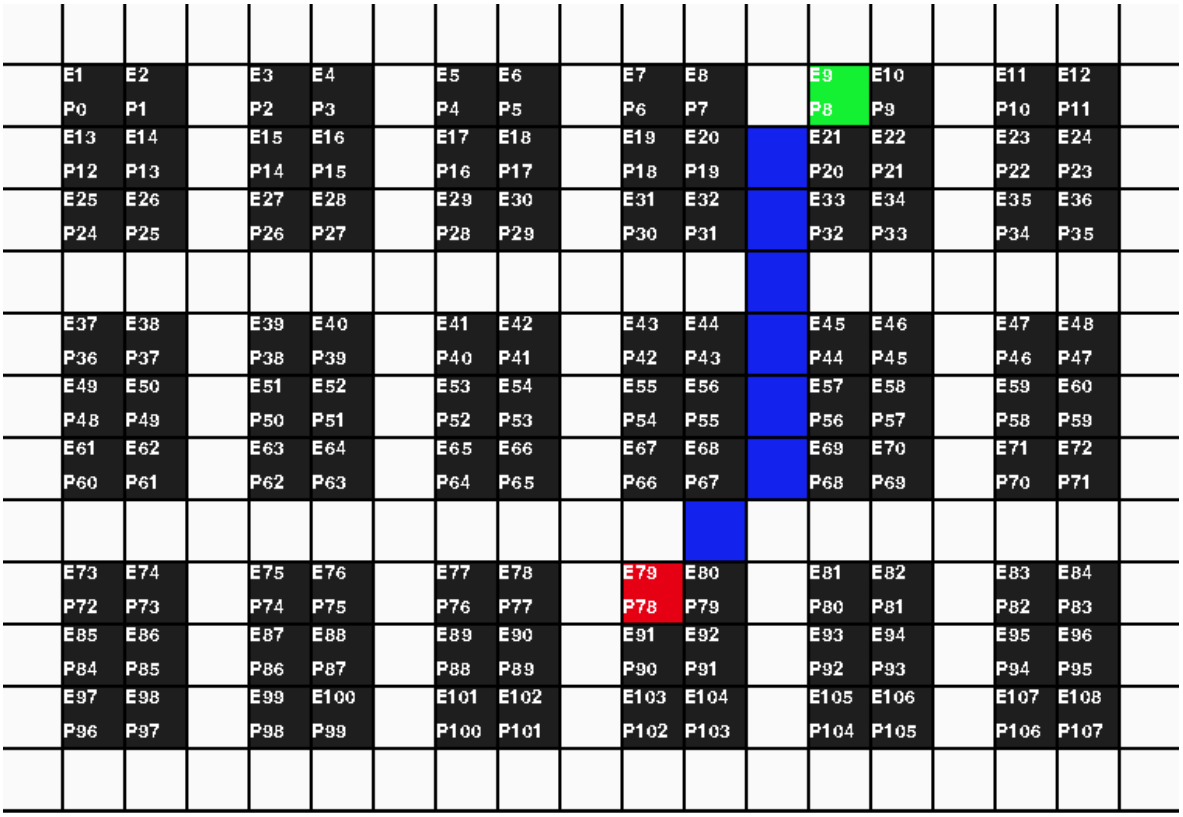



Figura 4.

Resultado:
El mejor camino del producto 8 que se encuentra en la posición [13, 1] al 78 que se encuentra en la posición [10, 9] es:
[[13, 1], [12, 2], [12, 3], [12, 4], [12, 5], [12, 6], [12, 7], [11, 8], [10, 9]]

**Ejercicio 3.** Dada una orden de pedido, que incluye una lista de productos del almacén anterior que deben ser despachados en su totalidad, determinar el orden óptimo para la operación de picking mediante Temple Simulado. ¿Qué otros algoritmos pueden utilizarse para esta tarea?

Explicaremos como funciona el algoritmo de temple simulado realizado:

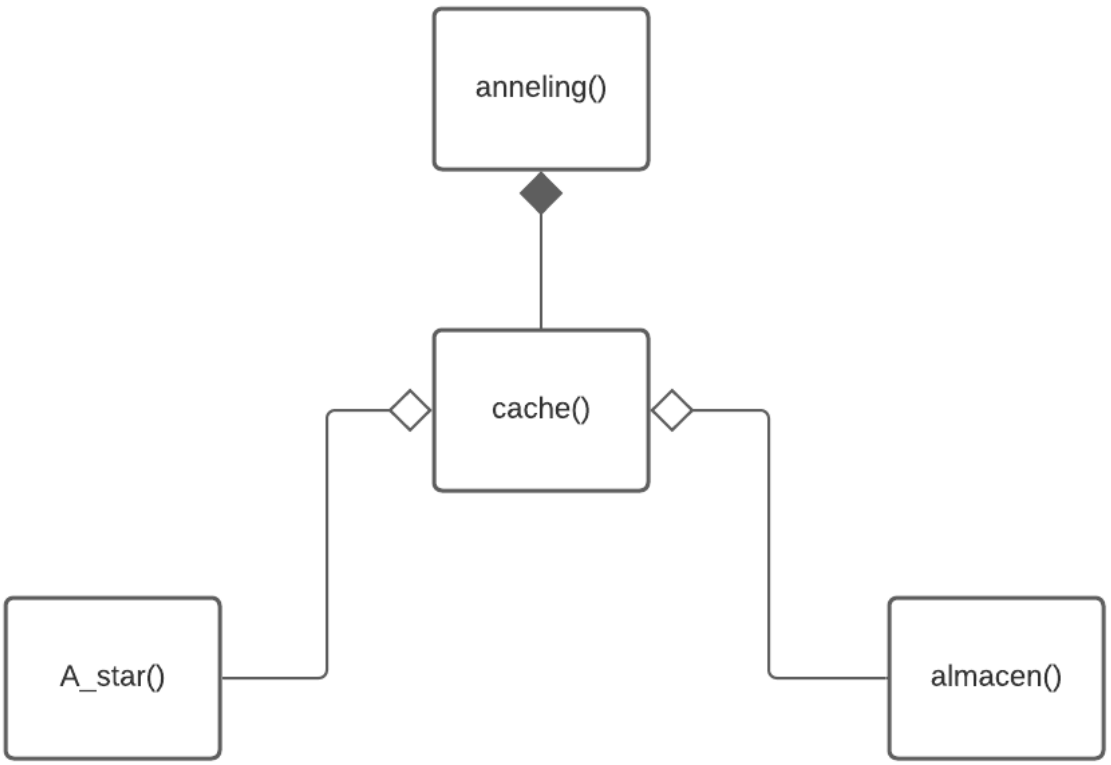



Figura 5.

