Trabajo práctico N°1: Búsqueda y Optimización

Gino Avanzini Emiliano Cabrino Adrián Cantaloube Gonzalo Fernández

Análisis y selección de algoritmos según el tipo de problema

En los problemas enunciados abajo - Qué algoritmo/s utilizaría en cada caso. Justifique. - Defina el problema (estado inicial, modelo de transición, prueba de meta, funciones heurísticas si aplican, función de costo/calidad) - Estime la complejidad del problema y del algoritmo

Diseño de un proceso de manufactura

Es un problema de satisfacción de restricciones: necesitamos encontrar el plan de manufactura. La resolución del problema nos permite crear dicho plan describiendo un estado inicial y un objetivo, siendo ambos fijados por el encargado de la manufactura, y las tareas que se podrán llevar a cabo. Un estado está definido por el instante de tiempo en que cada tarea debe comenzar a realizarse. Cada tarea lleva consigo precondiciones que se deberán cumplir para que esta tarea se lleve a cabo y el planificador deberá diseñar el plan para que la manufactura sea correcta y lo mas fluida posible describiendo el modelo de transición. Al finalizar el algoritmo se calcularán costos de dinero y tiempo de manufactura y se comprobará si la planificación cumple con las expectativas. De otra forma se deberá reconstruir para buscar una mejor solución.

Para encontrar la solución podemos recurrir a algoritmos genéticos o a un algoritmo como el de backtracking.

Planificación de órdenes de fabricación

Es un problema de planificación. Cuando las empresas trabajan con pedidos es común que estos lleguen todos los días de forma desorganizada. Es necesario agruparlos para que la producción se lleve a cabo de forma simultánea. Por ejemplo: llegaron pedidos de un mismo modelo, de clientes diferentes, ellos pueden ser cortados y fabricados al mismo tiempo. Con ese agrupamiento se estará elaborando una única orden de fabricación. Ahorrando tiempo al evitar un seccionamiento innecesario.

El orden del plan y el ahorro a la hora de planificar se pueden basar según el valor de facturación o según el plazo y cantidad de productos.

Para poder resolver este problema podemos usar un algoritmo de temple simulado. Un estado sería una asignación completa de tiempos en los que deben comenzar los procesos de manufactura de cada orden. Además verificamos que las asignaciones sean válidas mediante algoritmos de consistencia de arco y nodo. Como función de energía se puede tener en cuenta el tiempo total de ejecución de todas las órdenes, los tiempos muertos, etc. Se buscaría minimizar esta función de energía, premiando los trabajos que se realizan en paralelo y penando aquellas asignaciones que implican mayor tiempo total de ejecución.

Encontrar la ubicación óptima de aerogeneradores en un parque eólico

Es un problema de optimización: Se busca maximizar la energía generada por un determinado número de aerogeneradores en un campo eólico dada la información sobre el viento en la zona. Para lograrlo hay que determinar la mejor disposición de los generadores en el parque.

La resolución del problema se puede encasillar dentro de la familia de algoritmos de búsqueda continua. Además se puede discretizar el espacion y convertir el problema a uno de búsqueda clásico. Entre ellos se encuentran, por ejemplo, los algoritmos genéticos y de recocido simulado. El estado inicial del problema es obtener un configuración aleatoria de los generadores en el espacio del campo. El modelo de transición consiste en reubicar alguno de los aerogeneradores en otra posición del campo y proseguir con el algoritmo, evaluando si el nuevo estado produce una mejora o no con la función objetivo.

La función de energía (en el caso del temple) es una función que dada la información del viento en la zona, valora qué tan buena es la distribución de generadores. Para generar estados vecinos se puede reubicar cierta cantidad de generadores y evaluar la nueva configuración.

Además, si se quisiera, se podría fraccionar el campo eólico en sectores y hacer optimizaciones locales en esos sectores. De esta forma se podría paralelizar el problema y obtener una solución más rápidamente.

Planificar trayectorias de un brazo robótico con 6 grados de libertad

Es un problema de planificación y/o optimización: Se busca una combinación entre ambas, planificación para conocer las acciones que debe tomar, y optimización para llegar del estado inicial al final con el menor gasto posible. Ambas nos ayudarán a tomar decisiones en tiempo real ya que trabajan en conjunto.

El estado inicial y final son descriptas por el operario y no generadas aleatoriamente. Con el modelo de transición vemos el movimiento de los eslabones del robot para llegar al próximo estado (punto discretizado de la trayectoria). Podríamos usar la heurística del ejercicio 2)a de este práctico (A*) para llegar al objetivo. En este caso no podríamos usar algoritmos de búsqueda local ya que no solo debemos llegar al estado final, sino que necesitamos seguir una trayectoria establecida.

Diseño de un generador

Los problemas de diseño generalmente son de optimización. Esto significa que debemos encontrar la forma de diseñar el generador maximizando el ahorro de recursos y el rendimiento del mismo. Para que el generador cumpla con las especificaciones deseadas, habrá que implementar un algoritmo relacionando las propiedades y cantidades de los materiales a utilizar para saber que tanta eficiencia aportarán al generador. La combinación mas optima es la que utilizaremos como meta.

Para la resolución de este problema podemos utilizar un algoritmo genético. Cada individuo estaría caracterizado por, por ejemplo, cantidad de polos del rotor, tipo de polos, cantidad de vueltas por bobina, material del núclo del estator y el rotor, potencia de salida, etc. Para cada configuración habrá un valor de fitness que será función del costo de construcción, rendimiento estimado y peso, entre otros.