En la figura 5 podemos observar un diagrama de clase donde se puede observar que la clase annealing se compone de una clase cache la cual agrega a la clase almacen y A_star.

La clase Cache:

- Cache(almac): se encarga de administrar una memoria intermedia entre el temple simulado y el A estrella, esta función es consultada por el temple por la distancia entre 2 puntos del Layout. La función caché tiene varias distancias ya calculadas gracias al algoritmo A estrella, que pre-calcula las distancias y las va guardando en memoria para hacer más rápido el cálculo cuando el temple lo requiera.
- Tiene varias funciones, una para crear el archivo que contiene las distancias (se ejecuta una vez), luego tenemos una función distanciaEntre() que cuando le piden una distancia entre 2 puntos se fija si la tiene en memoria y la devuelve, sino la calcula con A*, la guarda en el archivo y también la devuelve. Al cabo de varias iteraciones ya va a tener muchos valores calculados entonces los tiempos se reducen mucho.

Explicaremos la clase annealing y sus métodos:

- annealing(cache,T): es el constructor de la clase annealing y para crear un objeto de esta clase es necesario pasarle el objeto cache y la agenda de temperatura.
- next_state(): método que nos permite crear un vecino del estado que se está explorando.

- energy(estado,camino_total=True): metodo que nos permite calcular la energía del estado actual, para lo cual hace uso del objeto cache y además permite obtener el camino seguido para recorrer el estado actual.

- probability(): Calcula la probabilidad de elegir un estado malo en función de la diferencia de energía de ese estado y el anterior, y la temperatura actual.

- search(): La función search ejecuta el temple simulado partiendo de la temperatura inicial, con un estado aleatorio y generando estados vecinos, evaluando su energía y así va tomando nuevos estados vecinos y aceptándolos si son mejores o no.

- ley de enfriamiento(): Acá es donde se va decrementando la temperatura conforme pasan las iteraciones, con un hiper parámetro coef y una ley de decrecimiento.

**Resolución del ejercicio:**

Para probar el algoritmo generamos una orden de n elementos de manera aleatoria y ejecutamos el algoritmo varias veces con esa misma orden para así poder verificar la consistencia de las soluciones obtenidas con este mismo.

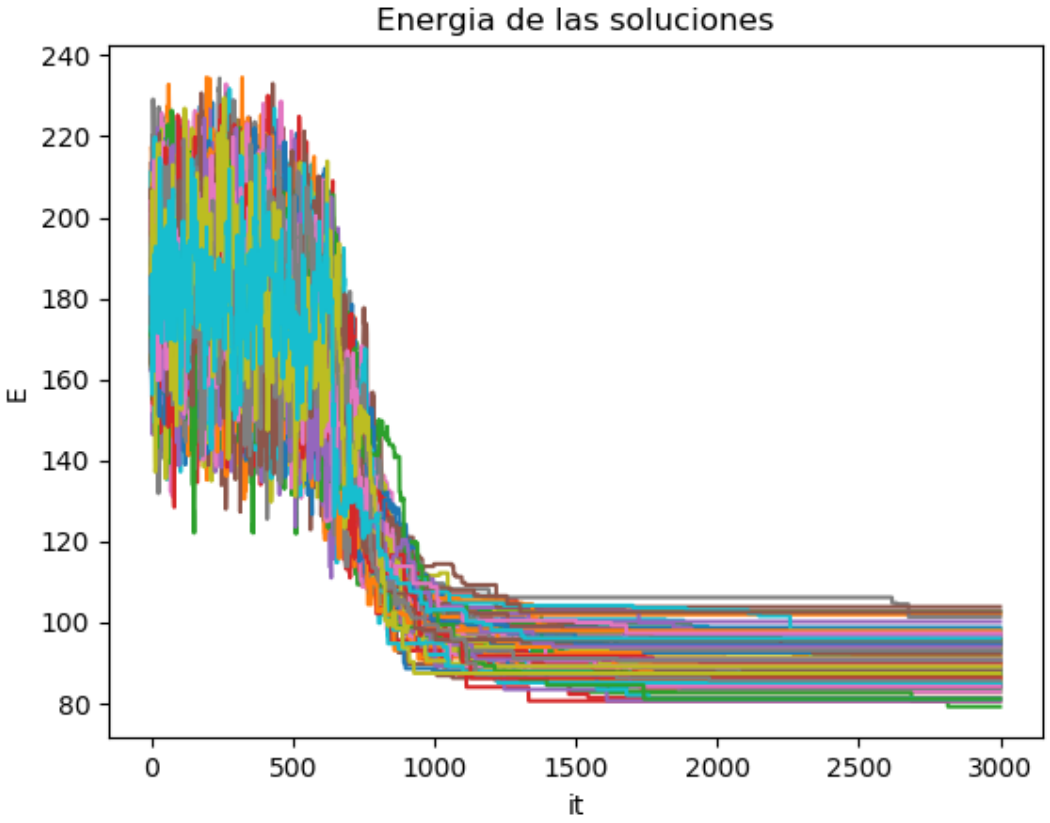



Figura 6.

En la figura 6 podemos ver el resultado de aplicar 100 veces el algoritmo para una orden de 20 elementos. De donde obtuvimos las siguientes estadísticas:

- Promedio: 90.88
- Desviación estándar: 5.70
- Covarianza: 6%

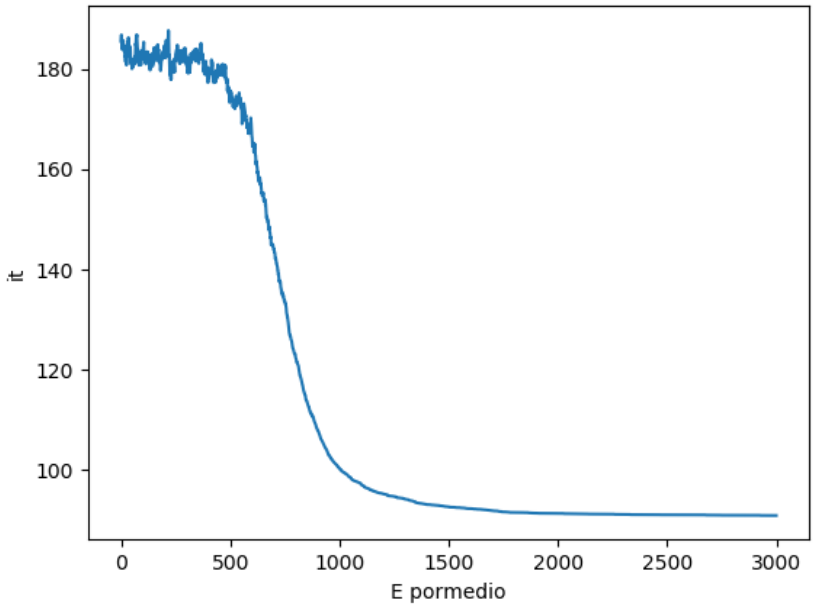



Figura 7.

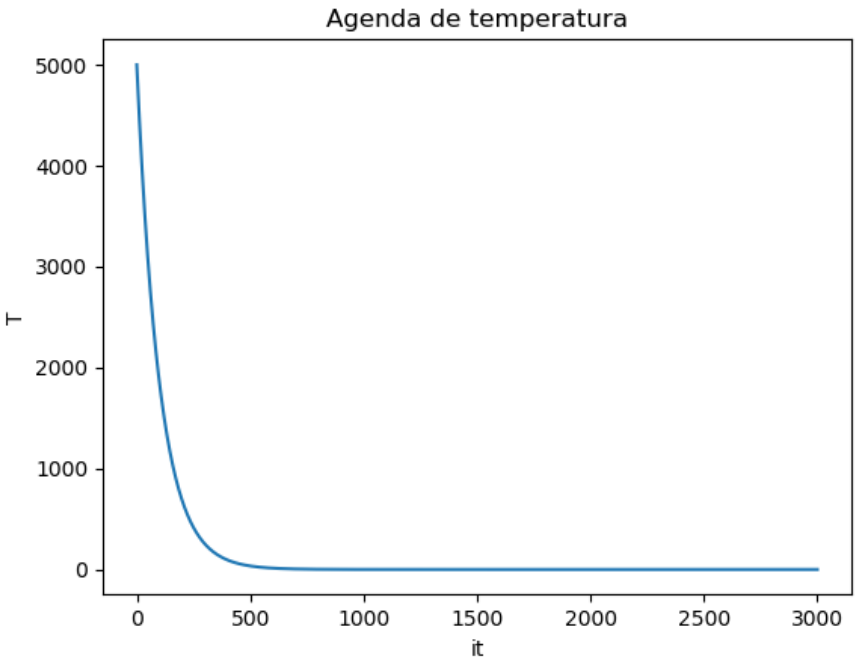



Figura 8.

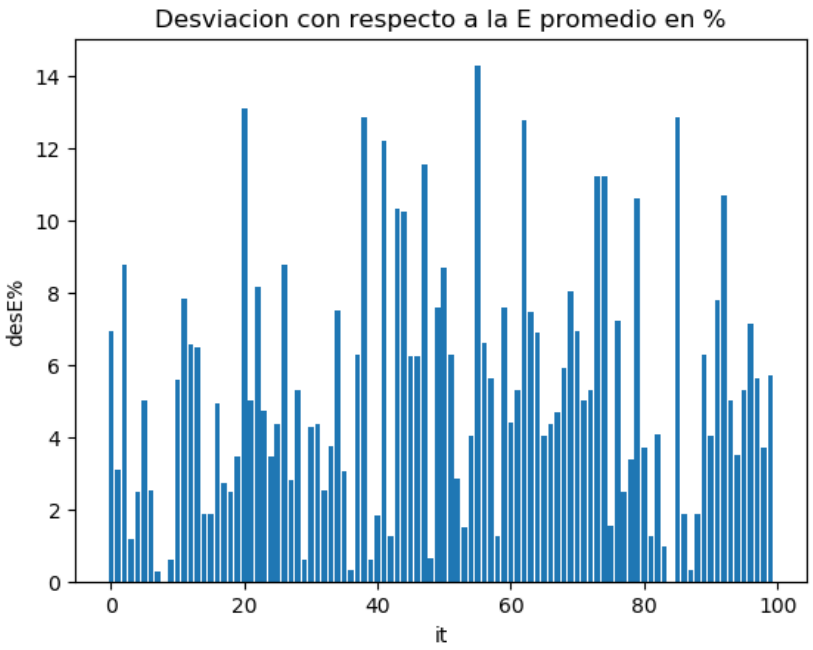



Figura 9.

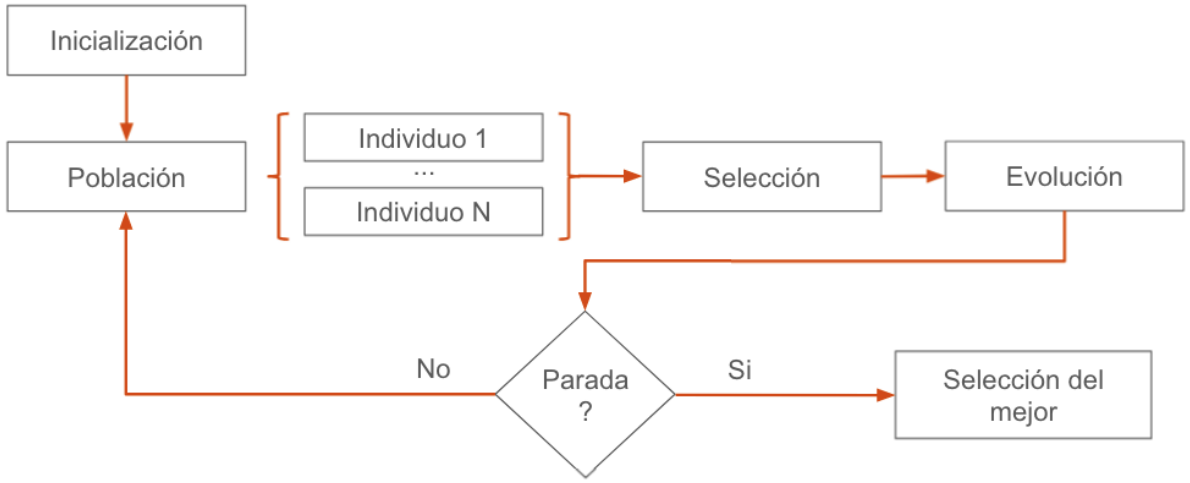
En la figura 7 vemos la curva de energía promedio, en la figura 8 la agenda de temperatura utilizada y en la figura 9 el error relativo de la energía obtenida en cada solución del algoritmo con respecto a la energía promedio calculada, donde vemos que casi todas las soluciones difieren menos del 10% con respecto a la energía promedio calculada.