Definición de una secuencia de ensamblado óptima

Es un problema de búsqueda. Se quiere encontrar la secuencia óptima de partes a colocar en, por ejemplo, una placa electrónica.

Para esto podríamos utilizar un algoritmo como el A* en el cual partimos de un componente inicial y queremos buscar la mejor secuencia para colocar todos los componentes, considerando que puede haber varios unidades del mismo tipo en distintos lugares de la placa.

La heurística sería la distancia en línea recta entre un componente y otro. El costo aumenta al realizar cada viaje a un punto de la placa y al realizar un cambio en el tipo de componente.

Dado que buscamos optimalidad con un A* obtendremos la mejor secuencia.

Planificación del proyecto de una obra

Es un problema de planificación. Se busca encontrar un orden en las tareas a realizar (algunas de forma paralela o simultanea) de manera que se cumplan los ordenes de precedencia de cada una de ellas. Se inicia con las tareas de la obra que no requieren requisitos previos para ser ejecutadas. Se procederá con las tareas que requieran la finalización de la/s primera/s y así sucesivamente hasta concluir la obra. Cada tarea conlleva cierto tiempo, parámetro que utilizaremos para calcular las ganancias y estimar el rango de tiempo que nos llevará a cabo realizar la obra.

Se puede utilizar un algoritmo de backtracking junto con un algoritmo para verificación la consistencia de las asignaciones (AC3, etc.)

Implementación de algoritmo A* para trayectoria de robot de 6 grados de libertad

El problema del robot con 6 grados de libertad fue modelado en el espacio articular, donde cada estado del problema es el valor angular de cada articulación con una discretización de ángulos sexagecimales enteros, es decir, un vector de estado es una lista de 6 componentes enteros.

La función heurística adoptada en el modelo es la raíz cuadrada de la suma del cuadrado de cada componente, distancia euclidiana del espacio de estados, desde el estado en cuestión al estado objetivo.

El estado inicial es sencillamente la configuración inicial del brazo robótico. La prueba de meta es la comprobación de si el estado actual es igual al estado objetivo, es decir, comprobar si el valor angular de cada articulación es igual el objetivo.

Análisis de tiempo de ejecución del algoritmo en función de la cantidad de grados de libertad

El tiempo de ejecución del algoritmo se incrementa al aumentar la cantidad de grados de libertad del espacio articular del sistema. A continuación se analiza de 2 a 6 grados de libertad.

```
from time import time

from a_star2b import Node, a_star
from a_star2a import generate_neighbours, obstaculos
```

Los parámetros TOP y STEP están definidos en el archivo a_star2a.py.

```
TOP = 50
STEP = 10

register = []

i = 2
while (i <= 6):

    GDL = i

    initial_state = GDL * [0]
    start = Node(initial_state)

    end_state = GDL * [TOP]
    goal = Node(end_state)

    # wall = obstaculos(GDL)
    # wall = None

    t_start = time()
    # ans = a_star(start, goal, generate_neighbours, wall=wall)
    ans = a_star(start, goal, generate_neighbours, dim=GDL)

    if ans[1]:
        t_end = time()
        register.append(t_end - t_start)
        i += 1
```

```
1555960528.8629613
2
1555960528.8864892
3
1555960529.4057345
4
1555960551.4310658
5
```

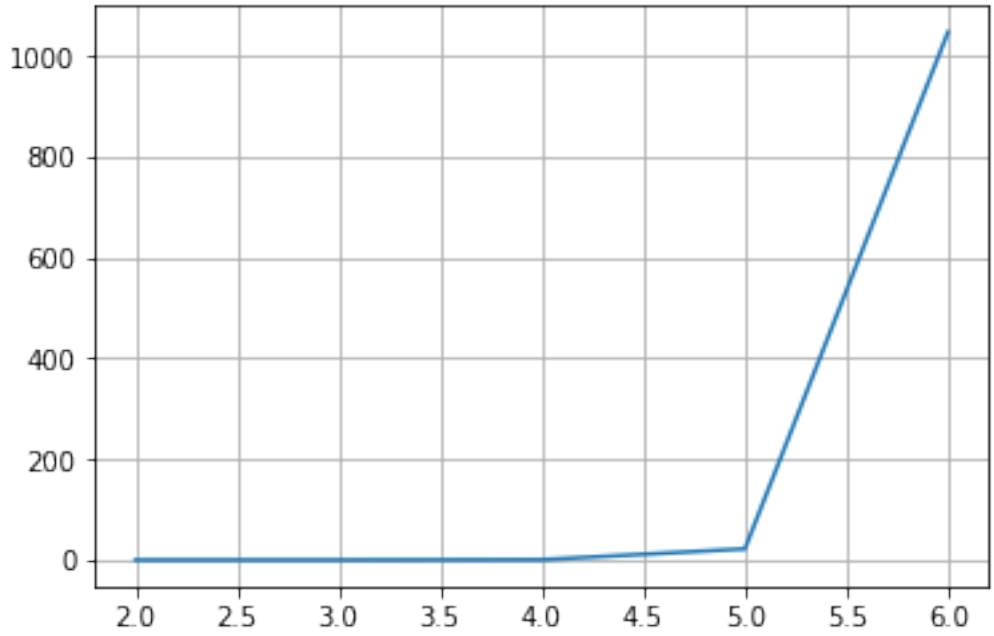
```
1555961599.8360023
6
```

```
from matplotlib import pyplot as plt
```

```
print(register)
```

```
[0.0005648136138916016, 0.02310776710510254, 0.5191671848297119, 22.02523684501648, 1048.4048516750336]
```

```
gdl = [2, 3, 4, 5, 6]
plt.plot(gdl, register)
plt.grid()
```



Si se analiza el tiempo de ejecución en relación del tiempo de ejecución máximo, es decir para 6 grados de libertad, los resultados son los siguientes:

```
for i in register:
    print(i / register[len(register)-1])
```

```
5.387361695142868e-07
2.2040881505062974e-05
0.0004951972360679559
0.02100833166674765
1.0
```

Conclusión

Para observar el crecimiento exponencial, se muestra cuánto más grande es un tiempo de ejecución con relación al tiempo para un grado de libertad menos:

```
for i in range(0, len(register)-1):
    print(register[i+1]/register[i])
```

```
40.912199240185736
22.467215567317712
42.42416987938252
47.600162443304214
```

Se puede observar que incrementar en 1 la dimensión del espacio articular del robot implica escalar aproximadamente en 40 el tiempo de ejecución del programa.

La complejidad del algoritmo A* es O(bˆd) siendo b la cantidad de estados vecinos y d el factor de ramificación o profundidad.

Por lo tanto, para reducir el tiempo de ejecución debería modificarse la generación de estados vecinos aumentando su cantidad, y así obtener un árbol de estados más ancho pero de menor profundidad.

Implementación de algoritmo A* para calcular camino más corto entre dos posiciones de un almacen

El problema del camino más corto entre dos posiciones de un almacen con un determinado layout, se modeló de tal forma que cada estado es una posición del almacén en coordenadas rectangulares. Estas coordenadas son tales que el (0, 0) se encuentra en la esquina superior izquierda, y x e y crecen hacia la derecha y abajo respectivamente.

La función heurística adoptada en el modelo es la raíz cuadrada de la suma del cuadrado de cada componente, distancia euclidiana del espacio de estados, desde el estado en cuestión al estado objetivo. Esto debido a que ya se encontraba implementado en el ejercicio anteriror.

El estado inicial es la posición inicial, por ejemplo (0, 0), y el estado final la posición objetivo. Se consideran posiciones válidas únicamente los pasillos, por lo que la posición de una determinada estantería o producto es ubicado estando en el pasillo frente a éste.

El algoritmo se implementó separando las funciones independientes del modelado del problema de las dependientes de éste. Por lo tanto aquellas funciones propias del algoritmo A* pudieron reutilizarse sin inconvenientes en ambos ejercicios sin necesidad de una nueva implementación. Para información más específica consultar documentación del código.

Implementación de recocido simulado para optimización de orden de picking en almacen

En este problema el objetivo es determinar el orden óptimo de picking para una serie de productos en el almacén. Esto se determina mediante un algoritmo de recocido simulado cuyos parámetros e implementación se explicarán a continuación.

Análisis del algoritmo

El algoritmo de recocido simulado tiene varios parámetros que afectan los resultados. A continuación hablaremos brevemente de los mismos.

- Perfil de decrecimiento de temperatura

Como perfil de decrecimiento se eligió, por simplicidad y para evitar el tuning de otro hiperparámetro, una función lineal en la que la temperatura disminuye un grado por cada iteración.

- Temperatura inicial

Se evaluaron distintas temperaturas iniciales y se hizo un análisis en el que se evalúa cuán mejor es la solución final que otorga el algoritmo en comparación a la solución inicial. Dicho análisis se explica más adelante pero la conclusión es que no existe una gran mejora en la calidad de la solución con temperaturas iniciales mayores a 200.

- Energía

Como función de energía para obtener la calidad de la solución para cada estado se utiliza el algoritmo A* ya implementado en el problema anterior. Para cada secuencia de coordenadas que visitar, se calcula la longitud del camino óptimo para ir de una coordenada a la siguiente dentro de la secuencia. El valor de la energía del estado será la suma de las distancias recorridas para ir de una coordenada a la otra, comenzando y finalizando en [0, 0].

- Generación de vecinos

Un estado en el algoritmo es una lista con una secuencia de coordenadas que visitar. Para generar vecinos se seleccionan las posiciones de dos elementos aleatorios dentro de la lista de coordenadas y se intercambian. Luego se calcula la energía del estado vecino y este se acepta siempre que tenga menor costo y cuando tenga mayor costo se acepta con probabilidad exp(-delta/T).

Modelado del problema

El input del programa es una lista de productos que hay que ir a buscar y como salida se obtiene el orden óptimo del picking para minimizar la distancia recorrida. Antes de la ejecución en sí del temple, se realiza una abstracción en la que se convierte de número de producto (o posición en el almacén) a la coordenada cartesiana a la que hay que ir. Esto se hace para poder reutilizar el algoritmo ya usado del A* y para, además, poder usar a futuro este mismo código para el algoritmo genético.