Por lo que podemos concluir que los resultados obtenidos para este algoritmo de características estocásticas son consistentes ya que presenta una covarianza baja.

**Ejercicio 4.** Implementar un algoritmo genético para resolver el problema de optimizar la ubicación de los productos en el almacén, de manera de optimizar el picking de los mismos. Considere que:

- El layout del almacén está fijo (tamaño y ubicación de pasillos y estanterías), solo debe determinarse la ubicación de los productos
- Cada orden incluye un conjunto de productos que deben ser despachados en su totalidad
- El picking comienza y termina en una bahía de carga, la cual tiene ciertas coordenadas en el almacén (por generalidad, puede considerarse la bahía de carga en cualquier borde del almacén)
- El “costo” del picking es proporcional a la distancia recorrida

Para la resolución de este ejercicio hicimos un algoritmo basándonos en el siguiente diagrama de flujo.




Para esto creamos un objeto/clase genetic() la cual se instancia en un principio con algunos hiperparametros por defecto y conociendo el layout del almacén. Al iniciar la solución por algoritmo genético se llama a process().

- process():
  - Define y crea la población inicial con la que vamos a partir en un principio. Luego empieza un bucle que va a buscar una solución al problema, es decir, se de agarrar los mejores individuos (para esto usa una función de fitness) para después hacer un crossover con esos mejores individuos, luego les aplica una mutación y al cabo de una cantidad de iteraciones termina.

- set_poblacion():
  - Esta función se encarga de crear la población inicial haciendo uso de un genoma que crea de forma aleatoria una asignación válida de los individuos.

- get_best():
  - En esta función hemos programado que se analice de cada individuo, su calidad, a través de una función fitness que mide la distancia que se recorrería para buscar todos los paquetes de cada orden.

- cross_over():
  - Acá en el crossover lo que se hace es el cruzamiento de los padres, para esto se calculan las probabilidades de elegir cada padre en función de cuales son mejores. Luego se agarran esos pares de padres, seleccionados con la función fathers() que agarra una cantidad de padres que mantengan la población constante. Luego por cada par de padres hace el crossover o cruzamiento con la técnica cruce de orden.

- mutacion():
  - Luego se hace una mutación o evolución de los nuevos individuos para poder continuar evolucionando hacia individuos de mejores calidades, asegura que no se pierdan los genes que hacen que el individuo sea una solución completa del problema del Layout.

**Resolución del ejercicio:**

Para resolver este problema utilizamos las órdenes históricas provistas con lo cual obtuvimos los siguientes resultados:

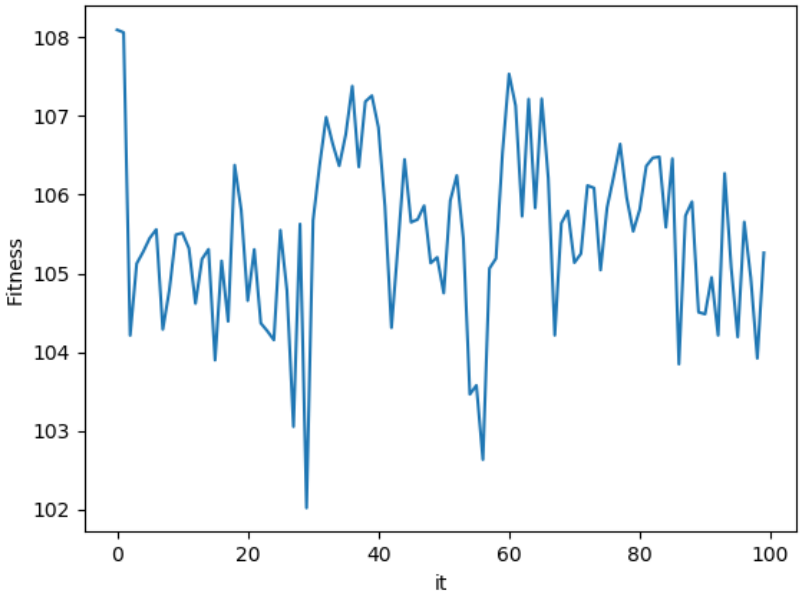



Figura 10.

En un primer momento obtuvimos los resultados observados en la figura 10, como vemos esto no son los deseados, ya que no se ve una tendencia en la mejora del fitness de la solución obtenida. Este resultado se obtuvo para 100 iteraciones, un crossover por cruce y una probabilidad de mutación del 30%.

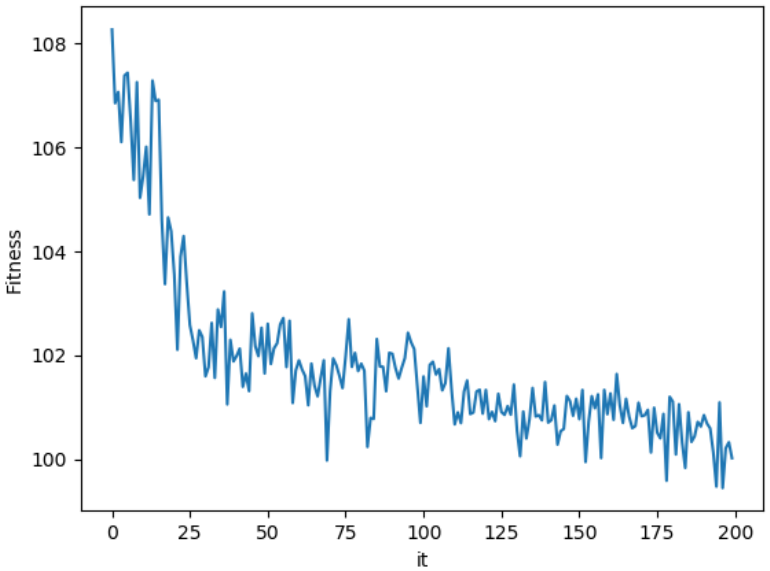



Figura 11.

Luego, ante la no conformidad con los resultados, corrimos el algoritmo con 200 iteraciones,un crossover por cruce y una probabilidad de mutación del 10% y obtuvimos los resultados que se ven en la figura 11. Vemos que esta vez se muestra una tendencia en baja lo que indica que el algoritmo anda.

Sin embargo, el fitness final obtenido no es mucho mejor, pero esto probablemente se debe a las pocas iteraciones realizadas. En nuestro caso no se realizaron más iteraciones debido a la gran complejidad temporal que presenta el algoritmo y los grandes tiempos requeridos por este, para tener una idea, el algoritmo tomó unas 2 horas en realizar las 200 iteraciones. Esta complejidad temporal se da, ya que, como vemos en la figura 12 el algoritmo genético agrega a la clase annealing para poder calcular el fitness de sus individuos, es decir, este algoritmo realiza por cada orden histórica para cada invididuo un temple simulado, osea que si tenemos una población de 20 individuos y 100 orden como dato se realizarán 2000 temples por iteración lo que resulta muy costoso. Y esto sería más complejo si no se hubiera creado el objeto cache que libera casi por completo a la clase annealing de tener que utilizar el A* para calcular la energía de un estado, si no la complejidad temporal se torna mucho más grande.

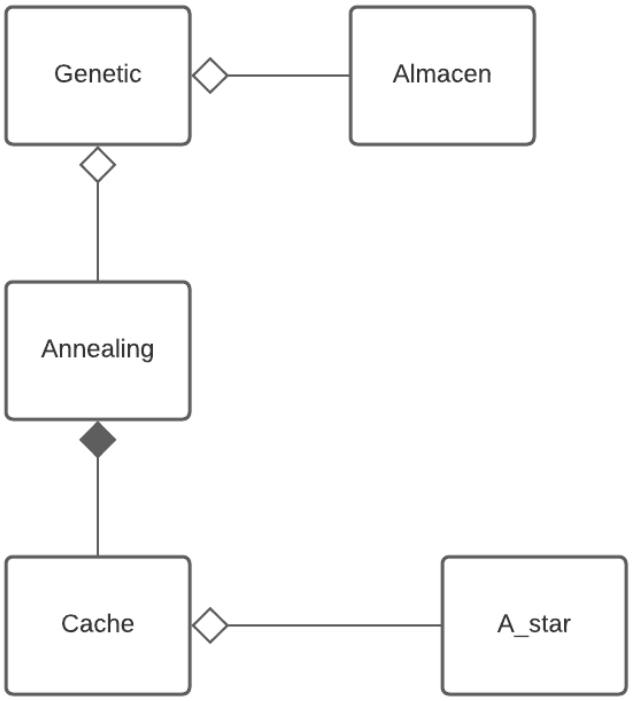



Figura 12.

En la figura 13 podemos ver la disposición del almacén obtenida.

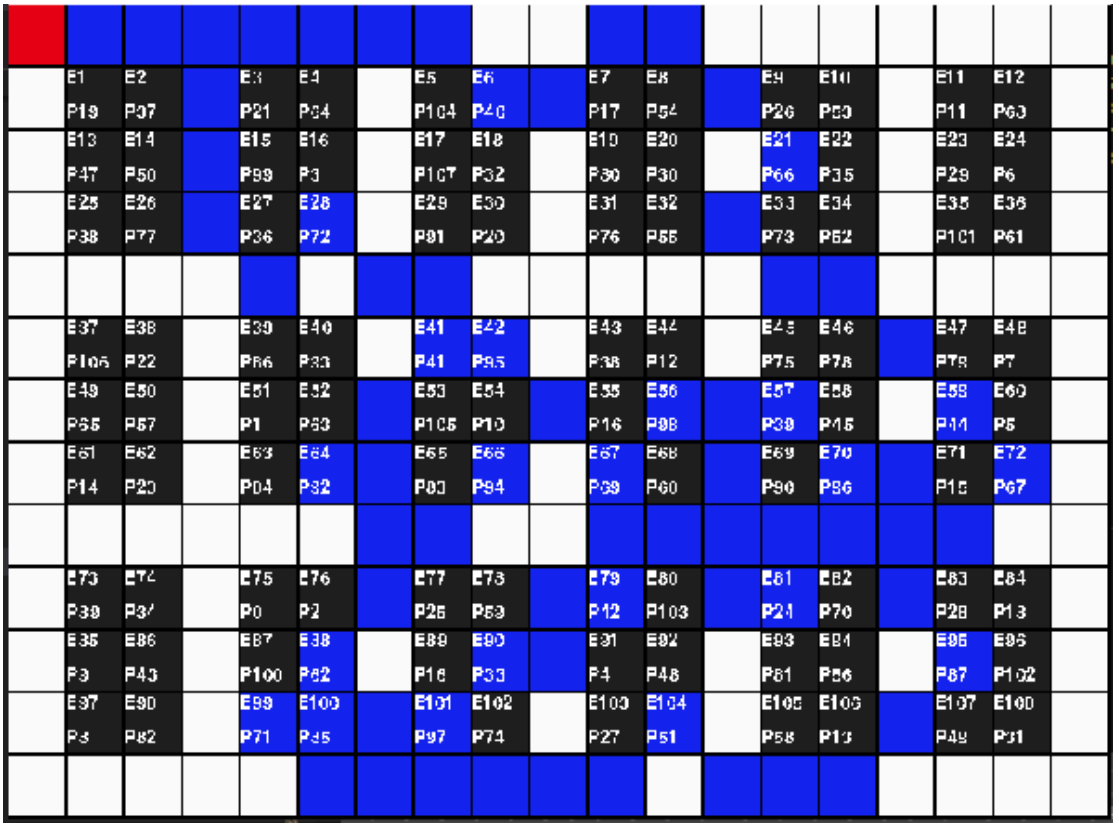


Figura 13.

Por último realizamos el temple simulado con cada orden histórica sobre la solución obtenida contra una disposición cualquiera del almacén.