Luego, a cada orden se le agregan dos coordenadas indicando que debe partir y volver a un punto determinado del almacén el cual es, en nuestro caso, [0, 0]. Así, por ejemplo, para ir a buscar los productos 8, 6, 2, 14, 30 y 17 obtendremos algo así:

```
from simulated_annealing import map_to_coord, distance, temple_simulado, neighbours_annealing
```

La secuencia de coordenadas, con su respectiva energía, antes de la ejecución del algoritmo es:

```
pos_pick = [8, 6, 2, 14, 30, 17]
coordenadas = map_to_coord(pos_pick)
print("Estado inicial: ", coordenadas)
print("Energía: ", distance(coordenadas)) # Energía del estado inicial
```

```
Estado inicial:  [[0, 0], [6, 0], [4, 0], [2, 0], [9, 0], [9, 3], [1, 6], [0, 0]]
Energía:  40
```

Y luego de la ejecución, la secuencia de coordenas con su respectiva energía es:

```
T0 = 200
best_path = temple_simulado(coordenadas, T0, neighbours_annealing, distance)

print("Estado final: ", best_path[0])
print("Energía: ", best_path[1]) # Energía del estado inicial
```

```
Estado final:  [[0, 0], [2, 0], [4, 0], [6, 0], [9, 0], [9, 3], [1, 6], [0, 0]]
Energía:  32
```

Primero se genera la secuencia de coordenadas a las que hay que ir para buscar los productos tal como se dieron en la orden y el costo (en pasos) de busar los productos en dicha secuencia. Al finalizar la ejecución del algoritmo se devuelve la secuencia de coordenadas "óptimas" a las que hay que ir junto con el costo del nuevo camino.

La solución obtenida en el algoritmo no es la final que se obtendría con el algoritmo vanilla sino la solución con mejor costo de todos los estados generados. Como nuestro objetivo es optimizar el picking, lo único que nos interesa es obtener el mejor resultado y no el último obtenido.

Para calcular el costo de la secuencia se utiliza el algoritmo A* para obtener el camino óptimo al pasar de la primera coordenada de la secuencia a la segunda, de la segunda a la tercera y así sucesivamente. En cada ejecución del A* se suma la distancia recorrida al ir de una coordenada a la otra y la suma total será el costo de la secuencia dada. Como vemos en el ejemplo dado, el costo del camino disminuyó de 40 a 32 pasos.

Análisis de duración de la ejecución vs cantidad de productos en una orden para temperatura inicial constante

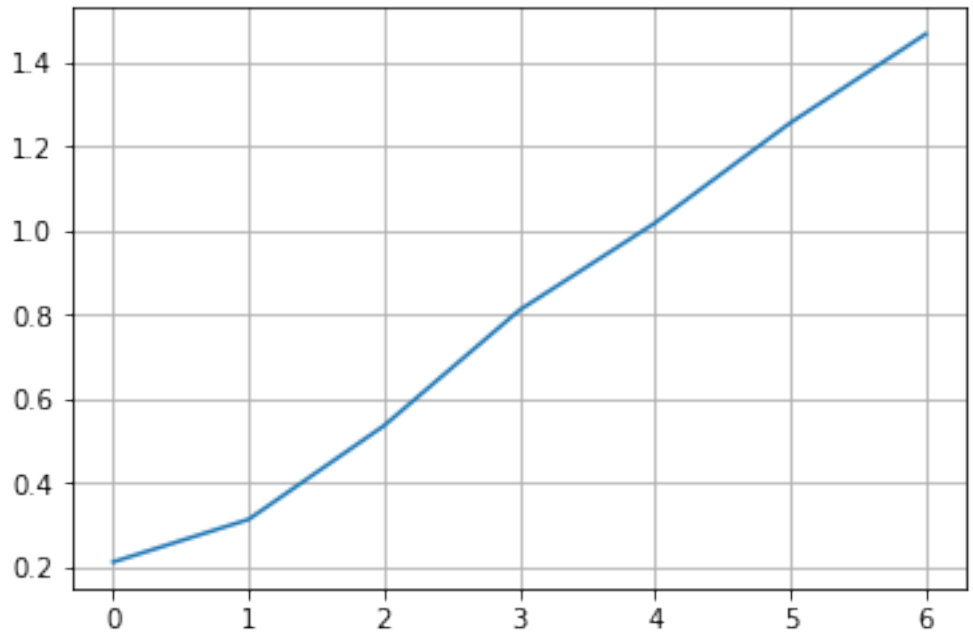
En este análisis se ve la complejidad temporal del algoritmo en función de la cantidad de productos que tenga una orden para temperatura inicial constante. Así es que se calcula el tiempo de ejecución del algoritmo para órdenes con 3, 5, 7, 10, 13, 15, 20, 25 y 30 productos. Para eliminar las variaciones debido a la aleatoriedad en la generación de las órdenes, se toma, para cada tamaño de orden (3, 5, 7, etc.) el tiempo promedio de ejecución luego de 50 iteraciones. Esto es, se hace un promedio de tiempo con 50 ejecuciones para cada tamaño de orden.

```
from time import time
from random import randint

ITER = 50
T0 = 200

# Cantidad de productos en una orden
inputs = [3, 5, 10, 15, 20, 25, 30]
```

```
time_exec = []

for i in inputs:
    time_avg = 0
    pos_pick = []

    for j in range(0, ITER):
        pos_pick = [randint(0, 31) for i in range(0, i)]

        coordenadas = map_to_coord(pos_pick)

        start = time()
        _ = temple_simulado(coordenadas, T0, neighbours_annealing, distance)
        end = time()

        time_avg += end - start

    time_avg = time_avg/ITER
    time_exec.append(time_avg)
```

Se obtiene, como es de esperarse, un perfil lineal. Duplicar la cantidad de productos en la orden implica duplicar el tiempo de ejecución.

```
from matplotlib import pyplot as plt

plt.plot(time_exec)
plt.grid()
```



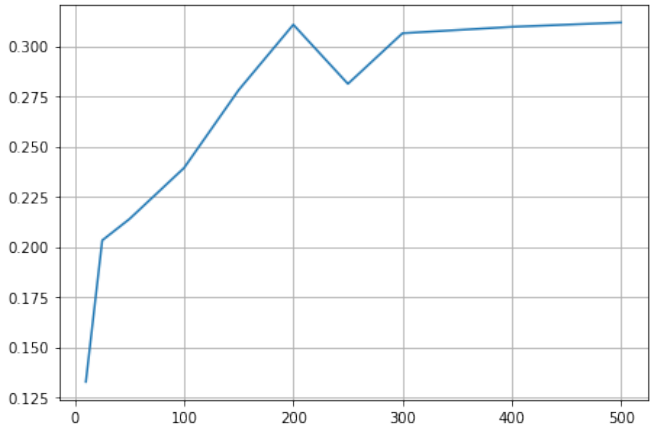
Análisis de temperatura inicial (cantidad de iteraciones) vs mejora en la solución

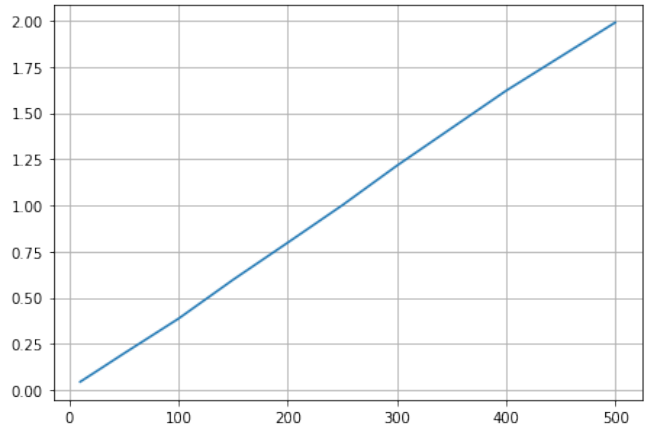
Luego, se hizo el análisis de mejora en la solución en función de la temperatura inicial. En este análisis se quiere ver qué tanto mejora porcentualmente la solución final del algoritmo respecto a la secuencia de productos inicial (generada aleatoriamente). Se calculó la mejora para temperaturas iniciales de [10, 25, 50, 100, 150, 200, 250, 300, 400, 500] para un tamaño de orden constante de 15 productos. Para eliminar las variaciones debido a la generación aleatoria de las órdenes se ejecuta 50 veces el temple simulado con cada temperatura inicial. Luego se saca el promedio de las mejoras para cada set de 50 ejecuciones de cada temperatura inicial.

```
ITER = 50
N = 15

t0 = [10, 25, 50, 100, 150, 200, 250, 300, 400, 500]

improvement = []
time_exec = []

for temp in t0:

    improvement_avg = 0
    time_avg = 0

    for i in range(0, ITER):

        pos_pick = [randint(0, 31) for i in range(0, N)]

        coordenadas = map_to_coord(pos_pick)

        energy_start = distance(coordenadas)

        start = time()
        _, energy_end = temple_simulado(coordenadas, temp, neighbours_annealing, distance)
        end = time()

        time_avg += end - start
        improvement_avg += (energy_start - energy_end)/energy_start

    improvement_avg = improvement_avg/ITER
    improvement.append(improvement_avg)

    time_avg = time_avg/ITER
    time_exec.append(time_avg)
```

En el gráfico se observa, en abscisas, las temperaturas iniciales, las cuales son equivalentes a las iteraciones del algoritmo debido al perfil lineal del decrecimiento de la temperatura. En ordenadas se obtiene la mejora porcentual **promedio** del costo final de la solución respecto al costo de la orden generada aleatoriamente.

```
fig, (ax0, ax1) = plt.subplots(1, 2, figsize=(16, 5))
ax0.plot(t0, improvement)
ax0.grid()
ax1.plot(t0, time_exec)
ax1.grid()
```





$mejora = \frac{(energia(inicial) - energia(final))}{energia(inicial)}$

Este número indica cuánto mejoró la solución final, en promedio, respecto a la inicial en función de la temperatura inicial. Se puede observar que a partir de una t0 de 200 se obtienen resultados suficientemente aceptables. Duplicar la temperatura inicial significa duplicar el tiempo de ejecución pero esto no se ve reflejado en una mejora considerable en la calidad de la solución. Con 200 iteraciones ya podemos considerar que la solución provista es buena.

Implementación de algoritmo genético para optimizar la ubicación de los productos en el almacén

En este problema se implementó un algoritmo genético para resolver el problema de optimización de la ubicación de los productos en un layout definido del almacen, de tal forma de optimizar el picking para un conjunto de ordenes de productos habituales.

Análisis del algoritmo

El algoritmo es más específicamente un algoritmo genético con permutaciones. Para la implementación fue necesario predifinir diferentes etapas:

- **Individuos**:

A la hora de definir individuos se decidió hacer una especie de "mapeo" sobre el layout original. Es decir, si el layout es para 32 productos, cada individuo será una lista de números de 0 a 31 ordenados de tal manera que el índice dará su posición. Por ejemplo, si un individuo es [5, 2, 4, 0, 1, 3] el producto 5 estará ubicado donde antes estaba el producto 0.