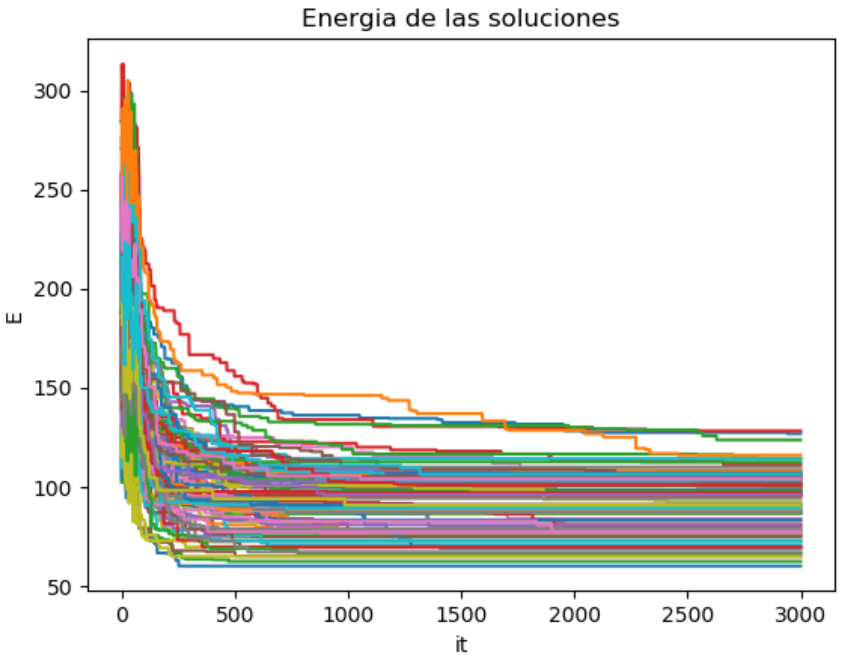



Figura 14.

En la figura 14 podemos ver la curva energía de cada orden para la disposición del almacén calculada y sus valores obtenidos son:

- Promedio: 91.94
- Desviación estándar: 15.0
- Covarianza: 16.31

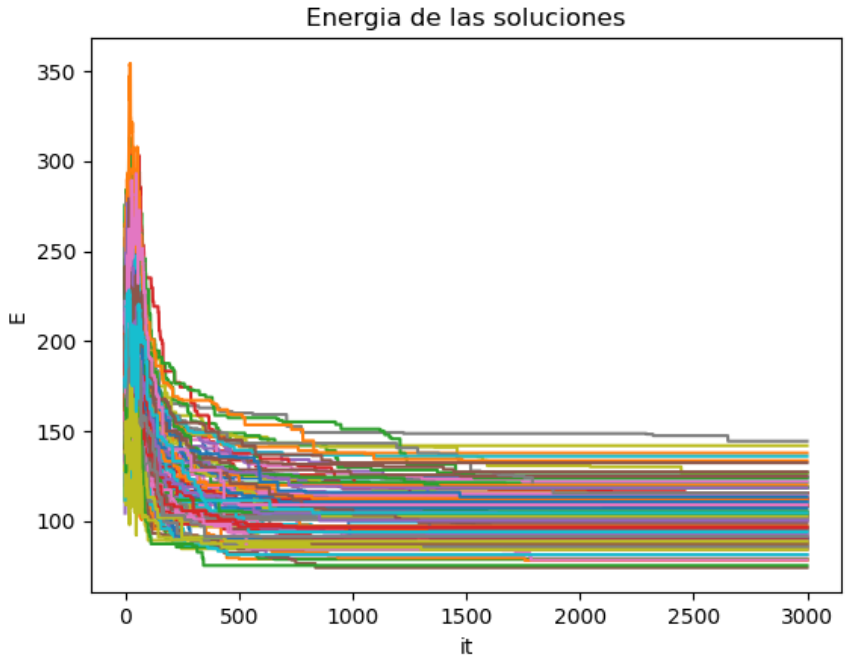



Figura 15.

En la figura 15 podemos ver la energía de cada orden para una disposición del almacén aleatoria y sus valores obtenidos:

- Promedio:103.43
- Desviación estándar: 15.98
- Covarianza: 15.45

Vemos por lo tanto que el resultado obtenido tiene mejores prestaciones en cuanto a promedio pero presenta un poco más de variación..

**Ejercicio 5.** Implemente un algoritmo de satisfacción de restricciones para resolver un problema de scheduling. El problema del scheduling consiste en asignar recursos a tareas. Modele las variables, dominio de las mismas, restricciones, etc. Asuma que:

- Existe una determinada cantidad de tareas que deben realizarse
- Cada tarea requiere una máquina determinada (no todas las máquinas son del mismo tipo)
- Cada tarea tiene una duración específica

- Se dispone de una determinada cantidad (limitada) de máquinas de cada tipo (puede haber más de una máquina de cada tipo)
- No puede sobrepasarse la capacidad de cada máquina en un momento determinado: una máquina solo puede realizar una tarea en un momento determinado

En primera instancia se modela el problema como una industria metalmecánica, en la cual se dispone de distintas máquinas - herramientas para dar solución a diversas tareas o procesos que lo requieren.
La industria modelo posee 5 máquinas de distinto tipo, algunas repetidas hasta dos veces, las cuales pueden resolver hasta 20 tareas diversas, con tiempos de operación aleatorios que van desde 1 hs hasta 20 hs. Cabe destacar que el software obtenido es flexible y admite mayor o menor cantidad de máquinas, tareas y duración de las mismas. Esto puede resolverse cambiando cualquiera de las tres variables mencionadas anteriormente (cantidad de máquinas, cantidad de tareas, duración de tareas).
Se observa que al ejecutar el script “Punto5.py” se modela el problema de acuerdo a lo explicado anteriormente. En cada ejecución se obtendrá un conjunto de 5 maquinas seleccionadas de forma aleatoria de un repertorio de máquinas, como el que se muestra a continuación:

```
tiposmaquinas=['torno','amoladora','mezcladora','trituradora','alesadora','fresadora','CNC','oxicorte','impresora 3D','soldadora']
```

Por otra parte se obtienen las tareas definidas como diccionarios con los siguientes parámetros:

- “id”: identificador de la tarea. Número entero que puede tomar valores de 1 a 20.
- “M”: máquina que requiere la tarea para ser resuelta.
- “D”: tiempo en horas que necesita la tarea para resolverse.

Se observa que los dos últimos parámetros (M y D) se asignan a cada tarea de forma aleatoria, considerando las 5 máquinas disponibles en la industria y el límite de tiempo que cada tarea podría durar (20hs).
Luego se obtiene el tiempo total en el cual se resolverá todo nuestro “schedule”, sumando las duraciones de cada tarea. Y por último se obtiene una lista del dominio temporal, la cual tiene los valores que van de 1 al tiempo total obtenido en el paso anterior.
A continuación se muestra la estructura de los dominios:

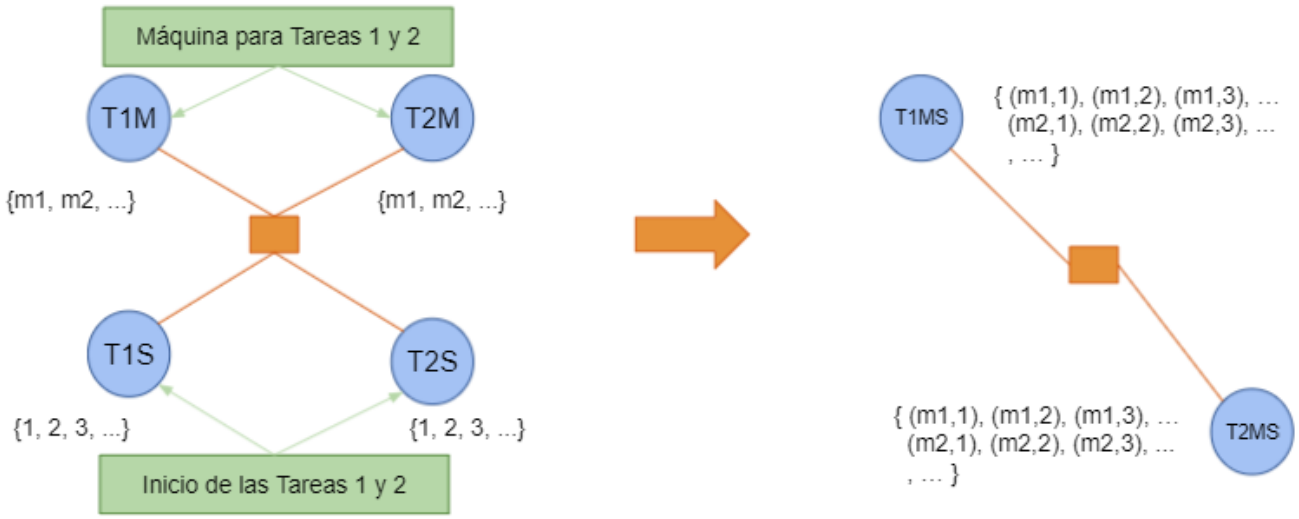
- Maquinas:
  [{'id': 0, 'tipo': 'torno'}, {'id': 1, 'tipo': 'mezcladora'}, {'id': 2, 'tipo': 'oxicorte'}, {'id': 3, 'tipo': 'oxicorte'}, {'id': 4, 'tipo': 'torno'}]

- Tiempos:
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127,

128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240]

Tareas, máquinas disponibles y tiempos, serán las variables que darán origen a una instancia de la clase “csp” (grafo de restricciones del problema) en el script principal. A continuación se detalla la estructura general del problema.

**Estructura del problema: “Grafo de restricciones”**
El grafo de restricciones se estructura del siguiente modo:




Se observa que cada variable (tarea) presenta dos dominios distintos e independientes en principio: máquinas y tiempos de ejecución de cada tarea. Además los dominios son discretos y finitos. Por lo tanto, el problema de satisfacción de restricciones puede reducirse a restricciones binarias, como las que se muestran a la izquierda de la figura anterior. Partiendo del esquema mostrado anteriormente, se creó una clase del tipo “grafo_csp” dentro del archivo “csp.py”. Como se mencionó anteriormente dicha clase será instanciada con los argumentos: tareas, dominio temporal y máquinas. Luego inicializará las tres variables necesarias para definir el grafo de restricciones:

- Variables (X): cada variable es un diccionario estructurado como sigue:

```
TSM={"Tarea":self.tareas[i]['id'],'Maquina':None,'PeriodoInicio':
None,'PeriodoFin':None}
```

donde la clave “Tarea” almacena el id de una tarea dada; “Maquina”, el id de una máquina dada; “PeriodoInicio”, el momento en que comienza la tarea y “PeriodoFin” cuando finaliza.

- Dominio (D): cada elemento del dominio es un diccionario como el siguiente:

```
{"Tarea":self.tareas[i]["id"],'Dominio':dom}
```

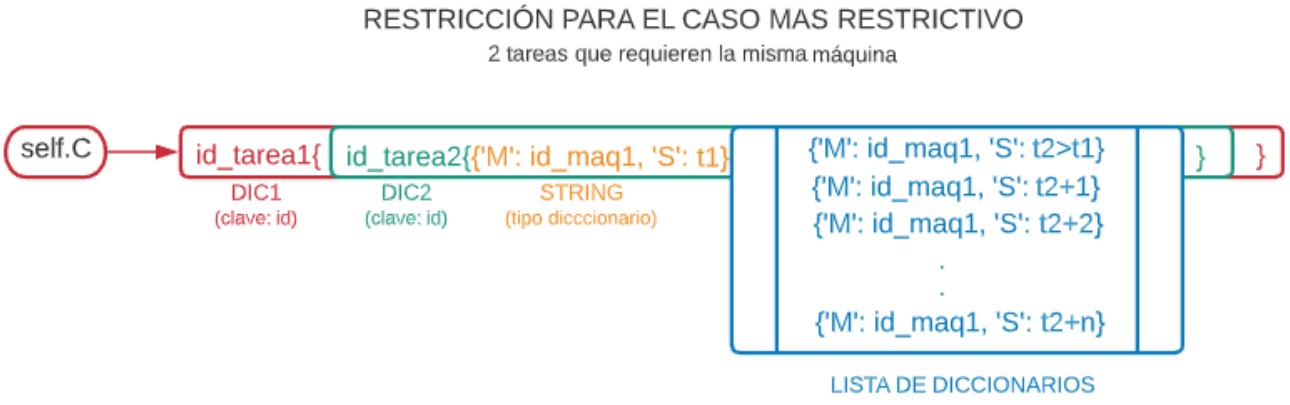
Donde “Tarea” contiene el id propio de una tarea, y “Dominio”,

```
dom.append({'M':M['id'],'S':T})
```

una lista de posibles combinaciones entre la máquina disponible (”M”), necesaria para la ejecución de la tarea, y un tiempo (“S”), en el cual podría utilizarse la máquina.

- Restricciones (C): definidas por extensión dentro de la función definida como “constraint()” propia de la clase “grafo_csp”. Las restricciones se almacenan dentro de una variable del tipo diccionario definida como “C”, la cual contendrá cada restricción en el formato ilustrado a continuación:


La idea general de este formato es acceder a una tarea dada mediante su id correspondiente, el cual es implementado como clave del diccionario ilustrado como “DIC1” en rojo. Al ingresar con el id de la tarea, se desplegará una lista que contiene otros diccionarios, correspondientes a las tareas que presentan alguna restricción con la primera. A modo de ejemplo se listaron 4 tareas, representadas por sus respectivos ids (id_tarea2, id_tarea5, id_tarea9 y id_tarea10).