- **Función de fitness**:

Para calcular el fitness de un determinado individuo, se realiza un **recocido simulado** (ya implementado en el ejercicio anterior) para cada orden en el conjunto de órdenes habituales, para las cuales se desea optimizar la ubicación de los productos. Si recordamos, la función donde se implementó el recocido simulado recibe como parámetros la lista de productos a recolectar, la temperatura inicial, la función de generación de vecinos y la heurística a utilizar. Debido a que el objetivo del algoritmo genético es justamente cambiar la ubicación de los productos en el almacen, y el recocido simulado se ejecuta para el layout original, en lugar de pasar el número de producto en la secuencia, se pasa su índice en la lista (que es en definitiva el lugar donde se encuentra).

Entonces, para cada orden del conjunto se ejecuta el recocido simulado, que a su vez ejecuta internamente el algoritmo **A***. Como resultado se obtiene el costo de camino para esa orden, dada el layout que es el individuo, y con el picking optimizado por el recocido simulado. En teoría, es el menor costo posible al buscar todos los productos de la orden en éste layout.

El fitness final del individuo será un promedio de los costos para cada orden.

- **Selección**:

A la hora de elegir un individuo de la población previo al *crossover*, se utilizó **roulette wheel selection**. La idea general es que la selección es aleatoria pero los individuos tienen mayor probabilidad de ser elegidos cuanto menor sea su *fitness* (recuerdese que lo que se desea es minimizar el costo de *picking*).

- **Crossover**:

Se eligió **cruce de orden** como procedimiento de cruce por ser apto para algoritmos genéticos de permutación. Consiste en seleccionar dos puntos de cruce al azar, se copian los elementos de los padres entre los puntos de cruce, cruzados, y luego se copian los valores restantes a partir del segundo corte, en orden, evitando duplicar valores.

- **Mutación**:

Se eligió **inserción** también por ser apto para el problema de permutaciones. Consiste en elegir 2 genes al azar de la solución, se mueve el segundo gen elegido a continuación del primero y el resto de los genes se mueven a la derecha.

- **Condición de corte**:

La condición de corte del algoritmo es por cantidad de iteraciones (generaciones), y no por convergencia.

Evaluación de rendimiento del algoritmo

Reducción de fitness

La primer evaluación que se realizó, y más intuitiva, fue la reducción del fitness del mejor individuo de la población a lo largo de las generaciones (iteraciones) del algoritmo.

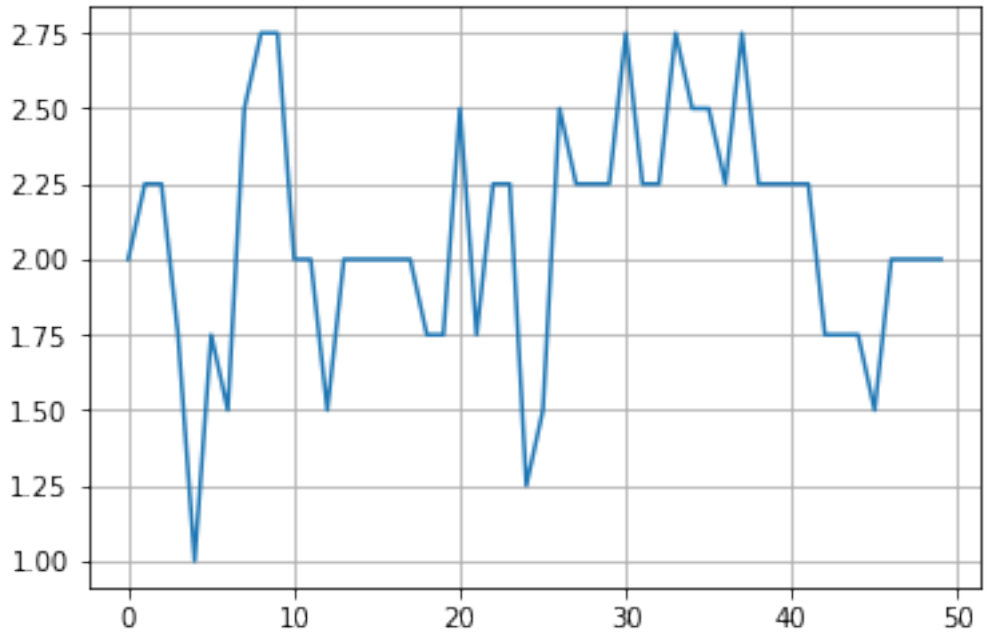
Se creó un conjunto de ordenes aleaorias (conjunto para el cuál se debe optimizar el almacén), de tal forma que ciertos productos tengan una mayor tendencia a aparecer. Esto es a modo de evaluación visual, intuyendo que los productos que más se repiten tenderían a aparecer ubicados más cercas de donde se comienza el *picking*.

```
from genetic import genetic, generate_ind
from matplotlib import pyplot as plt
from random import randint, choices

N_POB = 10 # Cantidad de individuos en la población
MAX_LENGHT = 32 # Cantidad de estanterías
T_0 = 200 # Temperatura inicial a la que inicia el algoritmo de temple simulado
MAX_GEN = 50 # Máxima cantidad de iteraciones a la que corta el algoritmo genético

conjunto = []

prob = []

estant = [i for i in range(0, MAX_LENGHT)]

for i in range(0, 25):
    prob.append(0.1)

for i in range(25, MAX_LENGHT):
    prob.append(0.5)

for i in range(0, 8):
    conjunto.append(choices(population=estant, k=randint(3, 7), weights=prob))

print(conjunto)
```

```
[[30, 30, 12, 24], [30, 25, 5, 31, 25, 19, 10], [18, 1, 27, 29, 30], [3, 14, 10, 27, 31], [8, 9, 31, 29, 22,
```

```
start = []
for i in range(0, N_POB):
    start.append(generate_ind())

best, history = genetic(start, conjunto, hist=True, status=False)
print(best)

plt.plot(history)
plt.grid()
```

```
[[26, 8, 29, 20, 17, 24, 30, 4, 25, 18, 27, 13, 10, 15, 5, 7, 31, 9, 2, 28, 14, 21, 3, 11, 6, 19, 23, 1, 12,
```



Como se puede observar en el gráfico, esta manera de evaluar no da resultados conclusivos del funcionamiento del algoritmo, esto podría deberse a que la diferencia de *fitness* entre el estado inicial y el óptimo no es la suficiente y al alcanzarse un mínimo este *fitness* comienza a oscilar.

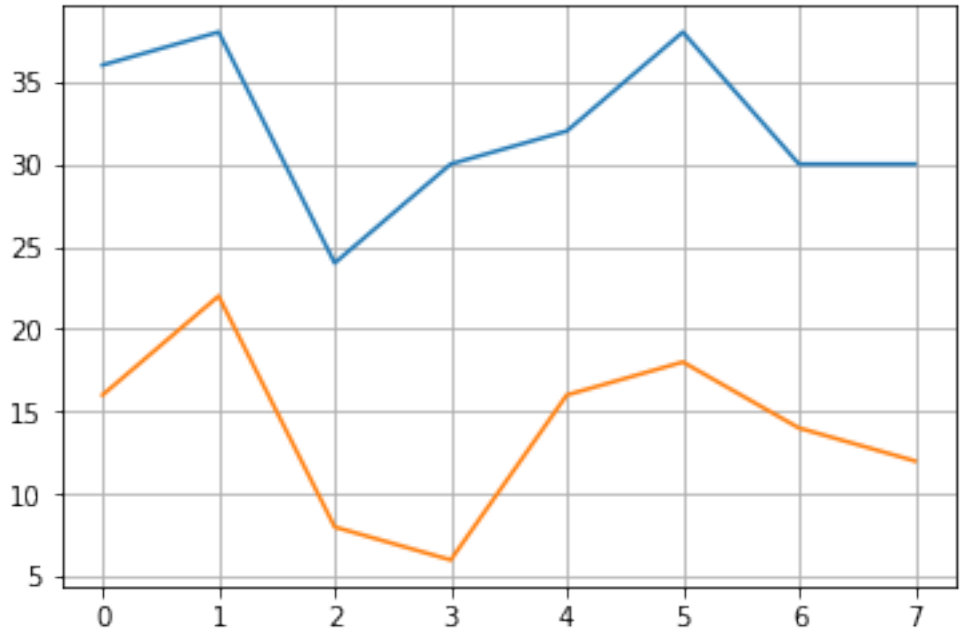
Comparación de costos con layout optimizado y sin optimizar

Dado que el análisis anterior no brinda resultados determinantes sobre el rendimiento y funcionamiento del algoritmo, se buscó otra forma de verificar que el algoritmo efectivamente funciona.

Se plantea ejecutar el recocido simulado para una determinada orden de productos de tal forma de optimizar la secuencia del *picking* esté optimizada para el layout original del almacén. Luego ejecutar el algoritmo genético para obtener el layout óptimo para dicha orden de productos. Y por último, ejecutar el recocido simulado para la misma orden pero esta vez con el layout del almacén optimizado.

```
from simulated_annealing import map_to_coord, temple_simulado, neighbours_annealing, distance

cost0 = []
cost1 = []

for orden in conjunto:
    coordenadas = map_to_coord(orden)
    _, costo = temple_simulado(coordenadas, 200, neighbours_annealing, distance)
    cost0.append(costo)

    # Generación de población inicial
    start = []
    for i in range(0, N_POB):
        start.append(generate_ind())

    new_layout, _ = genetic(start, [orden], status=False)

    orden_estant = []
    for prod in orden:
        orden_estant.append(new_layout.index(prod))

    coordenadas = map_to_coord(orden_estant)
    _, costo = temple_simulado(coordenadas, 200, neighbours_annealing, distance)
```

```
    cost1.append(costo)

plt.plot(cost0, label="costo con layout sin optimizar")
plt.plot(cost1, label="costo con layout optimizado")
plt.grid()
```




```
avg = 0
for i, j in zip(cost0, cost1):
    avg += (i - j) / i
avg = avg * 100 / len(cost0)

print(avg)
```