En la imagen anterior puede verse que dentro de una cualquiera de las tareas restringidas (“tarea2” para el caso), se observa en primer lugar un valor preciso del

dominio de la tarea 1, correspondiente a un id de máquina (id_maq1) y un valor temporal dado (t1), almacenados como cadena de caracteres (color amarillo). Luego, se almacenan los valores del dominio para los cuales podría ejecutarse la tarea 2. Estos se guardan en una lista de diccionarios, representada en color azul. Es decir, se arma una lista que contiene la máquina en cuestión, y los tiempos en que la segunda tarea podría ejecutarse sin generar conflictos.

**Funcionamiento general:**

El algoritmo consta de 3 scripts diferentes que convergen dentro del archivo principal "Punto5.py". El funcionamiento general del software es el siguiente::

1. Generación aleatoria de tareas y dominios.
2. Instanciación de la clase "csp" (archivo" csp.py") ⇒ generación del grafo de restricciónes.
3. Instanciación de la clase "backtrack" (archivo "backtrack.py") ⇒ retorno de asignación completa y consistente de los valores del dominio.
   Cabe destacar que dentro de la definición de esta clase se hace uso de un algoritmo de arco consistencia "ac3" cuyas subrutinas se implementan en un script aparte ("ac3.py").

"backtrack.py" implementa las siguientes subrutinas:

- backtrack():
  La función principal recibe como argumentos:
  1. El grafo de restricciones
  2. Una variable del tipo diccionario "assignament" que contendrá la lista de variables del problema, una lista con los valores que tomen dichas variables, y finalmente una lista "memoria" con el mapeo del problema hasta el nodo explorado, de manera que si en una exploración posterior se detectan inconsistencias, el algoritmo pueda realizar el backtrack necesario para explorar otro camino.
  3. Cantidad de tareas a resolver, es decir, el número de variables involucradas en el problema.

  Se verifica en primer lugar si la variable "assignament" está completa o llena (función "iscomplete"), en cuyo caso finaliza el problema y se retorna la asignación. Caso contrario, se selecciona una variable para asignar un valor del dominio (función "select_unassignedvariable"). La selección se hace según la heurística MRV, selecciona primero la variable más restringida,es decir, la variable que presenta más restricciones. En caso de que haya dos o más variables del tipo MRV, se procede a elegir la variable más restrictiva, es decir, aquella que se vincule con mayor cantidad de variables, de manera que cuando ésta tome un valor consistente, las restricciones se propaguen a más variables simultáneamente, generando mayor probabilidad de que el problema finalice.
  Luego se van seleccionando secuencialmente valores del dominio, y se verifica si el valor tomado se encuentra en el dominio de la variable seleccionada (función "isconsistent"). En caso de que esto sea así, se asigna dicho valor del dominio y se analiza si dicho camino brindará alguna posible solución (función "inference").

“ac3.py” implementa las siguientes subrutinas:

- ac3():
  Recibe un único argumento, el cual consta de una instancia de la clase “csp”, es decir, recibe el grafo de restricciones; que a su vez contiene las variables (X), los dominios (D) y las restricciones (C).
  En principio la función genera una cola que contiene todos los arcos o restricciones existentes entre dos variables distintas. De ésta se irán extrayendo sistemáticamente los pares de valores restringidos y se revisará si existen valores que satisfagan las restricciones entre ambas variables (función “revise”). “revise” procede del siguiente modo:
  1. Toma un valor del dominio para una variable dada Xi.
  2. Verifica dentro de la lista de restricciones (C) si existe algún valor para la variable Xj que satisfaga la restricción existente.
  3. En caso de que exista dicho valor, la función modificará el dominio y devolverá un valor booleando “True”. Caso contrario, el dominio no será modificado y se devuelve el valor “False”.

A continuación puede verse el resultado de la distribución de variables: máquinas y tiempos asignados para la solución de cada tarea. Por otra parte, se observa la lista de máquinas disponibles dentro de la industria. Ejemplo para una corrida aleatoria.

| Tareas\Tiempos | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 amoladora | 3 | 3 | 3 | 3 | | | | | | | | | | | | | | | | | | | | | | | | | |
| 1 impresora 3D | 2 | 2 | 2 | 2 | 2 | 2 | | | | | | | | | | | | | | | | | | | | | | | |
| 2 soldadora | 5 | 5 | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 3 amoladora | | | | | 3 | 3 | 3 | 3 | | | | | | | | | | | | | | | | | | | | | |
| 4 CNC | 6 | 6 | 6 | 6 | 6 | 6 | | | | | | | | | | | | | | | | | | | | | | | |
| 5 amoladora | | | | | | | | | 3 | 3 | 3 | | | | | | | | | | | | | | | | | | |
| 6 CNC | | | | | | | 6 | 6 | 6 | 6 | 6 | | | | | | | | | | | | | | | | | | |
| 7 alesadora | 9 | 9 | 9 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 8 amoladora | | | | | | | | | | | | 3 | 3 | | | | | | | | | | | | | | | | |
| 9 alesadora | | | | 9 | 9 | 9 | 9 | | | | | | | | | | | | | | | | | | | | | | |
| 10 oxicorte | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | | | | | | | |
| 11 amoladora | | | | | | | | | | | | | | 3 | 3 | 3 | 3 | 3 | 3 | | | | | | | | | | |
| 12 CNC | | | | | | | | | | | | 6 | 6 | 6 | 6 | 6 | 6 | | | | | | | | | | | | |
| 13 impresora 3D | | | | | | | 2 | 2 | 2 | | | | | | | | | | | | | | | | | | | | |
| 14 amoladora | | | | | | | | | | | | | | | | | | | | 3 | 3 | 3 | 3 | 3 | 3 | | | | |
| 15 soldadora | 7 | 7 | 7 | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 16 trituradora | Asignar | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 17 oxicorte | | | | | | 1 | 1 | 1 | 1 | 1 | 1 | | | | | | | | | | | | | | | | | | |
| 18 amoladora | | | | | | | | | | | | | | | | | | | | | | | | | | 3 | 3 | 3 | 3 |
| 19 CNC | | | | | | | | | | | | | | | | | | 6 | 6 | 6 | | | | | | | | | |

| Maquinas | |
|---|---|
| 0 mezcladora | Sin Usar |
| 1 oxicorte | |
| 2 impresora 3D | |
| 3 amoladora | |
| 4 trituradora | Asignar |
| 5 soldadora | |
| 6 CNC | |
| 7 soldadora | |
| 8 torno | Sin Usar |
| 9 alesadora | |