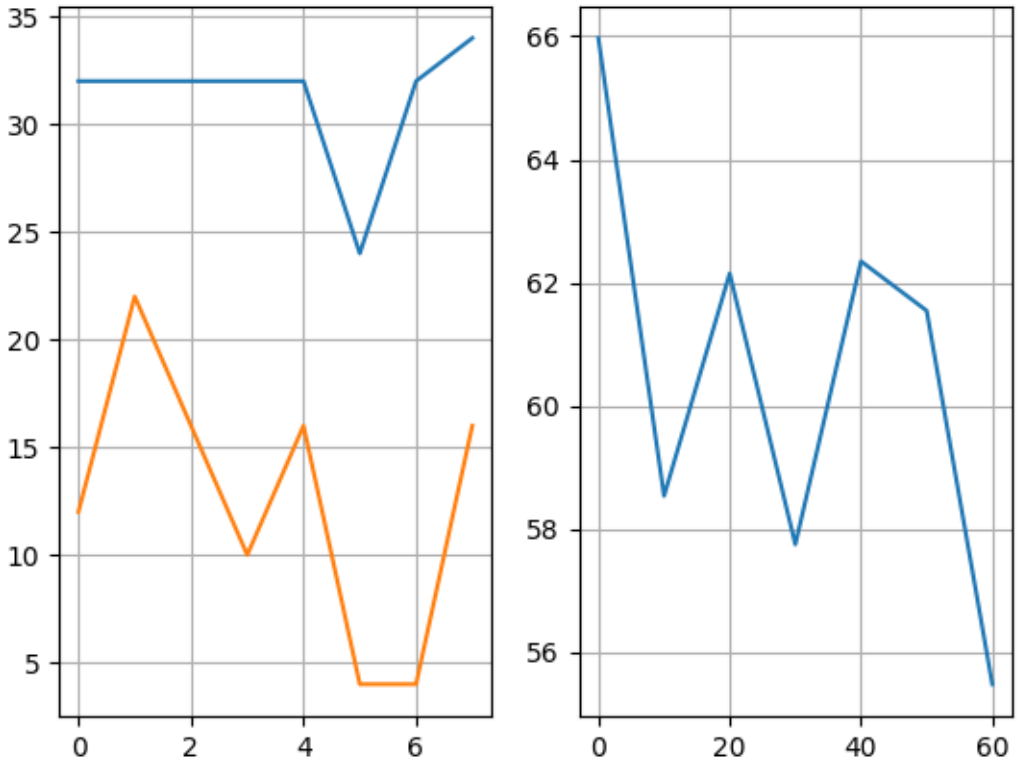
```
56.275638544891635
```

En el gráfico anterior pueden observarse diferencias claras al optimizar la ubicación de los productos en el almacen para un conjunto de ordenes, por lo que puede concluirse que el algoritmo está funcionando correctamente, reduciendo costos en aproximadamente **56,27%**.

Análisis de la probabilidad de mutación

En genetic-test.py se realiza un análisis para encontrar un valor aceptable de probabilidad de mutación. Esto se obtiene ejecutando el análisis anterior y comparando la reducción promedio obtenida del conjunto de ordenes.

Los resultados obtenidos son los de la gráfica:



Implementación de algoritmo de satisfacción de restricciones

Se implementó un algoritmo de *backtracking* para resolver el problema de satisfacción de restricciones, más especificamente, de *scheduling*.

Modelado del problema

El problema consiste en una determinada cantidad de tareas que deben realizarse, donde cada una demanda un determinado tiempo:

```
T = {"d0": 40, "d1":20, "d2":25, "d3":10, "d4":30}
```

Como restricción global a la hora de asignar valores, se considera desde un principio (acotando el dominio de las variables) que el proceso total debe finalizarse en un tiempo menor a *DLINE*. Además, el tiempo se discretizó en intervalos con valor *STEP*.

```
DLINE = 150
STEP = 5
```

Por lo tando, la asignación de los valores en el dominio de cada variable es de la forma:

```
D = {}
for i in T.keys():
    D[i] = list(range(0, DLINE-T[i], STEP))
```

Luego se crea un diccionario vacío que contendra las futuras asignaciones del algoritmo, y se elige una variable aleatoria no asignada (por ser la inicial se elige aleatoriamente de entre todo el conjunto de variables).

```
from backtracking import selection
```

```
assign = {}
print(selection(D, assign))
```

```
d0
```

Es importante tener en cuenta que a la hora de elegir una variable no asignada se puede implementar alguna heurística como *mínimos valores restantes*. En la práctica esto no se realizó.

Restricciones binarias de precedencia: Las restricciones binarias se describen *harcodeando* la relación de precedencia entre las distintas variables, teniendo en cuenta el orden.

A continuación se plantea un grafo de restricciones sencillos para que puede verificarse visualmente la solución.

```
R2 = {
    "d0":[("d0", "d1"), ("d0", "d2")],
    "d1":[("d0", "d1"), ("d1", "d3")],
    "d2":[("d0", "d2"), ("d2", "d4")],
    "d3":[("d1", "d3"), ("d3", "d4")],
    "d4":[("d3", "d4"), ("d2", "d4")]
}
```

Restricciones binarias de recursos: Cada tarea ocupa máquinas que están disponibles en cantidad limitada, por lo tanto no pueden ejecutarse tareas que requieren de la misma máquina en paralelo.

Este tipo de restricciones se describen asignando una lista de máquinas requeridas a cada tarea.

```
M = {
    "d0":["m1", "m3"],
    "d1":["m2", "m4"],
    "d2":["m0"],
    "d3":["m0", "m2", "m3"],
    "d4":["m0", "m4"]
}
```

Solución: La solución será el diccionario de tareas, con el valor de tiempo inicial en el que se ejecutará cada una.

```
from backtracking import backtrack

print(backtrack(T, assign, D, R2, M))
```

```
{'d4': 75, 'd3': 60, 'd2': 40, 'd0': 0, 'd1': 40}
```

Conclusión: El algoritmo por ser *backtracking* básico, sin la implementación de algún otro algoritmo para analizar arco consistencia como *AC-3*, es muy ineficiente pero da soluciones correctas.